InQuanto™

Release 2.1.1

Quantinuum

May 02, 2023

INTRODUCTION

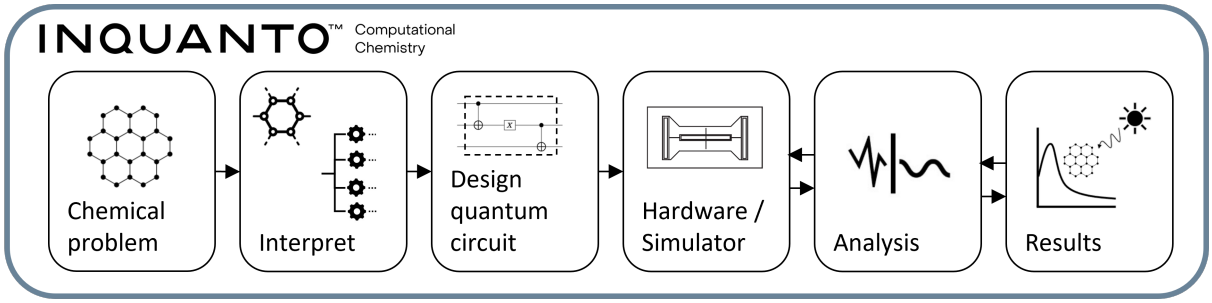
QUANTINUUM

INQUANTO™ Computational Chemistry

This user guide is for **InQuanto license holders** and details how to use the quantum computational chemistry package InQuanto, developed and maintained by Quantinuum. For all enquiries, please contact inquanto@quantinuum.com .

How to use this documentation

InQuanto contains a broad suite of methods for applying quantum algorithms to chemical problems. In this guide we provide an overview of this functionality, with a focus on getting users up and running with the most popular methods. For some users, the *quickstart guide* will be the best starting point, but we also provide detailed *installation instructions*, *tutorials*, and extensive *API documentation*.



CHAPTER

ONE

WHAT IS INQUANTO?

InQuanto is Quantinuum's state-of-the-art Python-based quantum computational chemistry platform. It is designed to facilitate quantum computational chemistry for chemical researchers in industry and academia, and to provide an ecosystem for quantum researchers to develop and implement novel algorithms for chemical problems.

1.1 Why use InQuanto?

Computational chemistry aims to accurately model the behavior of quantum particles inside of molecules and materials. Modelling these electrons and nuclei allows one to evaluate, from first principles, chemically meaningful properties such as bond energies or reaction rates. These particles are intrinsically quantum, and have properties that are challenging to compute accurately on classical computers. The most accurate classical methods have only been applied to tens of atoms, even on the largest computing resources, but chemically meaningful molecules often have thousands of electrons. Quantum computers are predicted to be better at storing and manipulating highly entangled states, such as systems of interacting electrons, than classical computers. Consequently, chemistry is generally considered to be among the first fields in which quantum computing can outperform classical computing, unlocking accurate modelling of larger, more complex systems. This would be an example of quantum advantage.

Whilst InQuanto contains a broad set of tools for quantum computational chemistry, it has also been developed and deployed to support collaborations with industry partners to ensure that there are robust, practical algorithms for calculating chemical quantities on Noisy Intermediate-Scale Quantum (NISQ) devices. For example, we have shown how InQuanto can be applied to carbon capture in metal-organic frameworks, utilizing NISQ experiments and an efficient fragmentation scheme to model dissociation. [1] A more complete list of examples publications is available online.

Looking beyond the NISQ era, InQuanto is scoped for the regime of fault-tolerant quantum computation in a number of ways. Firstly, current algorithms will become easier to evaluate and give more precise answers. Secondly, specifically fault-tolerant algorithms are currently in development in the context of chemistry, such as quantum phase estimation. InQuanto is also coupled to, and will take advantage of, other research work in Quantinuum on fault-tolerant methods, such as quantum signal processing, [2, 3] to improve the capabilities and performance of chemical calculations. Additionally, building on top of TKET ™ means users can easily switch between, and experiment with, different quantum hardware and quantum simulator, allowing easier pivoting to the best devices.

1.2 How it works

InQuanto is designed to support the complete quantum computational chemistry pipeline, therefore it has tools to process several steps. A simplified version of these steps is presented in Fig. 1.1, and we relate this to the codebase in the *manual introduction*.

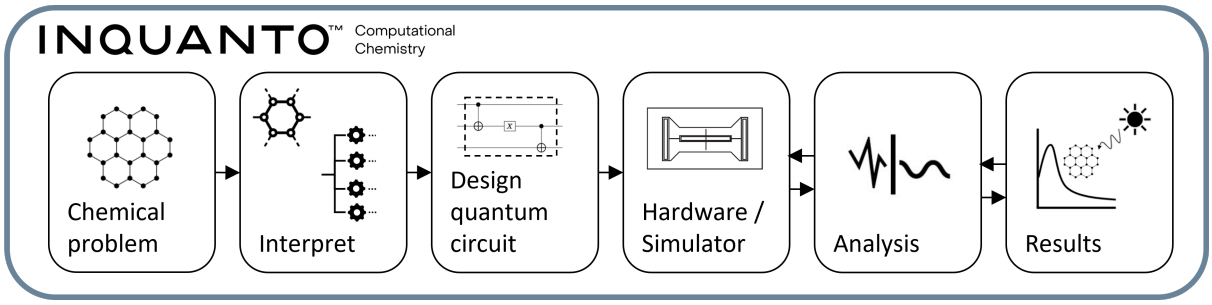


Fig. 1.1: Schematic overview of the InQuanto platform.

Chemical problem

Firstly, the user defines the **chemical problem**, which is given by the physical quantity of interest and the atomic structure of the system. For example, one could be interested in modelling protein binding strengths or chromophore excitation.

InQuantize

Next, the chemical problem must be **InQuantized**, or interpreted as an appropriate quantum computational problem. For example, excitation calculations can be performed by representing the electronic states using X and the Hamiltonian using Y, solving the problem using the quantum algorithm Z. This is where InQuanto excels: in creating efficient representations of chemical problems and offering a range of quantum algorithms and methods.

Circuit construction and experimentation

Thirdly, the quantum problem is compiled into **quantum circuits**. InQuanto utilizes TKET™, Quantinuum's quantum SDK, to a) further optimize the constructed circuits (reducing depth, and complexity), and b) to deploy the circuits to a wide range of quantum **hardware and simulator** options.

Analysis

When an experiment/simulation is complete, measurements are collected. These measurements need to be **analyzed** to be related back to the chemical problem. This may include error mitigation steps. Often there is a loop between analysis and further experimentation/simulation, for example in variational quantum algorithms.

Results

When the algorithm and analysis is complete, post-processing yields **results**, such as binding energies or spectra.

Throughout the whole pipeline, InQuanto has tools for analysing and limiting the amount of error that occurs in Noisy Intermediate-Scale Quantum devices. Whilst InQuanto includes streamlined routines, it is a modular Python toolbox which allows scientists to easily mix and match components and build their own research scripts.

1.3 Powered by TKET ™

Current quantum computers have limited capabilities. In order to exploit them efficiently, several considerations have to be made. First, in general it is desirable to reduce the circuit depth and complexity to reduce the amount of noise that is accumulated during circuit processing. However, this must be done without changing the accuracy of the circuit. There are also unique intricacies to each quantum architecture. For example, different operations may have more or less noise, or the device may have restricted connectivity between qubits.

To adapt the quantum circuits to difference devices we use TKET ™. Among other tools, TKET ™ has routines for: efficient qubit routing for device architecture, mapping circuits to different gate sets, and finding shortcuts in circuit structures. Utilizing TKET ™ underneath InQuanto allows one to focus on the chemistry and deploy effectively to a variety of backends without thinking too much about technical details.

Users can also take control of many InQuanto objects (e.g. circuits) using native pytket methods. pytket's documentation can be found here, and the list of devices and backends that InQuanto can interface with through pytket-extensions can be found here. In the *hardware tutorials* we present examples of using pytket-extensions to interface with difference devices, such as through using pytket-qiskit.

CHAPTER

TWO

INSTALLING INQUANTO

InQuanto and the associated extensions are distributed as a set of Python packages, therefore the below steps can be adapted for any popular package managers such as Conda, Poetry or other virtual environments. Please ensure you meet the *system requirements*.

As part of these instructions, **you will require your provided InQuanto API-KEY**. For all enquiries, please contact the InQuanto team.

1. Install InQuanto

The command below installs InQuanto, InQuanto-Extensions, and all the optional extra packages required to run the tutorials and examples contained in our documentation. A more minimal installation is possible by omitting the extra flags inside the square braces.

```
pip install "inquanto[ibm, quantinuum, jupyter, qulacs, projectq]" inquanto-pyscf␣
↪inquanto-nglview -i https://dl.cloudsmith.io/SkInhKAV92sMAuDe/quantinuum/customer/
↪python/simple/
```

Update InQuanto

Following any updates to InQuanto and extensions, the packages can be upgraded with:

```
pip install "inquanto[ibm, quantinuum, jupyter, qulacs, projectq]" inquanto-pyscf␣
↪inquanto-nglview --upgrade -i https://dl.cloudsmith.io/SkInhKAV92sMAuDe/quantinuum/
↪customer/python/simple/
```

2. Activate your InQuanto License

In a terminal, input the following command

```
python -c "from inquanto import activate_inquanto_license; activate_inquanto_license()
↪"
```

You will be prompted to provide your InQuanto API-KEY and a name for your current device.

Next, store your InQuanto API-KEY in the system keyring with:

```
python -c "from inquanto import store_inquanto_credentials; store_inquanto_
↪credentials()"
```

InQuanto should be ready for use. If you encountered any problems during installation, check out the *troubleshooting guide*. Otherwise, try the *Quick-start guide*.

2.1 System Requirements

2.1.1 inquanto

Python	3.9, 3.10
OS	Linux, macOS 11, WSL, Windows 10, Windows 11

2.1.2 inquanto-pyscf

Python	3.9, 3.10
OS	Linux, macOS 11, WSL
Runtime libraries	libomp, lapack

Note: Runtime libraries can be installed using the Linux distribution package manager. For macOS, the dependencies can be installed using Homebrew or conda (via conda-forge).

2.1.3 inquanto-nglview

Python	3.9, 3.10
OS	Linux, macOS 11, WSL, Windows 10, Windows 11

2.2 Troubleshooting

2.2.1 Licensing

No 'keyring' backend was found

Some Unix distributions do not have a default credentials backend. The `keyrings` package InQuanto uses to interact with the system has a list of available alternatives. We recommend installing `keyrings.alt` with:

```
pip install keyrings.alt
```

After installing the backend, follow the *installation steps* again to correctly store the InQuanto license.

Errors associated with license activation or use

Please contact InQuanto support, providing any traceback and the machine ID. The machine ID can be obtained with:

```
python -c "from inquanto import get_machine_id; get_machine_id()"
```

2.2.2 inquanto-pyscf

Missing .dylib files on macOS

PySCF requires working installations of LAPACK and OpenMP. The most reliable method is using Homebrew to install these packages (on Mac) at the user level and removing any other installations including virtual environments (conda); this makes lapack and libomp available to any Python installations and virtual environments.

CHAPTER

THREE

QUICK-START GUIDE

On this page we give a quick example of running a quantum computational chemistry calculation using InQuanto. This example evaluates the H_2 electronic wave function in the minimal basis using the *Unitary Coupled Cluster Singles Doubles Ansatz* and finding parameters using the variational quantum eigensolver and a state vector (noiseless) backend.

3.1 Express database

Most quantum chemical calculations on a quantum computer start with a classical computation of molecular integrals. In general this is achieved in InQuanto via separate *extensions*, which interface to external classical quantum chemistry packages (for example *inquanto-pyscf*). However, InQuanto also has a small internal database as part of its `express` module. This database contains molecular Hamiltonians, as well as other useful information, for a variety of small systems. These *examples* can be easily loaded and provide a simple way to start using InQuanto functionality.

In the cell below, we import the `load_h5()` function which we then use to load the H_2 STO-3G example data. We then inspect its Hamiltonian terms, and print its classically calculated CCSD energy, which is stored for easy comparison in the express database entry.

```
from inquanto.express import load_h5
h2_sto3g_data = load_h5("h2_sto3g.h5", as_tuple=True)
hamiltonian = h2_sto3g_data.hamiltonian_operator
print(hamiltonian.to_FermionOperator())
print(h2_sto3g_data.energy_ccsd)
```

```
(0.7430177069924179, ), (-1.2702927243904387, F0^ F0 ), (-0.45680735030940944, F2^ F2␣
↪), (-1.2702927243904387, F1^ F1 ), (-0.45680735030940944, F3^ F3 ), (0.
↪4889085974504739, F2^ F0^ F0  F2 ), (0.4889085974504739, F3^ F1^ F1  F3 ), (0.
↪680061857584128, F1^ F0^ F0  F1 ), (0.6685772770134896, F2^ F1^ F1  F2 ), (0.
↪1796686795630157, F1^ F0^ F2  F3 ), (-0.1796686795630157, F2^ F1^ F0  F3 ), (-0.
↪1796686795630157, F3^ F0^ F1  F2 ), (0.17966867956301566, F3^ F2^ F0  F1 ), (0.
↪6685772770134896, F3^ F0^ F0  F3 ), (0.7028135332762816, F3^ F2^ F2  F3 )
-1.1368465754747643
```

3.2 VQE wrapper function

In the manual sections we demonstrate how a user can build a variety of tools for performing quantum chemistry using InQuanto, but here we want a simple intuitive example that "just runs". To do this we will use the *run_vqe()* function from express to run a *variational quantum eigensolver algorithm*.

run_vqe() requires the user to provide: an *ansatz* , the *Hamiltonian operator*, and a pytket backend. Optionally, the user can also choose whether *run_vqe()* uses gradients, what flavour of classical *minimizer* strategy to use, and the starting parameters for the variational cycle. These are not required options, and when left undefined in the example below will default to i) using gradients, ii) using the Scipy L-BFGS-B minimizer, and iii) starting with symbol parameters all set to zero.

As stated, *run_vqe()* requires the Hamiltonian, an ansatz, and a backend. To construct these we take the stored Fermionic operator data and qubit encode it using the Jordan-Wigner mapping (more *here*). Then we prepare a 4 qubit ansatz circuit (corresponding to the 4 spin-orbitals of H_2 STO-3G) by defining the FermionSpace and FermionState objects (more *here*) and feeding them into the *FermionSpaceAnsatzUCCSD* class (more:ref:*here <ucc>*). Lastly we import and instantiate a pytket state vector backend from Qiskit.

```
from inquanto.express import run_vqe
from inquanto.states import FermionState
from inquanto.spaces import FermionSpace
from inquanto.ansatzes import FermionSpaceAnsatzUCCSD
from pytket.extensions.qiskit import AerStateBackend

hamiltonian = load_h5("h2_sto3g.h5", as_tuple=True).hamiltonian_operator.qubit_
↪encode()

space = FermionSpace(4)
state = FermionState([1, 1, 0, 0])
ansatz = FermionSpaceAnsatzUCCSD(fermion_space=space, fermion_state=state)

backend = AerStateBackend()

vqe = run_vqe(ansatz, hamiltonian, backend)
print(round(vqe.final_value, 8))
print(vqe.final_parameters)
```

```
# TIMER BLOCK-0 BEGINS AT 2023-05-02 11:25:25.430576
# TIMER BLOCK-0 ENDS - DURATION (s):  0.1640598 [0:00:00.164060]
-1.13684658
{s0: 0.0, s1: 0.0, d0: -0.10723347230091546}
```

After the VQE algorithm has converged we can inquire its *final_value()* which is its total energy. In this UCCSD ansatz, this energy is equivalent to the CCSD energy printed above (~1.1368465 Ha). We can also examine the parameters of the ansatz terms by printing the *final_parameters()*. These show that the singlets (*s*) have no contribution due to be symmetry forbidden but the double (*d*) excited term has significant weight in the optimized wave function.

The above is just a quick example of how a user can run meaningful quantum computational chemistry calculations easily using InQuanto. Whilst still using *run_vqe()* there are plenty of variables to explore. For example, you can try loading in a different system from *express* with more electrons or orbitals, modify the FermionSpace and FermionState accordingly and run a bigger calculation. Alternatively, you could examine how setting `with_gradient=False` on *run_vqe()* or modifying its minimizer changes the time to converge. Or how about comparing the speed of different pytket state vector backends?

After you're comfortable with this quick-start guide, we recommend diving into the *manual* or following further *tutorials*.

Note: The *run_vqe()* method is not recommended for production purposes, only testing. The method only permits

state vector based pytket backends. For example one may use the *AerStateBackend* or *QulacsBackend* (see info here). As the `run_vqe()` method only permits state vector it streamlines the selection and generation of the *computables and protocols*. Specifically, under the hood, the Protocol, which provides instructions for circuit measurement and post-processing, is set to `ProtocolStateVectorSparse`, and the Computable is `ExpectationValue`. Again, under the hood, the computables are built (`build()`) and then ran (`run()`). For non-state vector calculations, one must utilize a proper instance of `AlgorithmVQE`, which is much more flexible than `run_vqe()` .

CHAPTER

FOUR

QUANTINUUM H-SERIES

Although InQuanto is hardware agnostic and can be used with a variety of backends, the use of the Quantinuum H1 generation of devices and emulator is recommended.

4.1 Quantinuum System Model H1

The Quantinuum System Model H1 generation of quantum computers are packaged with some InQuanto licences and are available over the cloud. The H1 devices feature high-fidelity, fully connected qubits, along with features such as mid-circuit measurement and qubit reuse. This aids researchers and developers in the construction and implementation of quantum circuits.

4.2 Getting Started

To get started on System Model H1, your organization's administrator needs to add you as an user on the Quantinuum Systems User Portal. Once you are added, you will receive an email notifying you that your account has been set up. Follow the instructions in the email for resetting the temporary email and signing into the portal. You need to sign in at least once to agree to the Terms & Condtions for hardware usage.

4.3 Documentation

Once you're logged into the user portal, you can find guides for system usage under the **Examples** tab. You can find these in the user portal by navigating to **Examples**, then clicking **docs** on the left-hand side.

The following guides are found in this tab:

- *Quantinuum System Model H1 Product Data Sheet*: specification and performance data of System Model H1
- *Quantinuum System Model H1 Product Data Sheet*: specifications of System Model H1 Emulator
- *Quantinuum Systems User Guide*: information on the currently available devices and workflow, including job queue, data retention, and looking at remaining credits

4.4 Use

The access to Quantinuum Systems is provided for InQuanto use only. You may not utilize Quantinuum Systems for any purposes other than in working with InQuanto.

4.5 H1 Emulator

Frequently, classical simulators are used to debug and optimize quantum algorithms applied to systems larger than those available to currently existing quantum hardware. They also allow for verification and validation of data and results generated by real quantum devices. Often, noiseless classical simulation is used, wherein the "device" is assumed to be free of physical error. This is useful for analysis on the algorithmic level, however may obscure noise-dependent performance characteristics. It also prevents an analysis of physical noise mitigation techniques.

In contrast to noise-free classical simulators, *emulators* include noise models derived from experimentation on a given quantum device. This allows for the generation of more realistic data and thus provides a better sense of how a quantum circuit will perform. The H1 Emulator – access to which is packaged with InQuanto – includes a noise model that mimics the operations of the H1 generation quantum computers. In addition to the real H1 devices, InQuanto is capable of easily interfacing with the H1 emulator, as demonstrated below.

4.6 Access

Examples using running on Quantinuum Hardware are found on the Hardware tutorials page. Further information on `pytket-quantinuum` can be found on the pytket-quantinuum page.

Run the following command to install `pytket-quantinuum` and use it with InQuanto:

```
pip install "inquanto[quantinuum]" -i <private-index-url>
```

4.7 InQuanto Administrators

4.7.1 Administrator Responsibilities

As an andministrator, you are responsible for the following:

- Adding users for Quantinuum Systems access on the Quantinuum Systems User Portal
- Ensuring that the number of users added to Quantinuum Systems access aligns with the number of seats purchased.
- Ensuring users access Quantinuum Hardware Systems for InQuanto use only. Users may not utilize Quantinuum Systems for any purposes other than working with InQuanto, unless a separate Quantinuum H-Series subscription has been purchased.

4.7.2 Getting Started as an Administrator

As your organization's designated administrator, you will be granted access to Quantinuum systems. You will receive an email notifying you that you have access to the Quantinuum Systems User Portal with a temporary password. Follow the instructions in the email for signing in and resetting your password. Once you do this, you are ready to start adding users for access with InQuanto.

4.7.3 Adding Users

You can find instructions for managing your organization's hardware access in the *Quantinuum Systems Administrator Guide* under the **Examples** tab on the Quantinuum Systems User Portal.

You can find this document in the user portal by navigating to **Examples**, then clicking **docs** on the left-hand side. Instructions are provided on how to add users as well as directiosn for managing credits and tracking usage.

CHAPTER

FIVE

HOW TO USE INQUANTO

This page describes how the components of InQuanto fit together to allow bespoke quantum computational chemistry calculations. More in-depth guides on each of these components are contained in the following pages. Whilst InQuanto can be used in a reasonably black-box manner, the user is recommended some familiarity with modern computational chemistry and quantum computational chemistry. Some minimal suggested reading is presented in the *bibliography*.

InQuanto is comprised of a number of Python modules that are designed to work interchangeably with one another. Here, to explain how these modules can come together, we begin by explaining their roles in relation to *algorithms*. The classes in this module, such as `AlgorithmVQE`, are the quantum and hybrid-quantum methods we apply to evaluate meaningful chemical quantities such as ground and excited states. InQuanto contains a wide variety of algorithms, the choice of which may be determined by the result of interest.

5.1 Chemistry workflows

As stated, `algorithms` contains high level classes which solve quantum computational problems, but the quantum computational problem solved needs to represent the chemical system and quantities of interest. In InQuanto we use algorithms to evaluate quantities in relation to *`computables`* (more below). These algorithms use pytket backends to drive the quantum computation, and may also need classical functions or data, such as *minimizers* or a set of `initial_parameters` to aid solution. In the figure below we show how the key objects in InQuanto can be brought together.

From the figure above we emphasize how *computables* are built to contain quantum circuits representing *a*) our system, *b*) the operations to apply to give the quantity of interest, and *c*) instructions for measuring and interpreting quantum measurements (`protocols`). A basic example of a computable is the *`ExpectationValue`* of a quantum state, such as the total energy of the system as reported by the Hamiltonian.

Computables are a quantum computational form of a physical quantity which can be computed - for example an expectation value, it must to fed to the algorithm in terms of qubit states and qubit operations. To reach this form, one must InQuantize the chemical system. This refers to how InQuanto takes a chemical system, defined by the atomic coordinates or some model, and constructs its set of qubit states and operators. Broadly, this can be considered to split into two steps, *i*) the preparation of mean-field quantities, and *ii*) selecting and representing the electronic system.

Note:

InQuanto's main focus is solving the electronic structure problem of molecules within the Born-Oppenheimer (frozen nuclei) approximation. This is often described using second quantization, such that the Hamiltonian is:

$$\hat{H} = \sum_{i,j=0}^{N} h_{ij} a_i^\dagger a_j + \frac{1}{2} \sum_{i,j,k,l=0}^{N} h_{ijkl} a_i^\dagger a_j^\dagger a_k a_l \tag{5.1}$$

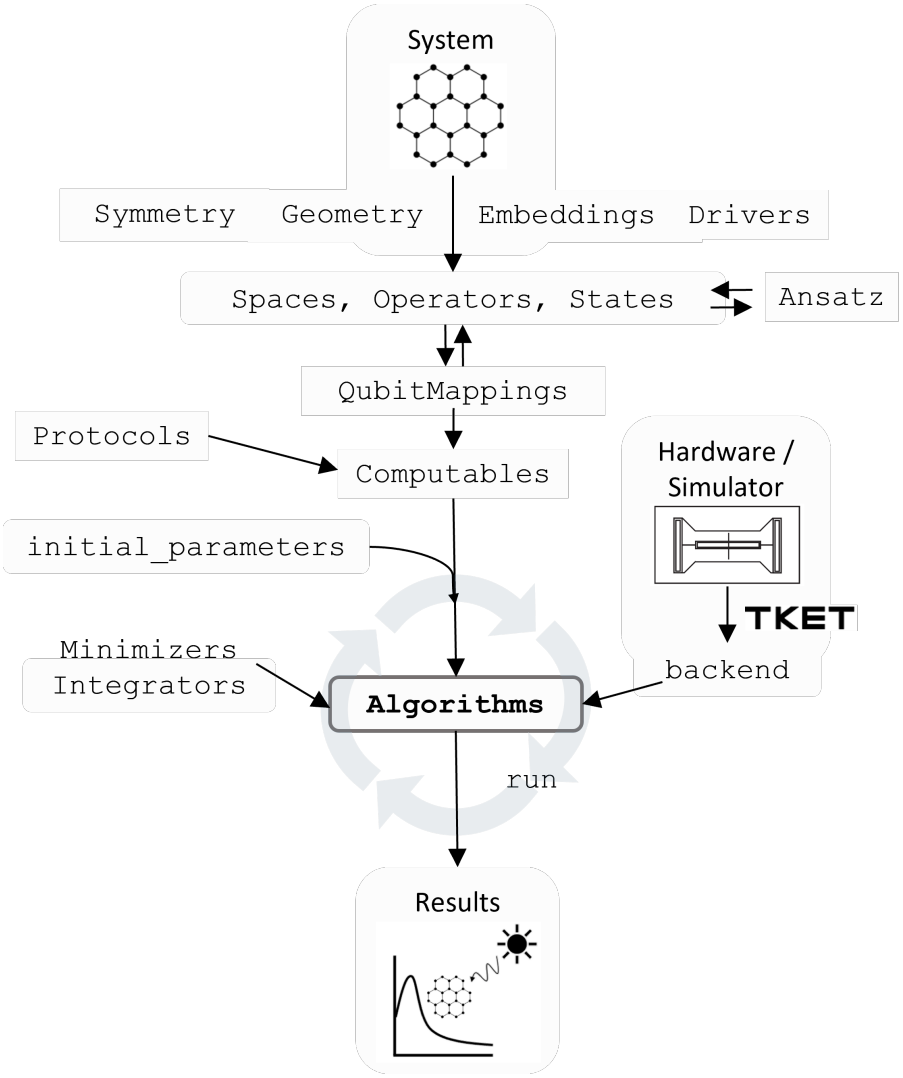


Fig. 5.1: Schematic overview of the InQuanto workflow.

where $a^\dagger$ and a are the Fermionic creation and annihilation operators and h_{ij} and h_{ijkl} are the one- and two- body electronic integrals respectively. The integrals are obtained from mean-field calculations, such as Hartree-Fock. For more details see e.g. [4, 5]

5.1.1 Preparing mean-field systems

Starting from a specification of molecular geometry or an atomic structure file (i.e. *mol.xyz*), one begins processing the molecule/material using the *geometry*, *symmetry* , *embeddings* modules and *drivers*. Drivers run mean-field calculations with small classical computational overhead, such as Hartree-Fock. The *geometry* module takes the structure and loads it into InQuanto. *Symmetry* contains a set of tools for reducing the computational complexity of chemical systems at the structural, electronic, and qubit levels. *Embeddings* (e.g. DMET) allow one to focus computational effort on a part of the system, reducing the overall cost, and `Drivers` are used to run classical computational chemistry calculations to construct components. These `Drivers` are generally provided by *InQuanto Extensions* but there are also model Hamiltonians and data in *`express`*.

5.1.2 Spaces, operators, states, and mappings

When the mean-field system is defined and the classical components calculated, one can specify the *Spaces, Operators, and States* of the system and perform *Qubit Mappings* . For example, systems of correlated electrons are modelled using some electronic Hamiltonian operator which acts on a Fermionic Hilbert space, with the state often being defined by a set of occupation numbers. In InQuanto we can construct these spaces, operators, and states and then convert them to qubit spaces, qubit operators, and qubit states.

Many quantum algorithms for quantum chemistry require the preparation of an *ansatz state* , which may be parametrized. These ansatzes are educated guesses for the state of the chemical system. InQuanto represents the generation of quantum circuits necessary for a variety of ansatzes using the *ansatz* classes.

5.1.3 Running computables and algorithms

When the qubit objects have been prepared, they are fed into the *Computables* and we are ready to focus on exactly how the quantum computation is performed. In addition to the InQuantized chemical system, we also add Protocols to the *Computables*. Protocols add instructions for how to measure the quantum circuit, and relate that measurement to the computable of interest. Protocols are also where noise mitigation capabilities may be provided, for example based on symmetry using PMSV.

The completed *Computables* are fed into the algorithm along with some complementary `initial_parameters`. The initial parameters for the qubit states vary between algorithms. Similarly, the form of the 'solver' also varies between algorithms. If performing a variational algorithm then the preference is for an efficient *Minimizers*, whilst time-evolution algorithms require *Integrators*

The last component we need to `build` an `algorithms` or *`computables`* class is a and pytket-extensions to access a broad range of quantum computational hardware and simulators. TKET ™ optimizes the circuit for performance, can deliver the circuit to the hardware or simulator (when provided credentials), and will collect the processed circuit results to pass to InQuanto.

Having provided the `algorithms` with the necessary input, all that remains is to `run` the circuit. This will automatically pass the information from the built `algorithms` to the backend, queue then run the experiment, and return results. The `algorithms` object can then be inquired for results, for example `algorithm.final_values`, which correspond to the computable and gives the chemical quantity of interest.

5.2 Expert use of InQuanto

InQuanto is a modular Python toolbox which allows scientists to easily mix and match components and build their own research scripts. For expert quantum computational chemistry users there are a couple of useful tips. Firstly, there are a number of fully customizable classes which allow users to construct objects from scratch or modify prebuilt objects. For example `QubitMapping` can be used to build your own mapping, or `operators` classes have many tools for adding, removing, or manipulating terms.

Ultimately, InQuanto constructs, runs, and interprets tket circuits. Therefore expert users can choose to obtain the circuit objects from InQuanto using functions such as `generate_circuits()` or `get_circuit()` for further manipulation using the pytket stack, such as manually appending gates. It is also possible to inject pytket native objects into InQuanto, such as passing a custom built compiler pass objects to computables. Due to its modular nature, these modifications can be made and become part of an InQuanto workflow.

CHAPTER

SIX

ALGORITHMS

InQuanto aims to facilitate the use and development of quantum algorithms for the simulation of quantum chemistry. Over recent years, this field has grown dramatically, and research efforts in this regard are expected to intensify as the capabilities of quantum hardware grow. These efforts are reliant on both studies in method development and in the application of known techniques to characterise their behaviour in meaningful contexts. While the lower-level functionality available in InQuanto is appropriate for the former of these, the latter may require a higher-level approach. The `inquanto.algorithms` module provides such high-level capabilities for running various predefined algorithms.

There are a variety of algorithm classes built into InQuanto - most of which are discussed below, with a full list provided in the API reference. Although the settings and input data differs between algorithms, they share a common structure in their usage. This structure is demonstrated in Fig. 6.1.

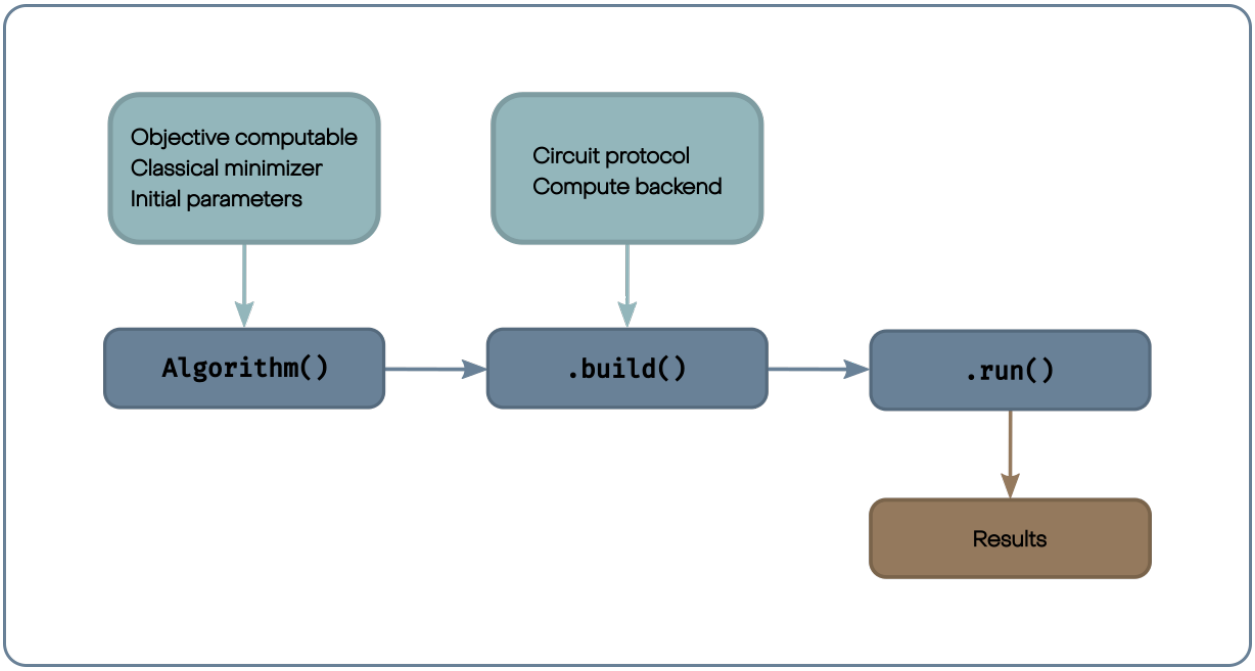


Fig. 6.1: An example of the usage of a high-level algorithm class in InQuanto. System and algorithmic details are provided to the `Algorithm` instance at construction, with compilation and simulation details provided when calling the `.build()` method. The `.run()` method is used to perform the computation. Although this process is followed for all algorithms, different algorithms will have different inputs at each step.

Firstly, the instance of the algorithm class must be created, with specification of problem and system details. These details correspond to the higher-level, abstract properties of the algorithm to be performed. For example, they may describe which expectation value is to be calculated, or which classical minimizer to use for a variational optimization. Note that this set of parameters does not include details with regards to circuit compilation or simulation - these details are

provided to InQuanto further down the pipeline. Although there is much overlap between algorithms, the precise types of input required here depends on the algorithm to be simulated; a discussion of the requirements for each algorithm is provided in the sections below.

After the `Algorithm` instance is created, it must be *built*. Building an algorithm in InQuanto refers to the building of the quantum circuits and surrounding computational processes necessary to run the algorithm. Again, the necessary parameters provided to the `build()` method are dependent on the algorithm in question, and specific details are provided below for each algorithm class. Typically, these involve providing the computational backend of choice to be used. Different simulation strategies - whether using classical simulation, an emulator of quantum hardware, or a quantum device itself - will require different backends. The `build` process also will typically require the specification of one or more *Protocols*. These objects instruct InQuanto in how to compile the quantum circuits themselves. The availability of various protocols will depend on the computational backend used, and on the algorithm which is to be performed.

Having constructed and built the `Algorithm` object, the algorithm can be executed using the `run` method. Typically, no further input is required at this stage. It is separated from the previous steps due to the fact that the heaviest computational burden (in the case of classical simulation of a quantum device, exponentially so) occurs in this step. This step performs the algorithm using the specified computational backend. Once this step is complete, results may be obtained from the `Algorithm` object - often via use of the `generate_report()` method.

In the remainder of this chapter, we cover each algorithm class provided by InQuanto in detail.

6.1 Variational Quantum Eigensolver `AlgorithmVQE`

The Variational Quantum Eigensolver (VQE) is the most well known and widely used Variational Quantum Algorithm (VQA). [6] It serves as one of the main hopes for quantum advantage in the Noisy Intermediate Scale Quantum (NISQ) era, due to its relatively low depth quantum circuits in contrast to other quantum algorithms.

$$\min E(\theta) = \langle \Psi(\vec{\theta}) | \hat{H} | \Psi(\vec{\theta}) \rangle \tag{6.1}$$

In InQuanto, the *inquanto.algorithms.vqe.AlgorithmVQE* class may be used to perform a VQE experiment. Like all `Algorithm` classes, this requires several precursor steps to generate input data. We will briefly cover these, however detailed explanations of relevant modules may be found later in this manual.

Firstly, we generate the system of interest. Here we choose the hydrogen molecule in a minimal basis. The H2 Hamiltonian (*FermionOperator* object) is obtained from the *inquanto.express* module, with the corresponding *FermionSpace* and a *FermionState* objects constructed explicitly thereafter.

```
from inquanto.express import load_h5
from inquanto.spaces import FermionSpace

h2 = load_h5("h2_sto3g.h5")
fermion_hamiltonian = h2["hamiltonian_operator"]
norb = h2["n_orbital"]
fermion_fock_space = FermionSpace(2 * norb)
fermion_state = fermion_fock_space.generate_occupation_state(
    n_fermion=h2["n_electron"],
    multiplicity=2 * h2["spin"] + 1,
)
```

We can then map the fermion operator onto a qubit operator with one of the available mapping methods.

```
from inquanto.mappings import QubitMappingJordanWigner
mapping = QubitMappingJordanWigner()
qubit_hamiltonian = mapping.operator_map(fermion_hamiltonian)
```

Next, we must define the parameterised wave function ansatz, which will be optimised during the VQE minimisation. InQuanto provides a suite of ansatzes, such as the Unitary Coupled Cluster (UCC) or the Hardware Efficient Ansatz (HEA). It is additionally possible to define a custom ansatz based on either fermionic excitations or explicit state generation circuits. Ansatz states in InQuanto are constructed using the `inquanto.ansatzes` module, which are detailed in the *Ansatzes* section.

```
from inquanto.ansatzes import FermionSpaceAnsatzUCCSD
ansatz = FermionSpaceAnsatzUCCSD(fermion_fock_space, fermion_state, mapping)
```

Now that we have the qubit Hamiltonian and the ansatz, we can define a `Computable` object we would like to pass to the minimizer during the VQE execution cycle. In this case we choose the expectation value of the molecular Hamiltonian - variationally minimizing this finds the ground state molecular energy. In InQuanto, *Computable* objects represent quantities that can be computed from a given quantum simulation. For variational algorithms, ansatz and qubit operator objects must be specified when constructing a `Computable` object.

```
from inquanto.computables import ExpectationValue
expectation_value = ExpectationValue(ansatz, qubit_hamiltonian)
```

We then define a *classical minimizer*. There is a variety of minimizers to choose from within InQuanto. In this case, we will use the *MinimizerScipy*.

```
from inquanto.minimizers import MinimizerScipy
minimizer = MinimizerScipy()
```

We can then define the initial parameters for our ansatz. Here, we use random coefficients, but this behaviour may be controlled as specified in the *Ansatzes* section.

```
initial_parameters = ansatz.state_symbols.construct_zeros()
```

We can then finally initialise the *AlgorithmVQE* object itself.

```
from inquanto.algorithms import AlgorithmVQE
vqe = AlgorithmVQE(
    objective_expression=expectation_value,
    minimizer=minimizer,
    initial_parameters=initial_parameters)
```

AlgorithmVQE also accepts an optional `auxiliary_expression` argument for any additional `Computable` (or their collection, *Computables*) object to be evaluated at the minimum ansatz parameters at the end of the VQE optimisation. It also accepts a `gradient_expression` argument for a special `Computable` type object, enabling calculation of analytical circuit gradients with respect to variational parameters (see below for an example).

A user must also define a protocol, defining how a `Computable` supplied to the objective expression will be computed at the circuit level (or using a state vector simulator). For more details see the *protocols* section. Here, we choose to use a state vector simulation.

```
from inquanto.protocols import ProtocolStateVectorSparse
protocol_expression = ProtocolStateVectorSparse()
```

One must also provide the `pytket Backend` of choice. These can be found in the pytket.extensions where a range of emulators and quantum hardware backends are provided.

```
from pytket.extensions.qiskit import AerStateBackend
backend = AerStateBackend()
```

Prior to running any algorithm, its procedures must be set up with the *build()* method. This method performs any classical preprocessing, and can be thought of as the step which defines how the circuit for the `Computable` is run.

```
vqe.build(
    backend=backend,
    protocol_expression=protocol_expression)
```

```
<inquanto.algorithms.vqe._algorithm_vqe.AlgorithmVQE at 0x7fa64c68a6d0>
```

We can then finally execute the algorithm using the *run()* method. This step performs the simulation on either the actual quantum device or a simulator backend specified above. During the VQE optimisation cycle an expectation value is calculated and minimised at each step.

```
vqe.run()
```

```
# TIMER BLOCK-0 BEGINS AT 2023-05-02 11:25:28.993948
```

```
# TIMER BLOCK-0 ENDS - DURATION (s):  0.8204258 [0:00:00.820426]
```

```
<inquanto.algorithms.vqe._algorithm_vqe.AlgorithmVQE at 0x7fa64c68a6d0>
```

The results are obtained by calling the *generate_report()* method, which returns a dictionary. This dictionary stores all important information generated throughout the algorithm, such as the final value of the `Computable` quantity and the optimized values of ansatz parameters (final parameters).

```
print(vqe.generate_report())
```

```
{'minimizer': {'final_value': -1.1368465754720545, 'final_parameters': array([ 0.    ␣
↪   ,  0.        , -0.10723349])}, 'final_value': -1.1368465754720545, 'initial_
↪parameters': [{'ordering': 0, 'symbol': 's0', 'value': 0.0}, {'ordering': 1, 'symbol
↪': 's1', 'value': 0.0}, {'ordering': 2, 'symbol': 'd0', 'value': 0.0}], 'final_
↪parameters': [{'ordering': 0, 'symbol': 's0', 'value': 0.0}, {'ordering': 1, 'symbol
↪': 's1', 'value': 0.0}, {'ordering': 2, 'symbol': 'd0', 'value': -0.
↪10723348543796078}]}
```

A modified initialisation of the *AlgorithmVQE* allows to use analytical circuit gradients as part of the calculation. Note here the additional `protocol_gradient` argument passed to the *build()* method.

```
from inquanto.protocols import ProtocolStateVectorSparse

gradient_expression = expectation_value.gradient_real()
vqe_with_gradient = (
    AlgorithmVQE(
        objective_expression=expectation_value,
        minimizer=minimizer,
        initial_parameters=initial_parameters,
        gradient_expression=gradient_expression,
    )
    .build(
        backend=backend,
        protocol_expression=protocol_expression,
        protocol_gradient=ProtocolStateVectorSparse()
    )
    .run()
)

results = vqe_with_gradient.generate_report()
```

(continues on next page)

(continued from previous page)

```
print(f"Minimum Energy: {results['final_value']}")
param_report = results["final_parameters"]
for i in range(len(param_report)):
    print(f"{param_report[i]['symbol']}: {param_report[i]['value']}")
```

```
# TIMER BLOCK-1 BEGINS AT 2023-05-02 11:25:29.857605
```

```
# TIMER BLOCK-1 ENDS - DURATION (s):  0.2135648 [0:00:00.213565]
Minimum Energy: -1.1368465754720536
s0: 0.0
s1: 0.0
d0: -0.10723347230091546
```

The use of analytic gradients may reduce the computational cost of the overall algorithm and can impact convergence.

6.2 Variational Quantum Deflation `AlgorithmVQD`

The Variational Quantum Deflation (VQD) algorithm is a variational minimisation algorithm that sequentially finds excited states by minimising an objective function (shown below) which penalises overlapping states over several VQE experiments [7]. Using the orthogonality of eigenvectors of a hermitian matrix, we constrain the state of interest to be orthogonal to the previously found states. We typically find the highest excited state first, then those between the lowest excited state to the second-highest.

$$E(\theta_k) = \langle\Psi(\theta_k)|\hat{H}|\Psi(\theta_k)\rangle + \sum_{i}^{k-1} \lambda_i |\langle\Psi(\theta_k)|\Psi(\theta_i)\rangle|^2 \tag{6.2}$$

In the above equation, $\{\theta_i\}$ are the parameters of the known states, and $\{\theta_k\}$ are the variational parameters of each excited state determined during the k-th VQD iteration. The λ_i parameter is the weight of the penalty corresponding to the overlap of the ith known state with the k-th excited state.

Before running a VQD algorithm, one must first run a VQE experiment to establish the electronic ground state such that the first excited state found during VQD can be constrained to be orthogonal to the ground state.

An example of how to run *AlgorithmVQD* is shown below. Following the same steps as before an initial VQE is run to obtain the ground state (and re-using the space, state, qubit mapping, and hamiltonian of the previous VQE example).

```
from inquanto.ansatzes import FermionSpaceAnsatzkUpCCGSD
from inquanto.states import FermionState

h2_sto3g = load_h5("h2_sto3g.h5", as_tuple=True)
space = FermionSpace(4)
state = FermionState([1, 1, 0, 0])

qubit_hamiltonian = h2_sto3g.hamiltonian_operator.qubit_encode()

ansatz = FermionSpaceAnsatzkUpCCGSD(space, state, k_input=2)

expectation_value = ExpectationValue(ansatz, qubit_hamiltonian)

minimizer = MinimizerScipy(method="L-BFGS-B")

vqe = (
    AlgorithmVQE(
```

(continues on next page)

(continued from previous page)

```
        expectation_value,
        minimizer,
        initial_parameters=ansatz.state_symbols.construct_zeros(),
    )
    .build(
        backend=AerStateBackend(),
        protocol_expression=ProtocolStateVectorSparse(),
        n_shots=0,
    )
    .run()
)
```

```
# TIMER BLOCK-2 BEGINS AT 2023-05-02 11:25:30.126728
```

```
# TIMER BLOCK-2 ENDS - DURATION (s):  1.7353607 [0:00:01.735361]
```

We then insist that any excited state optimised during the VQD algorithm is orthogonal to the VQE ground state. First we must create a deflationary ansatz (which defines the space in which we expand the excited states). In this example we simply use the same ansatz as we used in the VQE experiment, and modify the symbols such that the `protocols` can distinguish between the wave functions for the ground and excited states. Similarly to the VQE example, we also use the *`ExpectationValue`* class for the energy of our trial state.

```
ansatz_2  = ansatz.subs("{}_2") #Generate a copy of the ansatz with new symbols.
expectation_value = ExpectationValue(ansatz_2, qubit_hamiltonian)
```

Now we are left with calculating the weight and the overlap, shown in the second term of Eq. (6.2). We can define the weight arbitrarily. For this example we follow the recipe in the original paper [7], and simply take the expectation value of the Hamiltonian multiplied by -1. The overlap between the two ansatzes is defined as another `Computable` object, using the *`OverlapSquared`* class.

```
from inquanto.computables import OverlapSquared

weight_expression = ExpectationValue(ansatz_2, -1 * qubit_hamiltonian)
overlap_expression = OverlapSquared(ansatz, ansatz_2)
```

Finally we must instantiate the VQD object, build and run the algorithm. Unlike the *`AlgorithmVQE`* object, *`AlgorithmVQD`* takes in a number of different objective expressions; expectation value, overlap and weight, which define the energy function, penalty function and weight of the penalty, respectively. Similarly to *`AlgorithmVQE`*, we also provide some initial parameters. Importantly, for every expression there is a corresponding protocol supplied to the *`build()`* method.

```
from inquanto.algorithms import AlgorithmVQD

vqd = (
    AlgorithmVQD(
        expectation_value,
        overlap_expression,
        weight_expression,
        minimizer,
        ansatz_2.state_symbols.construct_random(seed=0),
        vqe._final_value,
        vqe._final_parameters,
        3,
    )
    .build(
```

(continues on next page)

(continued from previous page)

```
        backend=AerStateBackend(),
        objective_protocol=ProtocolStateVectorSparse(),
        overlap_protocol=ProtocolStateVectorSparse(),
        weight_protocol=ProtocolStateVectorSparse(),
        n_shots=0,
    )
    .run()
)

print("state_energies:", vqd.final_values)
print("state_parameters:", vqd.final_parameters)
```

```
# TIMER BLOCK-3 BEGINS AT 2023-05-02 11:25:31.951160
```

```
# TIMER BLOCK-3 ENDS - DURATION (s):  6.5476277 [0:00:06.547628]
# TIMER BLOCK-4 BEGINS AT 2023-05-02 11:25:38.498906
```

```
# TIMER BLOCK-4 ENDS - DURATION (s): 18.8504480 [0:00:18.850448]
# TIMER BLOCK-5 BEGINS AT 2023-05-02 11:25:57.349652
```

```
# TIMER BLOCK-5 ENDS - DURATION (s): 67.4893926 [0:01:07.489393]
state_energies: [-1.13684657547204, -0.49517377025688236, -0.13583641112011022, 0.
↪5515572309169572]
state_parameters: [SymbolDict({gs0k0: 0.0, gs1k0: 0.0, gd0k0: -0.05361669853564328,␣
↪gs0k1: 0.0, gs1k1: 6.516740071956524e-09, gd0k1: -0.053616706977538064}),␣
↪SymbolDict({gs0k0_2: 1.3552380762877467, gs1k0_2: -1.3552384468384036, gd0k0_2: -0.
↪8333046148929601, gs0k1_2: 0.6413321229514165, gs1k1_2: -1.2294319644914682, gd0k1_
↪2: -0.07212003092746834}), SymbolDict({gs0k0_2: 1.0441073195094441, gs1k0_2: -2.
↪097486525586573, gd0k0_2: 1.1114535447811609, gs0k1_2: 0.23891513802841635, gs1k1_2:
↪ -1.147902333433262, gd0k1_2: -0.07211871635398953}), SymbolDict({gs0k0_2: 0.
↪9999215252165565, gs1k0_2: -0.8377473481055973, gd0k0_2: -0.2984963289310002, gs0k1_
↪2: 0.43712605727922005, gs1k1_2: -0.6215231585969984, gd0k1_2: 0.005991590036164729}
↪)]
```

This algorithm is more expensive than VQE as it requires evaluation of more quantum circuits per step, due to the overlap and weight expressions. The procedure must also be repeated as many times as the number of the desired excited states!

6.3 Quantum Subspace Expansion `AlgorithmQSE`

The Quantum Subspace Expansion (QSE) obtains extra correlation and energetics of excited states, on top of the VQE ground state $|\Psi_{\text{VQE}}\rangle$ by doing a linear excitation expansion as [8]

$$|\Psi_{\text{QSE}}\rangle = \sum_{i>j} c_{ij}|\psi_{ij}\rangle \tag{6.3}$$

where

$$|\psi_{ij}\rangle = a_i^\dagger a_j|\Psi_{\text{VQE}}\rangle \tag{6.4}$$

and c_{ij} are the complex coefficients.

If $|\Psi_{\text{VQE}}\rangle$ is a correlated state, the basis functions of the subspace generated by applying the excitation operators are not orthogonal in general. The optimal solution for the ground and excited states within this subspace can be found by solving

the generalized eigenvalue problem expressed as

$$HC = SCE \tag{6.5}$$

where H is the Hamiltonian matrix and S is the overlap matrix in this subspace. E is the diagonal matrix of eigenvalues and C is the matrix of eigenvectors. The matrix elements of H and S are evaluated by quantum computers as

$$H_{ij,kl} = \langle\psi_{ij}|\hat{H}|\psi_{kl}\rangle \tag{6.6}$$

$$S_{ij,kl} = \langle\psi_{ij}|\psi_{kl}\rangle \tag{6.7}$$

And the eigenstates C and E are obtained with classical diagonalization.

AlgorithmQSE is a high level interface of inquanto to run the QSE algorithm for given qubit operator and ansatz with optimal parameters to evaluate the eigenstates. We take the minimal basis hydrogen molecule as an example to calculate the spin-adapted excited states from the VQE state with UCCSD ansatz.

The first step is constructing the molecular Hamiltonian and the Fock state in fermion objects.

```
from inquanto.express import load_h5
from inquanto.spaces import FermionSpace

expr = load_h5("h2_sto3g.h5")
hamiltonian = expr["hamiltonian_operator"]
norb = expr["n_orbital"]
fermion_space = FermionSpace(2 * norb)
fermion_state = fermion_space.generate_occupation_state(
    n_fermion=expr["n_electron"],
    multiplicity=2*expr["spin"]+1,
)
```

And then map them into the qubit objects. Here we use the UCCSD ansatz and then address the excited states with QSE, but one can use a cheaper ansatz for the VQE and obtain the extra correlation at the later QSE step, through the linear expansion and matrix diagonalisation.

```
from inquanto.mappings import QubitMappingJordanWigner
from inquanto.ansatzes import FermionSpaceAnsatzUCCSD

jw = QubitMappingJordanWigner()
qubit_hamiltonian = jw.operator_map(hamiltonian)
ansatz = FermionSpaceAnsatzUCCSD(fermion_space, fermion_state, qubit_mapping=jw)
```

Subsequently, we need to define the subspace in which to solve the generalised eigenvalue equations; namely the set of expansion operators to be applied to our ground state.

```
expansion_operators = jw.operator_map(
    fermion_space.generate_subspace_singlet_singles()
)
```

Note that we use singlet single excitation operators $E_{ij} = \sum_{\sigma}^{\text{spin}} a_{i\sigma}^{\dagger} a_{j\sigma}$ to span the spin-adapted subspace. We then define a new `Computable` object for the H and S matrices in Eq. (6.5), using the *ComputableQSEMatrices* class. This computable and the values of the parameters of the ansatz are then passed to *AlgorithmQSE*. In general we first need to perform a VQE calculation to obtain the optimized values of the parameters of the ansatz. In this specific example, we already know them from the previous VQE example, so we provide the values as an array.

```
from inquanto.computables import ComputableQSEMatrices
from inquanto.algorithms import AlgorithmQSE
```

(continues on next page)

(continued from previous page)

```
vqe_parameters = ansatz.state_symbols.construct_from_array(
    [0.0, 0.0, -0.107233493519281]
)

computable = ComputableQSEMatrices(
    state=ansatz,
    hermitian_operator=qubit_hamiltonian,
    expansion_operators=expansion_operators,
)

algorithm = AlgorithmQSE(
    computable_qse_matrices=computable,
    parameters=vqe_parameters,
)
```

The following procedure is basically the same as the examples above. No solver is required, as it is a nonvariational method.

```
from inquanto.protocols import ProtocolDirect
from pytket.extensions.qiskit import AerBackend

protocol_expression = ProtocolDirect()
backend = AerBackend()

algorithm.build(
    backend=backend,
    objective_protocol=protocol_expression,
    n_shots=2000,
)

algorithm.run()
print(algorithm.generate_report())
```

```
# TIMER Evaluate H and S matrix element BEGINS AT 2023-05-02 11:27:13.166813
# TIMER Evaluate H and S matrix element ENDS - DURATION (s):  0.0242956 [0:00:00.
↪024296]
# TIMER Solve HC=CSE BEGINS AT 2023-05-02 11:27:13.191165
# TIMER Solve HC=CSE ENDS - DURATION (s):  0.0026059 [0:00:00.002606]
{'final_value': array([-1.13473207, -0.13618023,  0.53953043]), 'final_states':␣
↪array([[-4.70593266e-01+1.71146618e-01j, -1.53931464e-03+2.60788005e-04j,
         4.58307519e-02-1.74730096e-02j],
       [-3.85604997e-02-1.42942043e-02j,  8.99795165e-02+2.82848880e-02j,
        -3.98741812e-01-1.37005117e-01j],
       [ 4.13218322e-03+1.09535205e-03j, -6.74494181e-01-1.93820392e-01j,
         8.40096355e-03-1.36276548e-03j],
       [-4.40292975e-01+1.04715055e-01j,  9.69848219e-02-2.34036237e-02j,
        -4.31167244e+00+1.02530300e+00j]])}
```

The eigenvalues and eigenvectors are stored as `final_value` and the `final_states`, respectively. Numerical instabilities in the matrix diagonalization are handled by removing near linear dependencies in the basis.

6.4 AlgorithmAdaptVQE and AlgorithmIQEB

AlgorithmAdaptVQE and *AlgorithmIQEB* are algorithms that construct an ansatz iteratively from a collection or pool of excitation operators [9] [10]. This differs from algorithms like *AlgorithmVQE* in which the ansatz is fixed, i.e. parameters of a fixed set of operators are optimized. At variance, in *AlgorithmAdaptVQE* and *AlgorithmIQEB*, excitation operators are selected and their exponentials appended to the ansatz iteratively until convergence criteria are met. Practically, this usually results in more circuits measurements required for these algorithms compared to VQE, however the resulting ansatz will (usually) have fewer terms compared to a straight-forward application of a fixed ansatz e.g. UCCSD optimized in VQE for the same accuracy. Hence, compared to typical VQE, these algorithms trade-off number of measurements with circuit depth.

After N iterations of the algorithm, the resulting (ADAPT/IQEB) ansatz will have the form

$$\hat{U}^{(N)}_{\text{ADAPT/IQEB}} = \left(e^{\theta_N \hat{A}_N}\right)\left(e^{\theta_{N-1}\hat{A}_{N-1}}\right)\ldots\left(e^{\theta_\lambda \hat{A}_\lambda}\right)\ldots\left(e^{\theta_1 \hat{A}_1}\right) \tag{6.8}$$

$$|\Psi^{(N)}_{\text{ADAPT/IQEB}}\rangle = \hat{U}_{\text{ADAPT/IQEB}}|\text{HF}\rangle \tag{6.9}$$

Where the excitation operators $\hat{A}$ are chosen by the user, and the corresponding parameters θ are optimized, by *AlgorithmAdaptVQE* or *AlgorithmIQEB*. In each iteration, the current ansatz is applied to a reference state; a Hartree-Fock state is a common choice. The two major differences between *AlgorithmAdaptVQE* and *AlgorithmIQEB* are i) the space in which the operators act on, and ii) the method to select the operators to append to the ansatz.

In the ADAPT (Adaptive Derivative-Assembled Pseudo-Trotter ansatz)-VQE algorithm [9], the operators $\hat{A}_\lambda$ in the pool consist of all possible spin complemented anti-hermitian operators within *UCC*, which act in fermionic space. While these are fermionic operators, they must be transformed to qubit operators via a fermion-to-qubit mapping to be accepted by *AlgorithmAdaptVQE*. The operators to be appended to the ansatz at the n^{th} iteration of the algorithm are chosen by calculating the following gradient of the total energy E

$$\frac{\partial E^{(n)}}{\partial \theta_\lambda} = \langle \Psi^{(n)}_{\text{ADAPT}}|[\hat{H}, \hat{A}_\lambda]|\Psi^{(n)}_{\text{ADAPT}}\rangle \tag{6.10}$$

for each excitation operator. The $\hat{A}_\lambda$ which yields the largest gradient is appended to the ansatz, and a regular VQE calculation is performed to determine the optimal ansatz parameters at this iteration. In the next iteration, the gradients are recalculated as before but with the $e^{\theta_\lambda \hat{A}_\lambda}$ from the previous iteration appended. This is repeated until all gradients are below a tolerance threshold.

The following shows an example of how to run *AlgorithmAdaptVQE* in which the operators of the `pool` are restricted to *UCCSD* (re-using the fermion space, qubit-encoded H_2 Hamiltonian, and qubit mapping of the *AlgorithmAdaptVQE* example).

```
from inquanto.algorithms import AlgorithmAdaptVQE
from inquanto.states import QubitState,FermionState
from inquanto.protocols import (
    ProtocolStateVectorSparse
)
from inquanto.spaces import FermionSpace

space = FermionSpace(4)
state = QubitState([1, 1, 0, 0])
jw_map = QubitMappingJordanWigner()

pool = space.construct_single_ucc_operators(state)
pool += space.construct_double_ucc_operators(state)
pool = jw_map.operator_map(pool)
```

(continues on next page)

(continued from previous page)

```
scipy_minimizer = MinimizerScipy(method="L-BFGS-B", disp=False)

adapt = AlgorithmAdaptVQE(
    pool,
    state,
    qubit_hamiltonian,
    scipy_minimizer,
    tolerance=1.0e-3
)

adapt.build(
    AerStateBackend(),
    ProtocolStateVectorSparse(),
    ProtocolStateVectorSparse(),
    ProtocolStateVectorSparse()
)

adapt.run()

results = adapt.generate_report()

print("Minimum Energy: {}".format(results["final_value"]))
param_report = results["final_parameters"]
for i in range(len(param_report)):
    print(param_report[i]["symbol"], ":", param_report[i]["value"])
```

```
# TIMER BLOCK-6 BEGINS AT 2023-05-02 11:27:13.251820
```

```
# TIMER BLOCK-6 ENDS - DURATION (s):  0.2109676 [0:00:00.210968]
```

```
Minimum Energy: -1.1368465754720536
d0 : -0.10723347230091546
```

Here, the pool of operators was generated by the *construct_single_ucc_operators()* and *construct_double_ucc_operators()* methods of the *FermionSpace* class. Like the Hamiltonian, these operators were qubit-encoded before passing into *AlgorithmAdaptVQE*. The `tolerance` parameter sets the gradient threshold for convergence of the algorithm. The protocol *ProtocolStateVectorSparse* is needed to calculate the gradients defined *above*.

Alternatively, one can use fermionic states and operators as arguments to the algorithm class *AlgorithmFermionicAdaptVQE*, which inherits from *AlgorithmAdaptVQE*, and performs the qubit mapping internally. Below is an example of *AlgorithmFermionicAdaptVQE* where again the fermionic space has been re-used (also the `scipy_minimizer`), and this time we use the fermionic H_2 Hamiltonian (defined in the *AlgorithmFermionicAdaptVQE* example).

```
from inquanto.algorithms import AlgorithmFermionicAdaptVQE

state = FermionState([1, 1, 0, 0])

pool = space.construct_single_ucc_operators(state)
pool += space.construct_double_ucc_operators(state)

fermionic_adapt = AlgorithmFermionicAdaptVQE(
    pool,
```

(continues on next page)

(continued from previous page)

```
    state,
    fermion_hamiltonian,
    scipy_minimizer,
    tolerance=1.0e-3
)

fermionic_adapt.build(
    AerStateBackend(),
    ProtocolStateVectorSparse(),
    ProtocolStateVectorSparse(),
    ProtocolStateVectorSparse()
)

fermionic_adapt.run()

results = fermionic_adapt.generate_report()

print("Minimum Energy: {}".format(results["final_value"]))
param_report = results["final_parameters"]
for i in range(len(param_report)):
    print(param_report[i]["symbol"], ":", param_report[i]["value"])
```

```
# TIMER BLOCK-7 BEGINS AT 2023-05-02 11:27:13.542434
```

```
# TIMER BLOCK-7 ENDS - DURATION (s):  0.1943107 [0:00:00.194311]
Minimum Energy: -1.1368465754720536
d0 : -0.10723347230091546
```

Note that in this case the reference `state` and `pool` have not been qubit encoded before passing into *AlgorithmFermionicAdaptVQE*. Jordan-Wigner encoding is performed by default.

In the IQEB (Iterative Qubit-Excitation Based)-VQE algorithm [10], the operators in the pool correspond to qubit excitations. Qubit excitation operators are generated from the following ladder operators which obey the so-called parafermionic [11] commutation relations

$$\{\hat{Q}_i, \hat{Q}_i^\dagger\} = I, \tag{6.11}$$

$$[\hat{Q}_i, \hat{Q}_j^\dagger] = 0 \quad (i \neq j), \tag{6.12}$$

$$[\hat{Q}_i, \hat{Q}_j] = [\hat{Q}_i^\dagger, \hat{Q}_j^\dagger] = 0 \quad \forall i, j \tag{6.13}$$

where $\hat{Q}_i$ ($\hat{Q}_i^\dagger$) is a qubit annihilation (creation) operator which changes the occupation of spin orbital i (assuming a Jordan-Wigner encoding of the Hamiltonian and reference state), and which can be represented in terms of Pauli gates

$$\hat{Q}_i = \frac{1}{2}(X_i + iY_i) \tag{6.14}$$

$$\hat{Q}_i^\dagger = \frac{1}{2}(X_i - iY_i) \tag{6.15}$$

The pool in *AlgorithmIQEB* consists of one- and two-body qubit excitation operators, built from these parafermionic operators, and acting on qubit (or spin orbital) indexes i, j, k, l (where the set of indexes is unique to the λ-th operator in the `pool`).

$$\hat{A}_\lambda^{(ik)} = \hat{Q}_{i_\lambda}^\dagger \hat{Q}_{k_\lambda} - \hat{Q}_{k_\lambda}^\dagger \hat{Q}_{i_\lambda} = \frac{1}{2}(X_{i_\lambda} Y_{k_\lambda} - Y_{i_\lambda} X_{k_\lambda}) \tag{6.16}$$

$$\hat{A}_\lambda^{(ijkl)} = \hat{Q}_{i_\lambda}^\dagger \hat{Q}_{j_\lambda}^\dagger \hat{Q}_{k_\lambda} \hat{Q}_{l_\lambda} - \hat{Q}_{k_\lambda}^\dagger \hat{Q}_{l_\lambda}^\dagger \hat{Q}_{i_\lambda} \hat{Q}_{j_\lambda} \tag{6.17}$$

$$= \frac{1}{8}\left(X_{i_\lambda}Y_{j_\lambda}X_{k_\lambda}X_{l_\lambda} + Y_{i_\lambda}X_{j_\lambda}X_{k_\lambda}X_{l_\lambda} + Y_{i_\lambda}Y_{j_\lambda}Y_{k_\lambda}X_{l_\lambda} + Y_{i_\lambda}Y_{j_\lambda}X_{k_\lambda}Y_{l_\lambda} \right.$$
$$\left. - X_{i_\lambda}X_{j_\lambda}Y_{k_\lambda}X_{l_\lambda} - X_{i_\lambda}X_{j_\lambda}X_{k_\lambda}Y_{l_\lambda} - Y_{i_\lambda}X_{j_\lambda}Y_{k_\lambda}Y_{l_\lambda} - X_{i_\lambda}Y_{j_\lambda}Y_{k_\lambda}Y_{l_\lambda}\right) \tag{6.18}$$

Note that the *AlgorithmIQEB* class inherits from *AlgorithmAdaptVQE*. While gradients are also used in the selection process of *AlgorithmIQEB*, their purpose here is to narrow down the candidate operators from the total IQEB `pool`. Hence the convergence of *AlgorithmIQEB* is not evaluated directly by gradients as in *AlgorithmAdaptVQE*. Instead, *AlgorithmIQEB* checks the total energy difference between iterations, and convergence is achieved when the decrease of energy between iterations is less than a threshold (the `energy_tolerance` parameter in the code block below).

The following example shows how to run *AlgorithmIQEB* (using the previously defined H_2 qubit Hamiltonian and `scipy_minimizer`).

```
from inquanto.algorithms import AlgorithmIQEB
from inquanto.spaces import ParaFermionSpace
from inquanto.protocols import (
    ProtocolStateVectorSparse,
    ProtocolStateVectorSparse,
    ProtocolStateVectorSparse,
)

space = ParaFermionSpace(4)
state = QubitState([1, 1, 0, 0])

pool = space.construct_single_qubit_excitation_operators()
pool += space.construct_double_qubit_excitation_operators()

iqeb = AlgorithmIQEB(
    pool,
    state,
    qubit_hamiltonian,
    scipy_minimizer,
    n_grads=3,
    energy_tolerance=1.0e-10
)

iqeb.build(
    AerStateBackend(),
    ProtocolStateVectorSparse(),
    ProtocolStateVectorSparse(),
    ProtocolStateVectorSparse(),
)

iqeb.run()

results = iqeb.generate_report()

print("Minimum Energy: {}".format(results["final_value"]))
param_report = results["final_parameters"]
for i in range(len(param_report)):
    print(param_report[i]["symbol"], ":", param_report[i]["value"])
```

```
System has zero net spin -> will append spin-complementary exponents.
# TIMER BLOCK-8 BEGINS AT 2023-05-02 11:27:13.837360
```

```
# TIMER BLOCK-8 ENDS - DURATION (s):  0.3199169 [0:00:00.319917]
# TIMER BLOCK-9 BEGINS AT 2023-05-02 11:27:14.157943
```

(continues on next page)

(continued from previous page)

```
# TIMER BLOCK-9 ENDS - DURATION (s):  0.0155191 [0:00:00.015519]
# TIMER BLOCK-10 BEGINS AT 2023-05-02 11:27:14.174013
# TIMER BLOCK-10 ENDS - DURATION (s):  0.0202541 [0:00:00.020254]
# TIMER BLOCK-11 BEGINS AT 2023-05-02 11:27:14.230376
# TIMER BLOCK-11 ENDS - DURATION (s):  0.0601858 [0:00:00.060186]
# TIMER BLOCK-12 BEGINS AT 2023-05-02 11:27:14.292411
# TIMER BLOCK-12 ENDS - DURATION (s):  0.0485228 [0:00:00.048523]
# TIMER BLOCK-13 BEGINS AT 2023-05-02 11:27:14.343550
```

```
# TIMER BLOCK-13 ENDS - DURATION (s):  0.0671466 [0:00:00.067147]

CONVERGED!!!
Final ansatz elements after 2 iteration(s):
r_0_6     [(0.125j, X0 Y1 X2 X3), (0.125j, Y0 X1 X2 X3), (0.125j, Y0 Y1 Y2 X3), (0.
↪125j, Y0 Y1 X2 Y3), (-0.125j, X0 X1 Y2 X3), (-0.125j, X0 X1 X2 Y3), (-0.125j, X0 Y1␣
↪Y2 Y3), (-0.125j, Y0 X1 Y2 Y3)]

Minimum Energy: -1.1368465754720536
r_0_6 : -0.10723347230091546
```

Notice that six separate VQE calculations have been performed (one for each `TIMER BLOCK` in the output log). This is due to two reasons. i) Our choice of `n_grads=3`, which tells *AlgorithmIQEB* that we want to narrow down the `pool` to those terms which have the three largest gradients, and a VQE calculation for each term will be run. Of these three, the term which has the largest effect on the energy will be appended to the ansatz in this iteration. ii) Since *AlgorithmIQEB* establishes convergence by comparing the energy difference between iterations, a second iteration is performed, again with three separate terms (appended to the previously found term). In this case, convergence is found at the second iteration, which means the resulting ansatz will have the form of the first iteration.

As in the case of *AlgorithmAdaptVQE*, we define a pool of operators. However here we employ the *ParaFermionSpace* class to handle the parafermionic operator algebra. The operators in the IQEB `pool` consist of all unique permutations of qubit indexes for one- and two-body terms, obtained by the *construct_single_qubit_excitation_operators()* and *construct_double_qubit_excitation_operators()* methods of *ParaFermionSpace()* (which do not need a reference state). This results in an asymptotically larger `pool` than *AlgorithmAdaptVQE* [10]. However, the advantage of *AlgorithmIQEB* is that excitation operators act directly in parafermionic space, hence the strings of Pauli-Z operators resulting from Jordan-Wigner encoding, in order to maintain fermionic exchange symmetry, are not required. Therefore each qubit excitation of *AlgorithmIQEB* acts on a fixed number of qubits, independent of the system size.

6.5 Time evolution using `AlgorithmVQS`, `AlgorithmMcLachlanRealTime` and `AlgorithmMcLachlanImagTime`

These three classes implement variational quantum simulation (VQS) - a family of methods, solving the time-dependent Schrödinger equation (TDSE) by propagating a parameterized wavefunction (ansatz) using equations of motion (EOM), derived from one of the existing time-dependent variational principles (see [12] for a comprehensive review).

AlgorithmVQS is a general implementation, accepting any EOM in the form of a `Computable` expression parameter and any integrator. *AlgorithmVQS* assumes that EOM has a form of a linear ordinary differential equation (ODE), i.e. has the general form $Ax = b$.

AlgorithmMcLachlanRealTime and *AlgorithmMcLachlanImagTime* are specialised classes. They implement the real and imaginary time evolution for a pure state respectively, following the EOM derived from the McLachlan

variational principle, as provided in Eq.12 of [12]. Following this reference, A is the real part of the ansatz metric tensor:

$$A = \Re(\frac{\partial < \phi(\theta(t))|}{\partial \theta_i} \frac{\partial |\phi(\theta(t)) >}{\partial \theta_j}) \tag{6.19}$$

and b is either:

$$b = \Re(\frac{\partial < \phi(\theta(t))|}{\partial \theta_i} H |\phi(\theta(t)) >) \tag{6.20}$$

for imaginary-time evolution, or:

$$b = \Im(\frac{\partial < \phi(\theta(t))|}{\partial \theta_i} H |\phi(\theta(t)) >) \tag{6.21}$$

for real-time evolution.

Below we provide an example of how these time evolution methods are applied, utilizing an ansatz to represent the wavepacket. The ansatz mixes the $|1,1,0,0>$ and $|0,0,1,1>$ states of a hydrogen molecule in the minimal basis set, allowing for an oscillation of the relative populations of those states during the time evolution.

First, we construct the H_2 Hamiltonian, as well as several observables that we would like to track during the wavepacket evolution (total energy, total particle number and orbital occupation number):

```
from inquanto.express import load_h5
from inquanto.spaces import FermionSpace

system = load_h5("h2_sto3g.h5", as_tuple=True)
hamiltonian_operator = system.hamiltonian_operator.qubit_encode()
fock_space = FermionSpace(4)
particle_number_operator = fock_space.construct_number_operator().qubit_encode()
orbital_number_operators = [
    op.qubit_encode() for op in fock_space.construct_orbital_number_operators()
]
```

Then, we define an ansatz that mixes the two configurations of interest. The configurations are one with both electrons in the first molecular spatial orbital ($|1,1,0,0>$, the Hartree-Fock ground state), and the other where both electrons are excited to the second spatial molecular orbital ($|0,0,1,1>$). Note that we add a global phase to the ansatz circuit and specify an initial condition for the wavepacket by setting the first ansatz parameter `theta1` to $\frac{\pi}{4}$:

```
from inquanto.ansatzes import TrotterAnsatz, CircuitAnsatz
from inquanto.operators import QubitOperatorList
from inquanto.core import OpType
from sympy import Symbol, pi

c1 = TrotterAnsatz(
    QubitOperatorList.from_string("theta1 [(1j, Y0 X1 X2 X3)]"), [1, 1, 0, 0]
).get_circuit()
c1.add_gate(OpType.from_name("U3"), [0, 0, Symbol("theta2") / pi], [0])  # Adds␣
↪global phase
ansatz = CircuitAnsatz(c1)
initial = ansatz.state_symbols.construct_from_array([pi / 4, 0.0])
```

We are now in a position to initialize the integrator object. Here we use *NaiveEulerIntegrator*, but a more accurate SciPy integrator can be chosen as well - note the fine step size we need to obtain converged dynamics. Having created the integrator object, we construct the VQS algorithm object and then run the wavepacket propagation:

```
import numpy
from pytket.extensions.qiskit import AerStateBackend
```

(continues on next page)

(continued from previous page)

```
from inquanto.minimizers import NaiveEulerIntegrator
from inquanto.algorithms import AlgorithmMcLachlanRealTime
from inquanto.protocols import ProtocolStateVectorSparse

time = numpy.linspace(0, 5, 501)
integrator = NaiveEulerIntegrator(
    time, disp=True, linear_solver=NaiveEulerIntegrator.linear_solver_scipy_pinvh
)
algodeint = AlgorithmMcLachlanRealTime(
    integrator,
    hamiltonian_operator,
    ansatz,
    initial_parameters=initial,
)
solution = algodeint.build(
    backend=AerStateBackend(),
    protocol=ProtocolStateVectorSparse(),
).run()
```

Finally, after the propagation has been completed, we can post-evaluate expectation values of the desired observables for each time step:

```
from inquanto.computables import ExpectationValue

evs_expression = ExpectationValue.ndarray_broadcast(
    ansatz, [hamiltonian_operator, particle_number_operator, *orbital_number_
↪operators]
).build(protocol)
evs = algodeint.post_propagation_evaluation(evs_expression)
evs = numpy.asarray(evs)
```

Here we plot the columns of the `evs` array to analyse results of the dynamics. We see that the electron number remains constant, while orbital occupations oscillate - this is due to the fact that we started by mixing the two electronic configurations, making the system non-stationary, and some electron (charge) dynamics is expected.

However, total energy is not conserved, i.e. the propagation was not very accurate. The reason for this is that the EOM directly derived from McLachlan variational principle doesn't properly account for the wavepacket phase evolution. This can be fixed by adding extra terms to it, as described in [12]. One can easily implement this EOM in InQuanto by creating a custom class, derived from the `ComputableSuper` class and overriding the constructor and `evaluate()` method (see the time evolution *examples*). Then one can pass an instance of such class, together with the integrator of choice, to the *`AlgorithmVQS`* constructor, build and run it to perform a more accurate propagation.

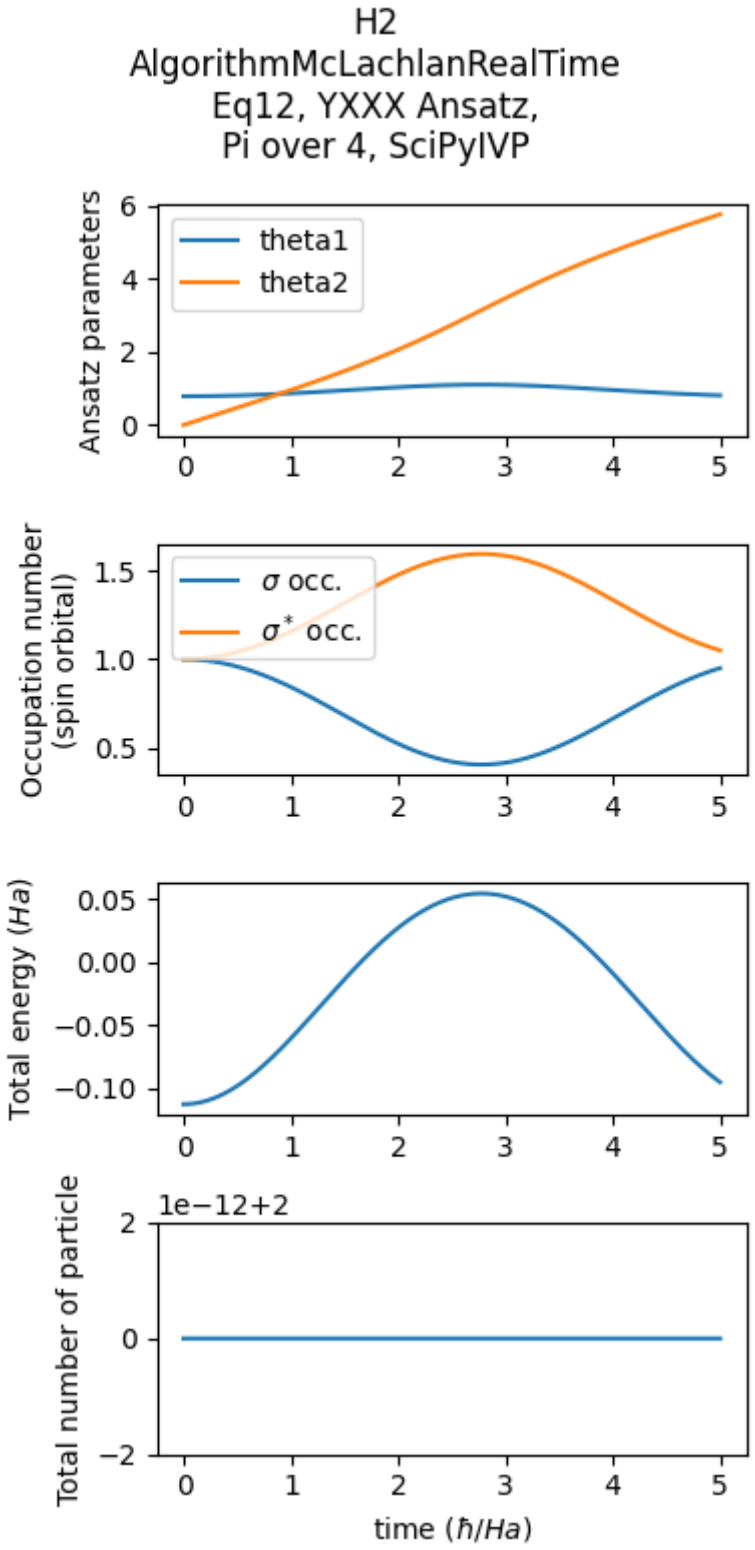


Fig. 6.2: Time evolution of various observables during a H_2 electronic wavepacket propagation.

CHAPTER

SEVEN

COMPUTABLES

Computables encapsulate the essential operations required to evaluate physical quantities on a quantum computer, either as directly measurable observables or as derived quantities that result from decomposing into many observables. They are simpler than general quantum *Algorithms* in the sense that measurement circuits may be pregenerated before an experiment is run. In contrast, an algorithm may involve alteration of quantum circuits based on the measurement outcomes obtained. InQuanto offers a range of prototype Computables that correspond to common quantities. It is possible to write expressions using Computables to represent derived quantities, for which a native prototype does not exist.

Computables do not represent an immediately available quantity, but are expressions that require evaluation through state vector or shot-based simulation. The evaluation of Computables is facilitated by *Protocols*, which define the strategies and procedures necessary to perform the computations.

7.1 Basic Usage and Composability

ExpectationValue is the fundamental computable in InQuanto. We instantiate an *ExpectationValue* computable with a *QubitOperator* and an ansatz object.

```
from inquanto.express import load_h5
from sympy import sympify
from inquanto.operators import QubitOperator, QubitOperatorList
from inquanto.states import QubitState
from inquanto.ansatzes import TrotterAnsatz

h2 = load_h5("h2_sto3g.h5", as_tuple=True)
hamiltonian = h2.hamiltonian_operator.qubit_encode()
theta = sympify("theta")
exponents = QubitOperatorList(QubitOperator("Y0 X1 X2 X3", 1j), theta)
reference = QubitState([1, 1, 0, 0])
ansatz = TrotterAnsatz(exponents, reference)

from inquanto.computables import ExpectationValue
energy = ExpectationValue(ansatz, hamiltonian)
```

Here, `energy` represents the expression typically written as $\langle\psi|\hat{H}|\psi\rangle$. InQuanto allows for basic linear algebra to be performed with Computable objects:

```
e1 = ExpectationValue(ansatz, op1)
e2 = ExpectationValue(ansatz, op2)
energy = e1 + 0.5 * e2
```

Multiple Computable objects can be combined into a list (*ComputableList*) or a tuple (*Computables*):

```
e1 = ExpectationValue(ansatz, op1)
e2 = ExpectationValue(ansatz, op2)
m1 = Computables(e1, e2)
m2 = ComputableList(e1, e2)
```

Due to the limited set of observable quantities that can be directly measured on quantum devices, many derived quantities must be determined as an expression composed of simpler observables. InQuanto's approach of treating Computables as expressions enables it to describe and measure these derived quantities.

7.2 Evaluating Computables

A quantity represented by a computable object is not available immediately after it is initialized. For our `ExpectationValue` example, we must provide a strategy by which the computable is evaluated and a set of parameters to construct a valid eigenstate from the ansatz.

```
energy.build([ProtocolDirect()])
```

First we build the Computable, passing a *Protocols* object. A Protocol describes how the expression is evaluated. Here, *`ProtocolDirect`* will build a set of circuits to measure the expectation of all Pauli terms contained in the qubit Hamilonian.

```
energy.run(backend, {theta:-0.111}, n_shots=10000)
energy.evaluate()
```

Using the `run` method, we provide a hardware or simulator backend to run circuits, a set of parameters to generate an approximate eigenstate preparation circuit from the ansatz, and any additional parameters required for simulation. Once a Computable has been run, we can call the `evaluate` method. A Computable object will store the measured coefficients of each Pauli term in the operator; it is therefore possible to evaluate another operator on the same eigenstate without repeating measurement if the Pauli terms in the new operator are shared.

A Computable instance needs to be built once only. When constructing composite computables, building once on the top level Computable is preferable to achieve the greatest reduction in the number of required measurements. However, this also allows us to have composite Computables where each component is evaluated using a different Protocol.

```
e1 = ExpectationValue(ansatz, op1)
e2 = ExpectationValue(ansatz, op2)
e3 = ExpectationValue(ansatz, op3)
m = Computables(e1, e2, e3)

e1.build([ProtocolIndirect()])
m.build([ProtocolDirect()])
```

Here, `e1` is built with *`ProtocolIndirect`*, while both `e2` and `e3` are built with `ProtocolDirect`.

7.3 Derived Quantities

High level derived quantities are described in terms of simpler calculations, such as expectation values or state overlaps. InQuanto provides several of these high level computables which act as wrappers around operations which can be performed with other InQuanto modules.

For instance, *ComputableSpinlessNBodyRDMTensorReal* calculates a general n-body RDM $\Gamma^{i\cdots j}_{n\ldots m} = \langle\Psi_0|\,\hat{E}^{i\cdots j}_{n\ldots m}\,|\Psi_0\rangle$, where $\hat{E}^{i\cdots j}_{n\ldots m}$ is a spin-traced excitation operator.

We can evaluate each term in the n-body RDM by evaluating many `ExpectationValue` computables on each spin-traced excitation operator.

```
qubit_hamiltonian = load_h5(
    "h2_631g_symmetry.h5", as_tuple=True
    ).hamiltonian_operator.qubit_encode()

space = FermionSpace(
    8,
    point_group="D2h",
    orb_irreps=numpy.asarray(["Ag", "Ag", "B1u", "B1u", "Ag", "Ag", "B1u", "B1u"]),
    )


ansatz = FermionSpaceAnsatzChemicallyAwareUCCSD(
    fermion_space=space, fermion_state=FermionState([1, 1, 0, 0, 0, 0, 0, 0])
    )

qubit_space = QubitSpace(space.n_spin_orb)

symmetry_operators = qubit_space.symmetry_operators_z2(qubit_hamiltonian)

rdm1_spinless = space.construct_n_body_spinless_rdm_operators(1)

rdm1_spinless_qubit_list = []
for fop in rdm1_spinless:
    rdm1_spinless_qubit = QubitMappingJordanWigner().operator_map(op)
    if not all(x.is_of_symmetry(rdm1_spinless_qubit) for x in symmetry_operators):
        rdm1_spinless_qubit_list.append(QubitOperator.zero())
    else:
        rdm1_spinless_qubit_list.append(rdm1_spinless_qubit)

rdm1_spinless_qubit_list = [op.hermitian_part() for op in rdm1_spinless_qubit_list]

expression = ExpectationValue.ndarray_broadcast(
    ansatz,
    numpy.asarray(rdm1_spinless_qubit_list).reshape(rdm1_spinless.shape)
    )

prototype = ComputableSpinlessNBodyRDMTensorReal(
    1,
    space,
    ansatz,
    QubitMappingJordanWigner(),
    symmetry_operators,
    )
```

Above we use the express module to load a Hamiltonian. Using a fermion space and ansatz objects, we generate the Z2 symmetries of our Hamiltonian. We then use the space again to generate a set of RDM operators, map them to qubit

space using JW mapping, and filter for Z2 symmetry. The *ndarray_broadcast* method generates a composite computable, where the expectation value of each RDM operator is evaluated with respect to the ansatz. As such, we manually reproduced the required steps to create an exact analogue of the high level `ComputableSpinlessNBodyRDMTensorReal` class for 1-RDM.

CHAPTER

EIGHT

PROTOCOLS

Protocols represent strategies by which *Computables* expressions are evaluated. Protocols evaluate Computables by means of state-vector or shot based simulation, describe how state preparation circuits are composed, and construct measurement circuits.

8.1 Shot-based simulation

Shot-based protocols employ the Pauli averaging technique to evaluate the expectation value of observable quantities on circuits. Where we have a Hamiltonian written as a sum of tensor products of Pauli operators by means of *Jordan-Wigner or Brayvi-Kitaev mapping*, the energy may be written as the average

$$\langle \hat{H} \rangle = \sum_{i,j,k\cdots \in x,y,z} h_{ijk\ldots} \langle \sigma_1^i \otimes \sigma_2^j \otimes \sigma_3^k \ldots \rangle. \tag{8.1}$$

Each Pauli term must be measured on an eigenstate, where the above average will correspond to an energy eigenvalue of the Hamiltonian with some sampling related error.

We prepare eigenstates by providing coefficients to parameterized terms in state preparation circuits generated by *Ansatzes* objects.

Pauli terms can be measured on this prepared eigenstate by appending some measurement circuit. This may either be by directly operating on and measuring the state register, or via projective measurement. Where direct measurement is used, we can reduce the number of measurement circuits to be generated by partitioning our Hamiltonian into commuting sets of Pauli terms which can be measured simultaneously.

Consider the minimal example below:

```
from inquanto.express import load_h5
from sympy import sympify
from inquanto.operators import QubitOperator,QubitOperatorList
from inquanto.states import QubitState
from inquanto.ansatzes import TrotterAnsatz

h2 = load_h5("h2_sto3g.h5", as_tuple=True)
hamiltonian = h2.hamiltonian_operator.qubit_encode()
theta = sympify("theta")
exponents = QubitOperatorList(QubitOperator("Y0 X1 X2 X3", 1j), theta)
reference = QubitState([1, 1, 0, 0])
ansatz = TrotterAnsatz(exponents, reference)

from inquanto.computables import ExpectationValue
energy = ExpectationValue(ansatz, hamiltonian)
```

(continues on next page)

(continued from previous page)

```
energy.build([ProtocolDirect()])
energy.run(backend, {theta:-0.111}, n_shots=10000)
energy.evaluate()
```

To evaluate the *ExpectationValue* computable, *ProtocolDirect* generated two circuits.

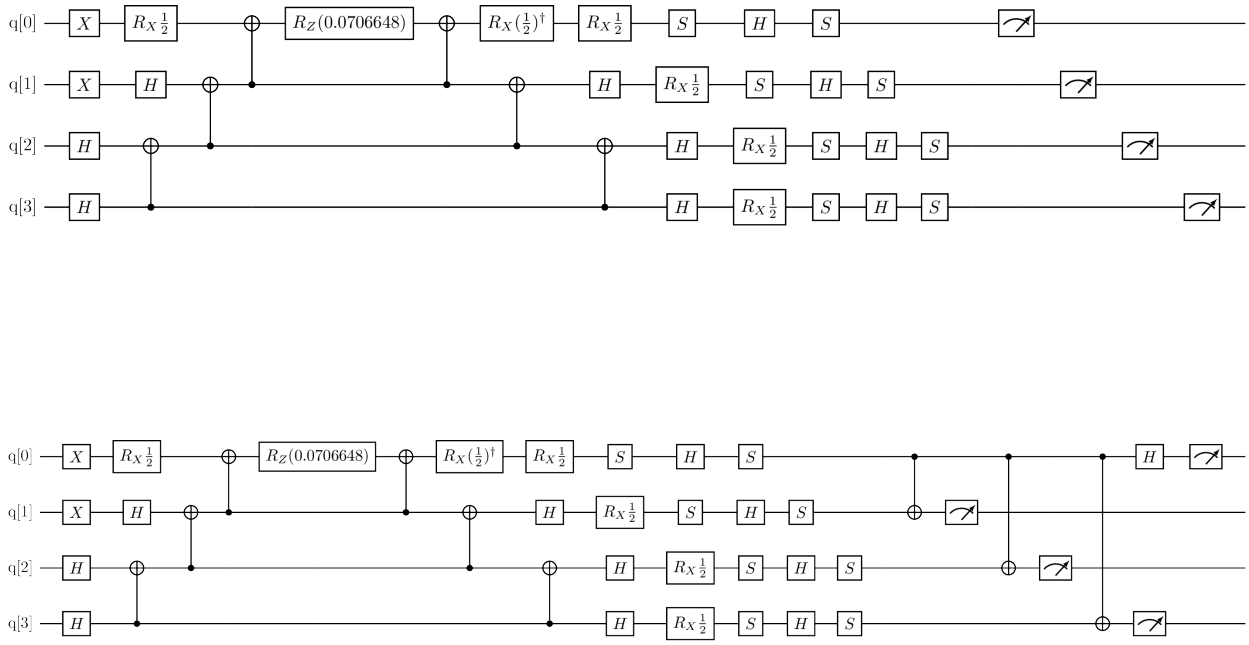


Fig. 8.1: Measurement circuits generated by ProtocolDirect

Through substituting coefficients in the parametrized terms of the ansatz circuit (here Rz gate) the protocol generates a guess for an eigenstate. Appended to the state preparation circuit, are the measurement circuits required to measure all Pauli terms of the Hamiltonian, where our mapped Hamiltonian takes the form (with coefficients ommited, and each term being an operator):

$$\begin{aligned}\hat{H} = Z_0 + Z_2 + Z_1 + Z_3 \\ + Z_0Z_2 + Z_1Z_3 + Z_0Z_1 + Z_1Z_2 \\ + Y_0X_1X_2Y_3 + X_0X_1Y_2Y_3 + Y_0Y_1X_2X_3 \\ + X_0Y_1Y_2X_3 + Z_0Z_3 + Z_2Z3\end{aligned} \tag{8.2}$$

The first circuit measures the individual Z terms. The second circuit measures the remainder.

As a convention, where a protocol is called “Direct”, any measurement happens directly on the system register. Where it is called “Indirect”, it utilizes ancilla qubits and performs measurements projectively.

8.2 Protocols for expectation values

8.2.1 ProtocolDirect

Given a Hermitian operator expressed as a weighted sum of Pauli terms (`QubitOperator`), `ProtocolDirect` constructs a set of measurement circuits from commuting sets of Pauli terms. The measurement circuits are appended directly to the system register. Multiple circuits may be measured, depending on the number of Pauli terms in the operator. The final expectation value of the operator is then the sum over the product of Pauli term weights with the measured expectations.

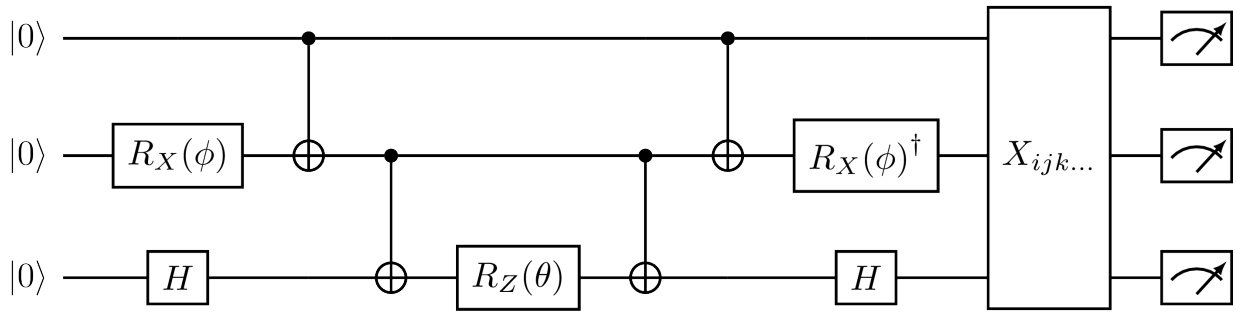


Fig. 8.2: Measurement circuit generated by ProtocolDirect, where $X_{ijk...}$ is the set of gates required to measure a commuting set of Pauli terms in the Hamiltonian.

8.2.2 ProtocolIndirect

Given a Hermitian operator expressed as a weighted sum of Pauli terms (`QubitOperator`), `ProtocolIndirect` constructs one measurement circuit per term in the operator where each term is controlled on an ancilla qubit. Only the ancilla qubit is measured. The expectation value of the operator is then the sum over the product of weighted Pauli strings with the measured expectations.

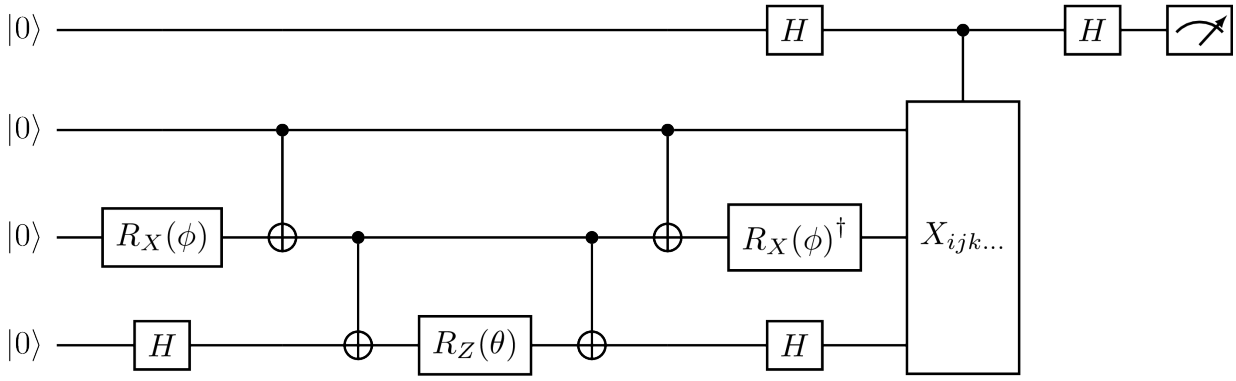


Fig. 8.3: Measurement circuit generated by ProtocolIndirect, where $X_{ijk...}$ is the set of gates required to measure a commuting set of Pauli terms in the Hamiltonian.

8.3 Protocols for overlaps

8.3.1 ProtocolVacuum

Let $\Psi_1(\theta) = \hat{U}_1|0\rangle$ where the state $\Psi_1(\theta)$ is prepared by applying the unitary circuit $\hat{U}_1$ to the vacuum state. The overlap between two states, $|\langle\Psi_0|\Psi_1\rangle|$ can be calculated by measuring the quantity $\langle 0|\hat{U}_0^\dagger\hat{U}_1|0\rangle$, where $\hat{U}_0^\dagger$ is the self-adjoint of the circuit which prepares state $|\Psi_0\rangle$. The state register is measured in the computational basis, where the overlap value is the probability of observing the vacuum state.

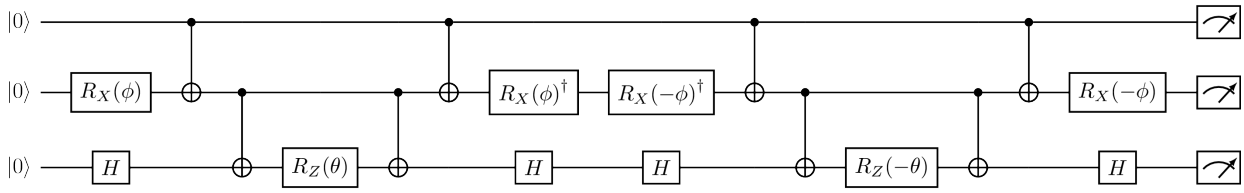


Fig. 8.4: Measurement circuit generated by ProtocolVacuum.

8.3.2 ProtocolCSP

This protocol implements the canonical swap test for evaluating orthogonality of two states. The procedure inserts a swap operation between registers of the bra and ket state, controlled on the ancilla qubit in the $|+\rangle$ state. Applying another Hadamard gate to the ancilla and measuring gives the probability of the bra and ket states being orthogonal as $P(|0\rangle) = \frac{1}{2} + \frac{1}{2}|\langle\psi|\phi\rangle|^2$.

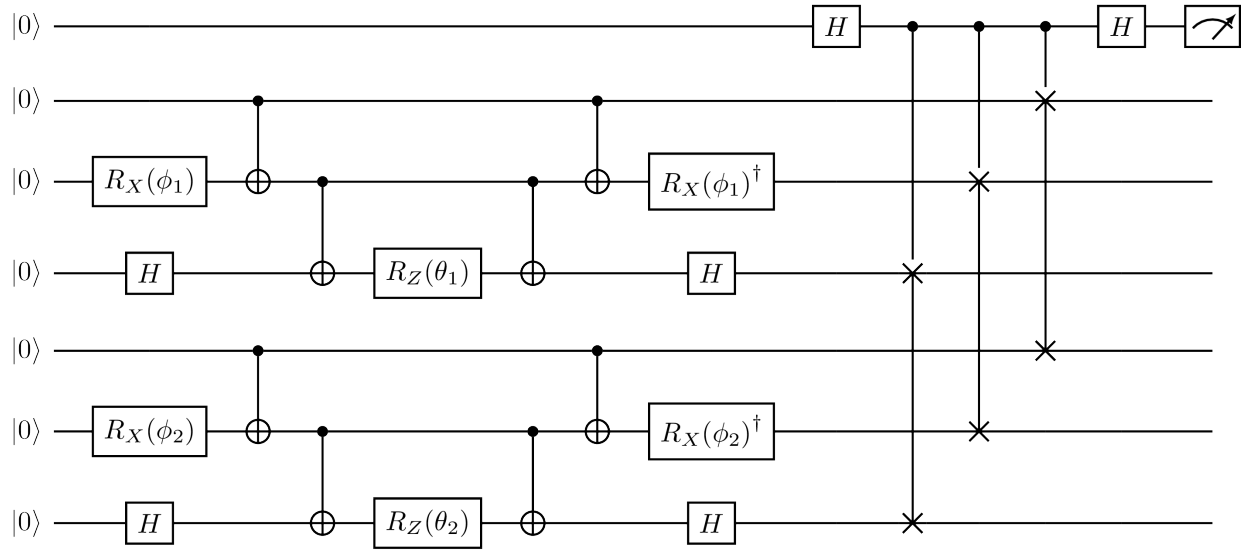


Fig. 8.5: Measurement circuit generated by ProtocolCSP.

8.3.3 ProtocolDSP

The ansatz circuits for the states are placed in parallel registers. Applying a CX ladder between the two state registers and measuring in the computational basis, an effective XOR operation is perfomed between the two registers. The total phase shift between the states is given as $\pi \sum_{i=1}^{n} M_i^1 M_i^2$, where M_i^r is the measurement outcome on the i-th qubit of the first or second state. A particular shot is valid, if the bitwise AND operation between the two state registers has even parity. An ancilla qubit is therefore not required.

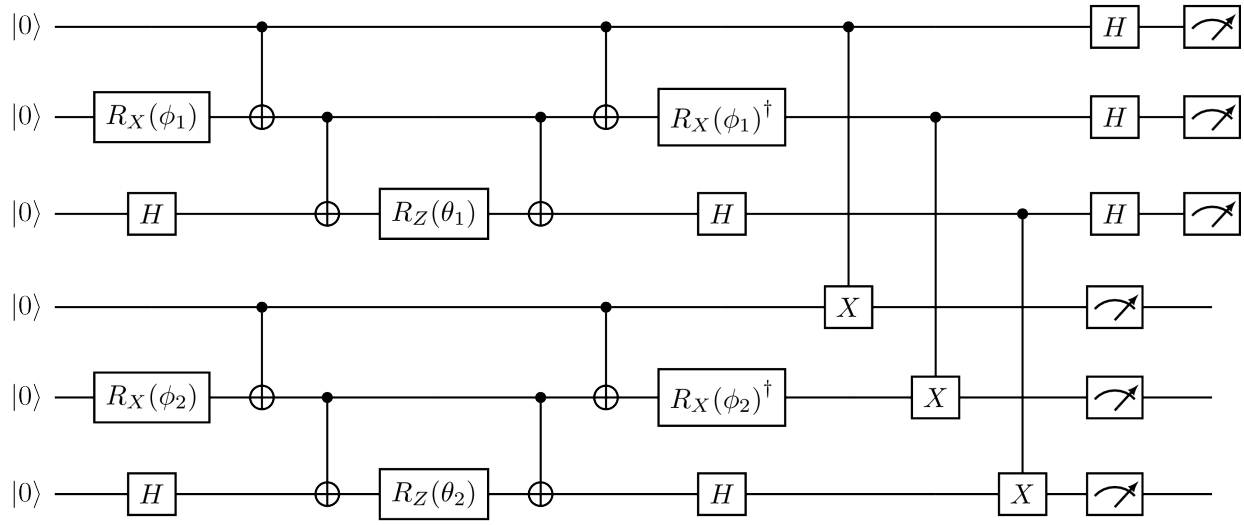


Fig. 8.6: Measurement circuit generated by ProtocolDSP.

8.4 Protocols for gradients

8.4.1 ProtocolHadamard(Direct/Indirect)(X/Y)

The circuit prepares the state

$$|\Psi_\mu\rangle = \frac{1}{2}[(|\sigma\rangle + i\hat{w}_n^P \hat{\sigma}_\mu^{(G)} \hat{W}_1^{n-1}|\phi_0\rangle) \otimes |0\rangle) + (|\sigma\rangle - i\hat{w}_n^P \hat{\sigma}_\mu^{(G)} \hat{W}_1^{n-1}|\phi_0\rangle) \otimes |1\rangle)] \tag{8.3}$$

Where $\hat{W}_n^k = U_k(\sigma_k) \dots U_n(\sigma_n)$ and is the ansatz circuit acting on state $|\phi_0\rangle$. G is the generator of Pauli gates representing the device register. Given $\langle i|\hat{\sigma}_y^{(a)}|j\rangle = (-1)^j \delta_{ij}$, we can measure $\langle\Psi_\mu|\hat{\sigma}_v^{(C)} \otimes \hat{\sigma}_y^{(a)}|\Psi_\mu\rangle$ where $\hat{\sigma}_v^{(C)}$ are the pauli terms of the Hamiltonian measured directly on the system register or projectively on an ancilla, and $\hat{\sigma}_y^{(a)}$ is the Pauli Y or Z gate applied to the ancilla register.

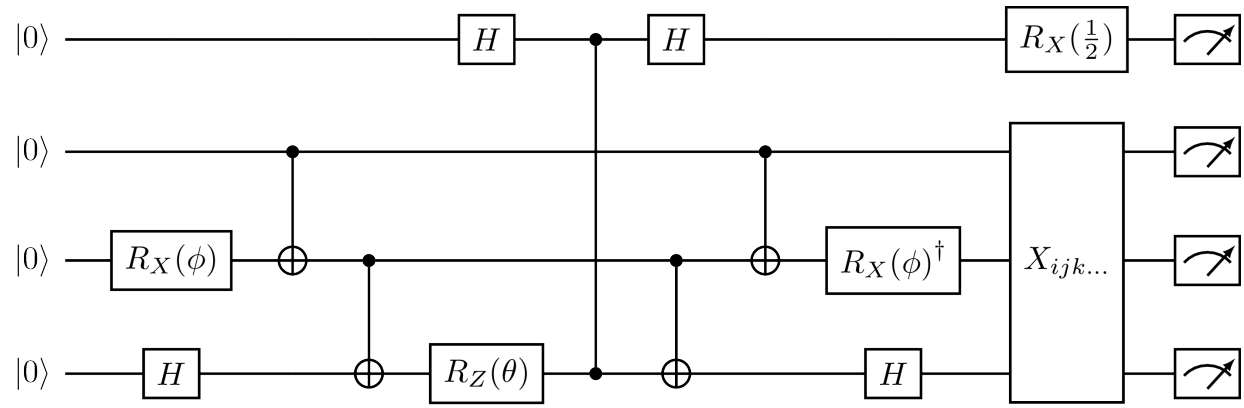


Fig. 8.7: Measurement circuit generated by ProtocolHadamardDirectY, where $X_{ijk...}$ is the set of gates required to measure a commuting set of Pauli terms in the Hamiltonian.

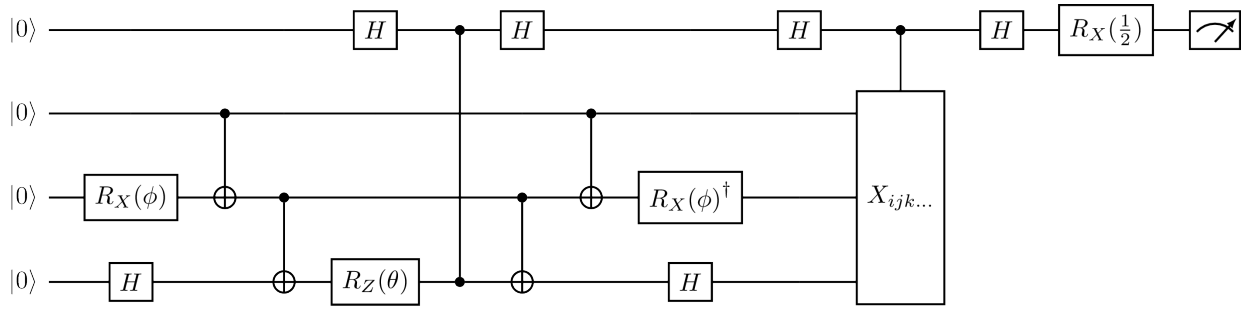


Fig. 8.8: Measurement circuit generated by ProtocolHadamardIndirectY, where $X_{ijk...}$ is the set of gates required to measure a commuting set of Pauli terms in the Hamiltonian.

8.4.2 ProtocolPhaseShift

The derivative of an expectation value expression can be written as

$$\partial_\mu f = \langle \Psi | G^\dagger \hat{O} (\partial_\mu G) | \Psi \rangle + h.c., \tag{8.4}$$

where $G(\mu) = e^{-i\mu G}$ is the gate generated from a Hermitian operator G and $\hat{O}$ is a Hermitian observable. We assume that a single parameter affects a single gate. The derivative of the gate is given as $\partial_\mu G = -iGe^{-i\mu G}$. Where the parametrised gate decomposition of a variational circuit unitary has two distinct eigenvalues, using the identity $G(\frac{\pi}{4r}) = \frac{1}{\sqrt{2}}(\mathbb{1} - ir^{-1}G)$, where r are the gate's eigenvalues, $\partial_\mu f$ can be estimated using two circuit evaluations by placing either the gate $G(\frac{\pi}{4})$ or $G(\frac{-\pi}{4})$ next to the gate being differentiated. Instead of placing additional gates, the phase difference is applied to the Z gate at the bottom of the Pauli gadget used for state preparation.

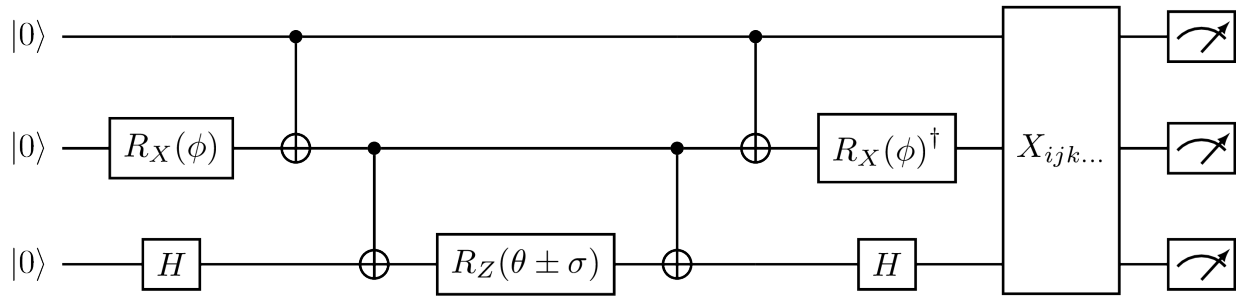


Fig. 8.9: Measurement circuit generated by ProtocolPhaseShift, where $X_{ijk...}$ is the set of gates required to measure a commuting set of Pauli terms in the Hamiltonian.

CHAPTER

NINE

SPACES, OPERATORS, AND STATES

The use of quantum computers for tackling problems in quantum chemistry involves a wide variety of quantum mechanical structures. Many of these are shared with quantum chemistry problems studied with classical computers. For example, the electronic Hamiltonian may be considered a second-quantized operator acting on a fermionic Hilbert space. However, quantum computing approaches often require objects and formalisms that are atypical to "conventional" quantum chemistry – for instance, fermionic operators and states must be mapped to operators acting on and states within a qubit Hilbert space.

InQuanto provides options for representing each of these objects. Broadly, three core types of objects are used - InQuanto operator and state classes represent operators and states within a given Hilbert space. Space objects (chiefly *FermionSpace* and *QubitSpace*) can be used to describe specific Hilbert spaces and then generate these states and operators. The relationship between these objects is depicted in Fig. 9.1.

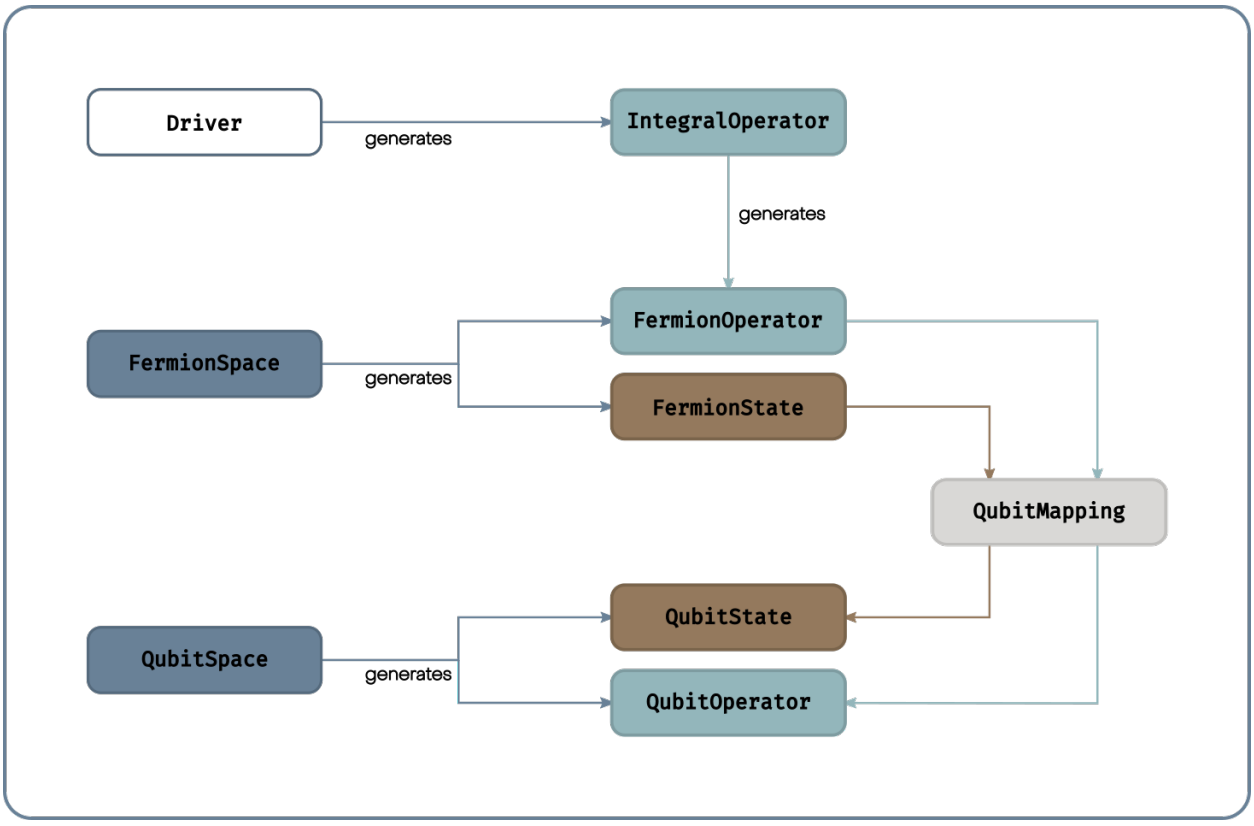


Fig. 9.1: A schematic of the vector space, state and operator classes in InQuanto. Spaces generate operators and states; fermion-to-qubit mappings transform fermionic states and operators to qubit states and operators.

We first discuss *fermionic* and *qubit* spaces and how operators and states can be *mapped* between the two. An option for the parafermionic [11] space is also provided by the `ParaFermionSpace` object. This mapping is described in the *Iterative Qubit-Excitation Based VQE algorithm* section. We also consider *integral operators* - a class for the efficient manipulation of molecular orbital integral tensors. Finally, we consider *orbital transformers* and optimization – several InQuanto classes for the convenient manipulation of molecular orbitals bases.

9.1 Fermionic Spaces

Given a system of N electrons described by an orthonormal basis of Q molecular spin orbitals, or momentum space modes, where $Q \geq N$, one can construct a fermionic Fock space [5], or "fermion space". Each basis vector of this abstract vector space corresponds to an occupation number (ON) vector

$$|\boldsymbol{\eta}\rangle = |\eta_0, \eta_1, ..., \eta_P, ..., \eta_{Q-1}\rangle \tag{9.1}$$

where an occupation number $\eta_P = 1$ (0) if orbital P is occupied (unoccupied). Each ON vector represents a canonically ordered N-electron Slater determinant. A fermion space is an abstract linear vector space equipped with the usual properties of an inner product and resolution of identity, for a given set of orbitals or modes. An arbitrary vector in this space corresponds to a linear combination of ON vectors

$$|\boldsymbol{\psi}\rangle = \sum_{\boldsymbol{\eta}} \psi_{\boldsymbol{\eta}} |\boldsymbol{\eta}\rangle \tag{9.2}$$

if the orbitals or modes used to construct the ON vectors are orthonormal, such that the ON vectors obey the relation

$$\langle \boldsymbol{\eta}' | \boldsymbol{\eta} \rangle = \prod_{P=0}^{Q-1} \delta_{\eta'_P, \eta_P} \tag{9.3}$$

This is a useful feature of fermion spaces: a well defined but zero overlap between ON vectors with different values of $N = \sum_{P=0}^{Q-1} \eta_P$ and $N' = \sum_{P=0}^{Q-1} \eta'_P$ allows for a consistent treatment of systems with different numbers of electrons. Thus the ON vectors form an orthonormal basis in the 2^Q-dimensional fermion space, which in principle can be decomposed into a direct sum of fermionic subspaces $\mathcal{F}(Q, N)$, where each subspace represents a different number of electrons distributed over the Q orbitals.

$$\mathcal{F}(Q) = \bigoplus_{N=0}^{Q} \mathcal{F}(Q, N) \tag{9.4}$$

The dimension of each $\mathcal{F}(Q, N)$ subspace is equal to the number of terms in the FCI expansion $\binom{Q}{N}$, such that the exact (FCI) wavefunction for N electrons (in a given basis) can be expressed as a vector in $\mathcal{F}(Q, N)$.

Anticommuting creation and annihilation operators act on the fermion space as linear maps between vectors within the space. From these operators, 1- and 2-body interactions can be defined, along with the fermionic ON vectors, and thus the entire fermionic problem can be described once the N-electron subspace is specified. In InQuanto, this logic is followed in the `FermionSpace` class which represents the fermionic Fock space $\mathcal{F}(Q)$, and contains related utilities for the construction of fermionic interaction operators (represented by the `FermionOperator` class) and ON vectors (represented by the `FermionState` class). In the next subsection `FermionSpace` is discussed, while `FermionOperator` and `FermionState` are covered in later sections.

9.1.1 The `FermionSpace` Class

This class provides a range of tools related to the fermionic Fock space. These tools include the definition of the space itself, and operations on this space related to the generation of fermionic operators and for specification of occupations. This class is a child of `OccupationSpace`, which is agnostic to particle statistics or commutation algebra. Instead, the (anti-)commutation algebra is built into the logic of the child class, in this case *`FermionSpace`*. The parent `OccupationSpace` class allows for consistent treatment of quantum numbers and basis function indexing, such that operators generated by an instance of the child class (*`FermionSpace`*) are guaranteed to have consistent indexing.

Various fermionic quantum operators can be constructed from the methods in this class. These include the number operators, spin operators, 1- and 2-body operators in various forms, and the antihermitian operators needed for unitary coupled cluster ansatz generation. For example, the UCC terms for minimal basis H_2 (with 4 spin-orbitals) are defined as follows

```
from inquanto.spaces import FermionSpace
from inquanto.states import FermionState

# create a FermionSpace object for, say, H2
space = FermionSpace(4)
state = FermionState([1, 1, 0, 0])

# with the state and space objects we can construct operators using native functions,␣
↪for example:
singles = space.construct_single_ucc_operators(state)
doubles = space.construct_double_ucc_operators(state)
uccsd_excitations = singles + doubles

print("number of uccsd excitations for minimal basis h2:", len(uccsd_excitations))
```

```
number of uccsd excitations for minimal basis h2: 3
```

In this case, the occupations specified by `state` are needed to construct the UCC operators, though this is not always required depending on the operators being generated. The operators are returned as *`FermionOperator`* objects (*`FermionOperator`* and *`FermionState`* are discussed in more detail later). Several types of operators can be generated by the various methods of *`FermionSpace`*, including various forms of number operators, excitation operators and symmetry operators. An exhaustive list of these operators can be found in *`the API reference`*.

Once the space and occupations are defined, information relevant to the ON vector can be displayed using the *`print_state()`* method:

```
from inquanto.spaces import FermionSpace
from inquanto.states import FermionState

# create a FermionSpace object for, say, H2
space = FermionSpace(4)
state = FermionState([1, 1, 0, 0])

print("fermionic fock state:")
space.print_state(state)
```

```
fermionic fock state:
 0 0a         :  1
 1 0b         :  1
 2 1a         :  0
 3 1b         :  0
```

The left-most column refers to spin-orbital indexes, the next column shows the spatial orbital indexes with the alpha (`a`)

and beta (b) spin labels, while the right-most column shows the occupation of each spin orbital.

Typically, operators generated using the *FermionSpace* methods will preserve particle number and spin conservation symmetries. In addition to these, *FermionSpace* can also optionally handle point group symmetry information. By passing in point group symmetries when instantiating the *FermionSpace* class object, we can generate only those excitations that are non-zero by symmetry. Below, the single excitations of H2 are symmetry forbidden, leaving only one double.

```
from inquanto.spaces import FermionSpace
from inquanto.states import FermionState
from inquanto.symmetry import PointGroup

# create a FermionSpace object for, say, H2
space = FermionSpace(4)
state = FermionState([1, 1, 0, 0])

# we can also pass symmetry information to remove redundant excitations
space = FermionSpace(
    4, point_group=PointGroup("D2h"), orb_irreps=["Ag", "Ag", "B1u", "B1u"]
)
singles = space.construct_single_ucc_operators(state)
doubles = space.construct_double_ucc_operators(state)
uccsd_excitations = singles + doubles

print("number of allowed uccsd excitations for minimal basis h2:", len(uccsd_
↪excitations))
```

```
number of allowed uccsd excitations for minimal basis h2: 1
```

This potentially reduces the size of the problem, for example the number of parameters in variational algorithms like VQE. When a *FermionSpace* object is generated from a driver, it will typically include point group information if the driver utilised this information in performing precursor classical computation. The *FermionSpace* classes can be used to generate explicit operator representations of the symmetries of the system with the `symmetry_sector` method - usage of this is detailed in the *symmetry* section.

9.1.2 Fermion Operators & States

In second quantization, operators acting on fermionic states are represented by linear combinations and tensor products of creation and annihilation operators. These obey the anticommutation relations

$$\begin{aligned} \{\hat{f}_P^\dagger, \hat{f}_{P'}^\dagger\} &= 0 \\ \{\hat{f}_P, \hat{f}_{P'}\} &= 0 \\ \{\hat{f}_P^\dagger, \hat{f}_{P'}\} &= \delta_{P,P'} \end{aligned} \tag{9.5}$$

Where $\{A, B\} = AB + BA$. Mathematically, these operators couple ON vectors belonging to different subspaces $\mathcal{F}(Q, N)$. Thus, a given arrangement of occupations can be used to build an ON vector by applying a product of creation operators to the vacuum state, such that creation operators are only applied to orbitals or modes that should be unoccupied

$$|\boldsymbol{\eta}\rangle = \prod_{P=0}^{Q-1} (\hat{f}_P^\dagger)^{\eta_P} |0_0, ..., 0_P, ..., 0_{Q-1}\rangle \tag{9.6}$$

In InQuanto, vectors in $\mathcal{F}(Q, N)$ space are represented by the *FermionState* object. This object represents the state as a dictionary, with the occupation configuration(s) (the ON vector(s)) as keys, and configuration coefficient(s) as values. Each occupation configuration is represented by a *FermionStateString*, which takes as argument a list of integers and returns a bitstring corresponding to the occupations (ordered by spin orbitals or modes indices). For example, for an ON vector of 4-spin orbitals with the first 2 orbitals occupied

```
from inquanto.states import FermionStateString

on_vector = FermionStateString([1, 1, 0, 0])

print(on_vector)
print(list(on_vector))
```

```
[1, 1, 0, 0]
[1, 1, 0, 0]
```

The *FermionState* can be initialized from a dict of {*FermionStateString*: coefficient} key-value pairs. If only one configuration (single reference) is the case, then a list of integers suffices

```
from inquanto.states import FermionState, FermionStateString

on_vector = FermionStateString([1, 1, 0, 0])
state = FermionState({on_vector: 1.0})
print(state)
state = FermionState([1, 1, 0, 0])
print(state)
```

```
(1.0, [1, 1, 0, 0])
(1.0, [1, 1, 0, 0])
```

Note that the *FermionState* object can also be generated as a method of *FermionSpace*, by passing the occupation list as an argument

```
from inquanto.spaces import FermionSpace
from inquanto.states import FermionState

space = FermionSpace(4)
state = space.generate_occupation_state_from_list([1, 1, 0, 0])
```

FermionState objects may be operated on with standard linear algebraic operations to return new *FermionState* objects, and may be iterated over.

```
new_state = 2.0 * state + FermionState([0,0,1,1])
for x in new_state.split():
    print(x)
```

```
(2.0, [1, 1, 0, 0])
(1.0, [0, 0, 1, 1])
```

Products of equal numbers of creation and annihilation operators are number conserving, and these products can be used to build general fermionic operators. For example the number operator $\hat{N}_P = \hat{f}_P^\dagger \hat{f}_P$ has eigenvalues corresponding to the occupation of each spin orbital and its eigenvectors are the ON vectors

$$\hat{N}_P|\boldsymbol{\eta}\rangle = \eta_P|\boldsymbol{\eta}\rangle \tag{9.7}$$

The total number operator yields the total number of electrons in the system

$$\hat{N}|\boldsymbol{\eta}\rangle = \sum_{P=0}^{Q-1} \hat{N}_P|\boldsymbol{\eta}\rangle = N|\boldsymbol{\eta}\rangle \tag{9.8}$$

In InQuanto, these operators are accessed from methods of the *FermionSpace* class. Spin-resolved versions of the total number operator are also available. For example, with 4 spin orbitals:

```
from inquanto.spaces import FermionSpace

space = FermionSpace(4)
n_p = space.construct_orbital_number_operators()
n = space.construct_number_operator()
n_alpha = space.construct_number_alpha_operator()
n_beta = space.construct_number_beta_operator()
```

An instance of the *FermionOperator* class is returned by these number operator methods. Analogous to *FermionState*, the *FermionOperator* stores the relevant information as a dictionary in which the keys are *FermionOperatorString* objects and the values are their coefficients. *FermionOperatorString* uses a `tuple` of pairs of integers, in which the first integer represents the spin orbital or mode and the second integer is 1 (0) for creation (annihilation) operators. *FermionOperatorString* can also be instantiated directly from a `string` using the `from_string()` method. As an example, consider an excitation operator in which an electron is annihilated from orbital 2 and created on orbital 3 ($\hat{f}_3^\dagger \hat{f}_2$), the corresponding *FermionOperator* can be specified from the *FermionOperatorString* using `tuple`, a `list` of `tuple`, or a `dict`

```
from inquanto.operators import FermionOperator, FermionOperatorString

f_op_string = FermionOperatorString(((3, 1), (2, 0)))
print(f_op_string)
f_op_string = FermionOperatorString.from_string("F3^ F2")
print(f_op_string)
f_op1 = FermionOperator(f_op_string, 1.0)
f_op2 = FermionOperator([(1.0, f_op_string)])
f_op3 = FermionOperator({f_op_string: 1.0})
print(f_op1)
print(f_op2)
print(f_op3)
```

```
F3^ F2
F3^ F2
(1.0, F3^ F2 )
(1.0, F3^ F2 )
(1.0, F3^ F2 )
```

Various methods for manipulating *FermionOperator* are also available in InQuanto. The following example (fermion_operator.) summarises the main features of InQuanto's *FermionOperator* and *FermionState* objects and their usage.

```
from inquanto.operators import FermionOperator
from inquanto.operators import FermionOperatorString

# construct a fermion operator
op0 = FermionOperator({FermionOperatorString([(0, 0)]): 1.0})

# now multiply by its adjoint
op = op0 * op0.dagger()

# does it commute with itself? Yes!
print("commutator of op with itself:", op.commutator(op))
print("does op commute with itself?", op.commutes_with(op))

# instantiate from string
op2 = FermionOperator({FermionOperatorString.from_string("F1 F2^"): 3.5})
```

(continues on next page)

(continued from previous page)

```
# is this normal ordered?
print("is op2 normal ordered?", op2.is_normal_ordered())
op2 = op2.normal_ordered()
print("what about now?", op2.is_normal_ordered())
print("is operator number conserving?", op2.is_two_body_number_conserving())

# sum of operators so far:
op3 = op + op2
print("op3 =", op3)
# remove terms with coefficients of absolute value < 3
print("truncated op3 =", op3.truncated(3.0))

# map this fermion operator to a qubit operator
print("JW mapped op3 =", op3.qubit_encode())

# we can apply the operators kets and bras defined in the fermion space to retrieve a␣
↪new FermionState
from inquanto.states import FermionState

fock_state = FermionState([1, 1, 0, 0])
print("<HF|op3 = ", op3.apply_bra(fock_state))
print("op3|HF> = ", op3.apply_ket(fock_state))
```

```
commutator of op with itself: (0.0, ), (0.0, F0^ F0 )
does op commute with itself? True
is op2 normal ordered? False
what about now? True
is operator number conserving? True
op3 = (1.0, F0  F0^), (-3.5, F2^ F1 )
truncated op3 = (-3.5, F2^ F1 )
```

```
JW mapped op3 = (0.5, ), (0.5, Z0), (-0.875j, Y1 X2), (-0.875, X1 X2), (-0.875, Y1␣
↪Y2), (0.875j, X1 Y2)
<HF|op3 =  (0)
op3|HF> =  (-3.5, [1, 0, 1, 0])
```

9.2 Qubit spaces, operators and states

For the analysis of many quantum algorithms, it is useful to work in a representation above the level of quantum circuits decomposed into some set of quantum gate primitives. Similarly to the *FermionSpace* class, qubit Hilbert spaces are represented in InQuanto using the *QubitSpace* class.

```
from inquanto.spaces import QubitSpace
qubit_space = QubitSpace(4)
```

The *QubitSpace* object represents the 2^N dimensional Hilbert space of an N-qubit system. For chemistry purposes, operators and states in a qubit Hilbert space are typically generated from fermionic operators and states, and thus the *QubitSpace* class does not have as many methods for directly generating operators and states as the fermionic equivalent. However, methods for finding symmetry operators of a given qubit operator are included, as discussed in the *symmetry* section.

9.2.1 Qubit Operators

Conventionally, operators acting upon states of qubits are typically represented as linear combinations of *Pauli strings*

$$\hat{O} = \sum_i h_i P_i \tag{9.9}$$

where each Pauli string

$$P_i = \bigotimes_{n=0}^{N} p_n \tag{9.10}$$

is a tensor product of single qubit Pauli $p_n \in \{I, X, Y, Z\}$ operators and N is the number of qubits. Any operator acting on a register of qubits can be decomposed in this form. This representation is particularly useful when considering quantum chemistry problems, where operators acting on states of fermions must be mapped to operators acting on states of qubits prior to simulation. In InQuanto, operators of this form are implemented using the *QubitOperator* class. In chemistry, instances of this will frequently derive from mapping a *FermionOperator*, however they can be directly instantiated in a number of ways (detailed in the API reference for *QubitOperator*. For example, we can construct by providing a string detailing which Pauli operator is acting on each qubit:

```
from inquanto.operators import QubitOperator
op = QubitOperator("X0", 1.0)
print(op)
```

```
(1.0, X0)
```

Basic linear algebra is supported on these objects, along with various useful tools detailed in the API reference for *QubitOperator*.

```
op1 = QubitOperator("X0",1.0)
op2 = QubitOperator("Z0",1.0)
op3 = op1 + op2
print('Op1 + Op1:', op3)
print('Op1 conjugated:', op1.dagger())
print('Commutator:', op1.commutator(op2))
print('Sparse matrix form:', op1.to_sparse_matrix(1))
print('Identity:', QubitOperator.identity())
```

```
Op1 + Op1: (1.0, X0), (1.0, Z0)
Op1 conjugated: (1.0, X0)
Commutator: (-2j, Y0)
Sparse matrix form:   (1, 0)        (1+0j)
  (0, 1)        (1+0j)
Identity: (1.0, )
```

Several helper methods for determining properties of an operator are also available:

```
print('Is operator Hermitian?', op3.is_hermitian())
print('Is operator anti-Hermitian?', op3.is_antihermitian())
print('Is operator unitary?', op3.is_unitary())
print('Is operator unit-norm?', op3.is_unit_norm(order=2))
print('Is operator self-inverse?', op3.is_self_inverse())
```

```
Is operator Hermitian? True
Is operator anti-Hermitian? False
Is operator unitary? False
```

(continues on next page)

(continued from previous page)

```
Is operator unit-norm? False
Is operator self-inverse? False
```

Single Pauli strings (tensor products of individual Pauli operators acting on single qubits) can be described with the *QubitOperatorString* class. Internally, a *QubitOperator* consists of a `dict` between these *QubitOperatorString* objects and their corresponding coefficients in the linear combination representing the operator. The terms may be extracted with the *pauli_strings()* property, whereas the coefficients may be extracted as a list with the *coefficients()* property.

```
print('Operator coefficients:')
print(op3.coefficients)
print('Pauli strings:')
print(op3.pauli_strings)
```

```
Operator coefficients:
[1.0, 1.0]
Pauli strings:
[(Xq[0]), (Zq[0])]
```

Often when considering quantum algorithms, it is useful to describe qubit operators in their symplectic representation. This consists of a $M * 2N$ binary matrix where M is the number of independent Pauli strings in the operator, and N is the number of qubits. The leftmost half of this matrix designates whether a Pauli X is acting on qubit n in term m, and the rightmost half designates whether a Pauli Z is acting on the same qubit in the same term. As $Y = iXZ$, both leftmost and rightmost entries are 1 if a Pauli Y is present. Note that in this representation, information regarding the coefficients of each term must be stored independently.

```
print(op1.symplectic_representation())
```

```
[[ True False]]
```

InQuanto also contains a class for dealing with sets of Pauli operators that are not linearly combined – the *QubitOperatorList*. Each separate *QubitOperator* in the *QubitOperatorList* may be associated with an additional scalar. This is particularly useful when describing sequences of exponentiated Pauli operators, for example when considering variational ansatzae.

```
from sympy import sympify
from inquanto.operators import QubitOperatorList
qol = QubitOperatorList([(sympify("a"),op1),(sympify("b"),op2)])
print(qol)
```

```
a          [(1.0, X0)],
b          [(1.0, Z0)]
```

Finally, note that these classes are also useful for containing Trotter sequences. The *QubitOperator* class contains various helper methods for generating such sequences.

```
op1 = QubitOperator("X0",1.0)
op2 = QubitOperator("Z0",1.0)
op3 = op1 + op2
op_trotterized = op3.trotterize(trotter_number=2)
print(op_trotterized)
```

```
0.5        [(1.0, X0)],
0.5        [(1.0, Z0)],
```

(continues on next page)

(continued from previous page)

```
0.5        [(1.0, X0)],
0.5        [(1.0, Z0)]
```

The *API reference* details these methods, and provides a full breakdown of the other functionality available for the `QubitOperatorList` classes.

9.2.2 Qubit States & Expectation Values

A register of N qubits corresponds to a $\mathcal{C}^{2^N}$ Hilbert space. As such, it can be represented with a 2^N dimensional vector of complex numbers. Clearly, generating such a vector is not scalable on a classical computer (otherwise we wouldn't need quantum computers), but this approach can be practical for small N. For some tasks (for instance, if we are simulating the action of a known Clifford operator), we can guarantee that a given state will have polynomial support. In this case, we can efficiently store the state in a sparse state vector. InQuanto provides an alternative to an explicit sparse state vector representation of states in the form of the *QubitState* class, instances of which consist of linear combinations of *QubitStateString* objects.

```
from inquanto.states import QubitState
qubit_state = QubitState([1,1,0,0],1.)
print(qubit_state)
```

```
(1.0, [1, 1, 0, 0])
```

These can be converted to state vector representations:

```
state_vector = qubit_state.to_ndarray()
print(state_vector)
```

```
[[0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]
 [1.+0.j]
 [0.+0.j]
 [0.+0.j]
 [0.+0.j]]
```

Most importantly for the purposes of analysing quantum algorithms, the *QubitState* representation allows for performing linear algebra with other *QubitState* and *QubitOperator* objects.

```
overlap = qubit_state.vdot(qubit_state)
expectation_value = op1.state_expectation(qubit_state)
print('Overlap:', overlap)
print('Expectation value:', expectation_value)
```

```
Overlap: 1.0
Expectation value: 0
```

QubitState objects are implicitly sparse, in contrast to full dense statevector representations. Performing calculations in this way allows for the analysis of qubit states and operators without the need to either generate full circuit representations, or the expensive generation of full 2^{2N} matrix representations of operators.

9.3 Fermion-to-Qubit Mapping

In quantum chemistry, we are most often concerned with the properties of electrons. As electrons are fermions, second-quantized fermionic creation and annihilation operators obey the fermionic anticommutation relations:

$$\{\hat{a}_i^\dagger, \hat{a}_j^\dagger\} = 0, \{\hat{a}_i, \hat{a}_j\} = 0, \{\hat{a}_i^\dagger, \hat{a}_j\} = \delta_{i,j} \tag{9.11}$$

This implicitly restricts the occupation number of a given fermionic mode to $\{0, 1\}$. Conversely, qubits can be considered to be *paraparticles*. Like fermionic modes, each qubit is a two-level system. However, the above fermionic anticommutation relations are not obeyed:

$$\begin{aligned} |1\rangle\langle 0|_i &= X_i - iY_i, \\ |0\rangle\langle 1|_i &= X_i + iY_i \\ \{X_i - iY_i, X_j + iY_j\} &\neq \delta_{i,j} \end{aligned} \tag{9.12}$$

As such, in order to use a system of qubits to simulate a system of fermions, the fermionic anticommutation relations must be encoded. An encoding scheme consists of a linear map between states and operators in the fermionic Hilbert space and the states and operators in the qubit Hilbert space. For quantum chemistry purposes, mapping both states and operators is important - for example, in a canonical VQE calculation, both the Hamiltonian operator and the reference state must be mapped to the qubit space.

In InQuanto, fermion-to-qubit mappings are stored in the *inquanto.mappings* module. Four mappings are included in the current version:

- *Jordan-Wigner transformation*
- *Bravyi-Kitaev mapping*
- *Parity mapping*
- *"Paraparticular" mapping* – this does not encode fermionic statistics

In order to map an InQuanto *FermionOperator* to a *QubitOperator*, the *operator_map()* method can be used:

```
from inquanto.operators import FermionOperator,FermionOperatorString
from inquanto.mappings import QubitMappingJordanWigner
fermion_operator = FermionOperator(FermionOperatorString([(1, 0)]), 0.5)
qubit_operator = QubitMappingJordanWigner.operator_map(fermion_operator)
print(qubit_operator)
```

```
(0.25, Z0 X1), (0.25j, Z0 Y1)
```

and similarly, to map *FermionState* objects to *QubitState* objects, the *state_map()* method is used:

```
from inquanto.states import FermionState
fermion_state = FermionState([0,0,1,1])
qubit_state = QubitMappingJordanWigner.state_map(fermion_state)
```

Note that both of these methods include an optional argument specifying the register of tket `Qubit` objects that comprise the target qubit space. Without this argument provided (as above), the mapping will if possible infer that a minimally sized register is to be used, indexed incrementally from 0. For some mappings (for example, the Bravyi-Kitaev mapping), this information cannot be inferred. If these mappings are used, it is necessary to specify the qubit register.

```
from pytket import Qubit
from inquanto.mappings import QubitMappingBravyiKitaev
qubit_operator_bk = QubitMappingBravyiKitaev.operator_map(fermion_operator,[Qubit(i)␣
↪for i in range(8)])
print(qubit_operator_bk)
```

```
(0.25, Z0 X1 X3 X7), (0.25j, Y1 X3 X7)
```

Finally, when performing state vector simulations one may wish to represent fermionic and qubit states in the form of full dense or sparse vectors of complex numbers. The *inquanto.mappings* classes support this, and will return in the same representation as the input.

```
from numpy import array
from scipy.sparse import csc_matrix
dense_fermionic_state = array([[1],[0],[0],[0]])
sparse_fermionic_state = csc_matrix(dense_fermionic_state)
dense_qubit_state = QubitMappingJordanWigner.state_map(dense_fermionic_state,
↪qubits=[Qubit(i) for i in range(4)])
sparse_qubit_state = QubitMappingJordanWigner.state_map(sparse_fermionic_state,
↪qubits=[Qubit(i) for i in range(4)])
print(dense_qubit_state)
print(sparse_qubit_state)
```

```
[[1]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]
 [0]]
  (0, 0)        1
```

9.3.1 Custom mappings

For advanced usage, it is possible to define custom mapping schemes. The *QubitMapping* class is a superclass containing the logic of the *operator_map()* and *state_map()* methods. Custom mappings can be designed by subclassing this and implementing the *update_set()* , *parity_set()* , *rho_set()* and *state_map_matrix()* methods. The first three of these methods return sets of qubits according to the formalism of Seeley, Richard and Love [13]. The *state_map_matrix()* method returns a matrix which transforms a binary column vector representation of the binary index corresponding to a given basis state in the fermionic space, to a similar vector in the qubit space. In the flip/update/parity/rho set formalism, fermionic creation and annihilation operators are mapped according to the following relation:

$$\begin{aligned} a_i^\dagger &\to \frac{1}{2}(X_{U(i)}X_iZ_{P(i)} - X_{U(i)}Y_iZ_{\rho(i)}) \\ a_i &\to \frac{1}{2}(X_{U(i)}X_iZ_{P(i)} + X_{U(i)}Y_iZ_{\rho(i)}) \end{aligned} \tag{9.13}$$

where $U(i)$ is the update set, $P(i)$ is the parity set, $U(i)$ is the rho set for orbital i. As an example, for the Jordan-Wigner mapping:

$$\begin{aligned} a_i^\dagger &\to \frac{1}{2}\left(X_i - iY_i\right) \otimes Z^{\otimes i} \\ a_i &\to \frac{1}{2}\left(X_i + iY_i\right) \otimes Z^{\otimes i} \end{aligned} \tag{9.14}$$

Comparing this against (9.13) we see the update and flip set are always empty, whereas the parity set consists of all qubits with index less than i. *state_map_matrix()* for the Jordan-Wigner transformation will always return the identity matrix, as state $|n\rangle$ in the fermionic space is always mapped to state $|n\rangle$ in the qubit space.

9.4 Integral Operators

We have described *above* how InQuanto stores and handles the fermionic Hamiltonian via the *FermionOperator* class. The *FermionOperator* class stores those operators and numeric integral values as items of a dictionary. Such storage is convenient for manipulating and accessing individual terms of the Hamiltonian, but is not tailored well to linear algebra operations, such as integral transformation.

The InQuanto *operators* module provides integral operator classes for storing and manipulating chemistry integrals (h_{ij} and h_{ijkl}, typically molecular orbital integrals) as algebraic objects. The standard integral operators are the *ChemistryRestrictedIntegralOperator* and *ChemistryUnrestrictedIntegralOperator* classes, for spin-restricted and spin-unrestricted formalisms respectively. In the restricted case, one-body integrals are stored in a 2-dimensional `numpy` array with shape `(n, n)`, and two-body integrals in a 4-dimensional array with shape `(n, n, n, n)`, where `n` is the number of spatial orbitals. In the unrestricted case, the ↑ and ↓ spin channels have independent integrals. Hence, two 2-dimensional arrays store the one-body integrals, one for each spin channel, and four 4-dimensional arrays store all spin-configurations of the two-body integrals (⇈⇈, ⇊⇊, ⇈⇊ and ⇊⇈).

The *inquanto-pyscf* extension is the primary tool for generating integral operators for a chemical system, though integral operators may also be instantiated directly with `numpy` arrays (see the *ChemistryRestrictedIntegralOperator* and *ChemistryUnrestrictedIntegralOperator* constructors). Below, we use the *Express* module to load in a pre-computed integral operator for LiH:

```
from inquanto.express import load_h5
lih_sto3g = load_h5('lih_sto3g.h5', as_tuple = True)
integral_operator = lih_sto3g.hamiltonian_operator
print(integral_operator.df())
```

```
      Coefficients                Terms
0         1.050650
1        -4.746695              F0^ F0
2         0.109103              F0^ F2
3         0.168024              F0^ F4
4        -0.026783             F0^ F10
...            ...                  ...
1496      0.009754    F11^ F10^ F8  F9
1497      0.009754    F11^ F10^ F8  F9
1498      0.228088  F11^ F10^ F10  F11
1499      0.228088  F11^ F10^ F10  F11
1500     -0.935480            F11^ F11

[1501 rows x 2 columns]
```

where the *df()* method produces a `pandas` dataframe of all integrals and their corresponding fermion operator terms. Integral operators may be converted directly into a qubit Hamiltonian with the *qubit_encode()* method, which uses Jordan-Wigner mapping by default:

```
qubit_hamiltonian = integral_operator.qubit_encode()
print("LiH STO-3G JW qubit hamiltonian:\n", qubit_hamiltonian)
```

```
LiH STO-3G JW qubit hamiltonian:
 (-4.107195904002317, ), (1.0104347328306078, Z0), (0.01507400738908326, Y0 Z1 Y2),␣
↪(0.01507400738908326, X0 Z1 X2), (0.025702830769369776, Y0 Z1 Z2 Z3 Y4), (0.
↪025702830769369776, X0 Z1 Z2 Z3 X4), (-0.0002848256587932635, Y0 Z1 Z2 Z3 Z4 Z5 Z6␣
↪Z7 Z8 Z9 Y10), (-0.0002848256587932635, X0 Z1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 X10), (-0.
↪11537391270277789, Z2), (-0.0077218092065717055, Y2 Z3 Y4), (-0.0077218092065717055,
↪ X2 Z3 X4), (0.006251930298691508, Y2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Y10), (0.
↪006251930298691508, X2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 X10), (-0.19742105041852576, Z4), (0.
↪014552642690300127, Y4 Z5 Z6 Z7 Z8 Z9 Y10), (0.014552642690300127, X4 Z5 Z6 Z7 Z8␣
↪Z9 X10), (-0.2289838304884416, Z6), (-0.2289838304884415, Z8), (-0.3968880238789371,
↪ Z10), (1.0104347328306078, Z1), (0.015074007389083259, Y1 Z2 Y3), (0.
↪015074007389083259, X1 Z2 X3), (0.02570283076936979, Y1 Z2 Z3 Z4 Y5), (0.
↪02570283076936979, X1 Z2 Z3 Z4 X5), (-0.0002848256587932618, Y1 Z2 Z3 Z4 Z5 Z6 Z7␣
↪Z8 Z9 Z10 Y11), (-0.0002848256587932618, X1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 X11), (-0.
↪11537391270277789, Z3), (-0.0077218092065717705, Y3 Z4 Y5), (-0.0077218092065717705,␣
↪X3 Z4 X5), (0.00625193029869151, Y3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Y11), (0.
↪00625193029869151, X3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 X11), (-0.19742105041852576, Z5), (0.
↪014552642690300127, Y5 Z6 Z7 Z8 Z9 Z10 Y11), (0.014552642690300127, X5 Z6 Z7 Z8 Z9␣
↪Z10 X11), (-0.2289838304884416, Z7), (-0.2289838304884415, Z9), (-0.
↪3968880238789371, Z11), (0.09086756126768071, Z0 Z2), (-3.508475660225329e-05, Z0␣
↪Y2 Z3 Y4), (-3.508475660225329e-05, Z0 X2 Z3 X4), (0.005481966935653534, Z0 Y2 Z3␣
↪Z4 Z5 Z6 Z7 Z8 Z9 Y10), (0.005481966935653534, Z0 X2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 X10), (0.
↪09360084027749466, Z0 Z4), (-0.004705162241416893, Z0 Y4 Z5 Z6 Z7 Z8 Z9 Y10), (-0.
↪004705162241416893, Z0 X4 Z5 Z6 Z7 Z8 Z9 X10), (0.09662264985846813, Z0 Z6), (0.
↪09662264985846812, Z0 Z8), (0.08857759609112856, Z0 Z10), (0.0033316360573752354,␣
↪Y0 Z1 Z3 Y4), (0.0033316360573752354, X0 Z1 Z3 X4), (0.0029361135995872667, Y0 Z1␣
↪Z3 Z4 Z5 Z6 Z7 Z8 Z9 Y10), (0.0029361135995872667, X0 Z1 Z3 Z4 Z5 Z6 Z7 Z8 Z9 X10),␣
↪(0.0029564798130477512, Y0 Z1 Y2 Z4), (0.0029564798130477512, X0 Z1 X2 Z4), (-0.
↪0013717375866723665, Y0 Z1 Y2 Y4 Z5 Z6 Z7 Z8 Z9 Y10), (-0.0009029079399192454, X0␣
↪Z1 X2 Y4 Z5 Z6 Z7 Z8 Z9 Y10), (-0.0004688296467531211, X0 Z1 Y2 Y4 Z5 Z6 Z7 Z8 Z9␣
↪X10), (-0.0004688296467531211, Y0 Z1 X2 X4 Z5 Z6 Z7 Z8 Z9 Y10), (-0.
↪0009029079399192454, Y0 Z1 Y2 X4 Z5 Z6 Z7 Z8 Z9 X10), (-0.0013717375866723665, X0␣
↪Z1 X2 X4 Z5 Z6 Z7 Z8 Z9 X10), (0.003032718131628096, Y0 Z1 Y2 Z6), (0.
↪003032718131628096, X0 Z1 X2 Z6), (0.0030327181316280945, Y0 Z1 Y2 Z8), (0.
↪0030327181316280945, X0 Z1 X2 Z8), (-0.0009545681042087755, Y0 Z1 Y2 Z10), (-0.
```

(continues on next page)

(continued from previous page)

```
↪0009545681042087755, X0 Z1 X2 Z10), (-0.001303571630197791, Y0 Z1 Z2 Z3 Z5 Z6 Z7 Z8␣
↪Z9 Y10), (-0.001303571630197791, X0 Z1 Z2 Z3 Z5 Z6 Z7 Z8 Z9 X10), (0.
↪003797228697268315, Y0 Z1 Z2 Z3 Y4 Z6), (0.003797228697268315, X0 Z1 Z2 Z3 X4 Z6),␣
↪(0.003797228697268314, Y0 Z1 Z2 Z3 Y4 Z8), (0.003797228697268314, X0 Z1 Z2 Z3 X4␣
↪Z8), (0.0038911615108736, Y0 Z1 Z2 Z3 Y4 Z10), (0.0038911615108736, X0 Z1 Z2 Z3 X4␣
↪Z10), (-0.0015616085633067796, Y0 Z1 Z2 Z3 Z4 Z5 Z7 Z8 Z9 Y10), (-0.
↪0015616085633067796, X0 Z1 Z2 Z3 Z4 Z5 Z7 Z8 Z9 X10), (-0.00156160856330678, Y0 Z1␣
↪Z2 Z3 Z4 Z5 Z6 Z7 Z9 Y10), (-0.00156160856330678, X0 Z1 Z2 Z3 Z4 Z5 Z6 Z7 Z9 X10),␣
↪(0.05351096774236478, Z2 Z4), (0.004341885966178415, Z2 Y4 Z5 Z6 Z7 Z8 Z9 Y10), (0.
↪004341885966178415, Z2 X4 Z5 Z6 Z7 Z8 Z9 X10), (0.0626670334454797, Z2 Z6), (0.
↪0626670334454797, Z2 Z8), (0.08349262828889345, Z2 Z10), (0.004593195603389777, Y2␣
↪Z3 Z5 Z6 Z7 Z8 Z9 Y10), (0.004593195603389777, X2 Z3 Z5 Z6 Z7 Z8 Z9 X10), (0.
↪0035943814139258335, Y2 Z3 Y4 Z6), (0.0035943814139258335, X2 Z3 X4 Z6), (0.
↪0035943814139258335, Y2 Z3 Y4 Z8), (0.0035943814139258335, X2 Z3 X4 Z8), (0.
↪002787634890910415, Y2 Z3 Y4 Z10), (0.002787634890910415, X2 Z3 X4 Z10), (-0.
↪001998446536799233, Y2 Z3 Z4 Z5 Z7 Z8 Z9 Y10), (-0.001998446536799233, X2 Z3 Z4 Z5␣
↪Z7 Z8 Z9 X10), (-0.0019984465367992284, Y2 Z3 Z4 Z5 Z6 Z7 Z9 Y10), (-0.
↪0019984465367992284, X2 Z3 Z4 Z5 Z6 Z7 Z9 X10), (0.060221371942898415, Z4 Z6), (0.
↪060221371942898415, Z4 Z8), (0.053913460265503775, Z4 Z10), (-0.003837218718848247,␣
↪Y4 Z5 Z7 Z8 Z9 Y10), (-0.003837218718848247, X4 Z5 Z7 Z8 Z9 X10), (-0.
↪0038372187188482473, Y4 Z5 Z6 Z7 Z9 Y10), (-0.0038372187188482473, X4 Z5 Z6 Z7 Z9␣
↪X10), (0.06558452315458399, Z6 Z8), (0.062418263064741734, Z6 Z10), (0.
↪062418263064741714, Z8 Z10), (0.09086756126768071, Z1 Z3), (-3.508475660225329e-05,␣
↪Z1 Y3 Z4 Y5), (-3.508475660225329e-05, Z1 X3 Z4 X5), (0.005481966935653534, Z1 Y3␣
↪Z4 Z5 Z6 Z7 Z8 Z9 Z10 Y11), (0.005481966935653534, Z1 X3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 X11),
↪ (0.09360084027749466, Z1 Z5), (-0.004705162241416893, Z1 Y5 Z6 Z7 Z8 Z9 Z10 Y11),␣
↪(-0.004705162241416893, Z1 X5 Z6 Z7 Z8 Z9 Z10 X11), (0.09662264985846813, Z1 Z7),␣
↪(0.09662264985846812, Z1 Z9), (0.08857759609112856, Z1 Z11), (0.0033316360573752354,
↪ Y1 Z2 Z4 Y5), (0.0033316360573752354, X1 Z2 Z4 X5), (0.0029361135995872667, Y1 Z2␣
↪Z4 Z5 Z6 Z7 Z8 Z9 Z10 Y11), (0.0029361135995872667, X1 Z2 Z4 Z5 Z6 Z7 Z8 Z9 Z10␣
↪X11), (0.0029564798130477512, Y1 Z2 Y3 Z5), (0.0029564798130477512, X1 Z2 X3 Z5), (-
↪0.0013717375866723665, Y1 Z2 Y3 Y5 Z6 Z7 Z8 Z9 Z10 Y11), (-0.0009029079399192454,␣
↪X1 Z2 X3 Y5 Z6 Z7 Z8 Z9 Z10 Y11), (-0.0004688296467531211, X1 Z2 Y3 Y5 Z6 Z7 Z8 Z9␣
↪Z10 X11), (-0.0004688296467531211, Y1 Z2 X3 X5 Z6 Z7 Z8 Z9 Z10 Y11), (-0.
↪0009029079399192454, Y1 Z2 Y3 X5 Z6 Z7 Z8 Z9 Z10 X11), (-0.0013717375866723665, X1␣
↪Z2 X3 X5 Z6 Z7 Z8 Z9 Z10 X11), (0.003032718131628096, Y1 Z2 Y3 Z7), (0.
↪003032718131628096, X1 Z2 X3 Z7), (0.0030327181316280945, Y1 Z2 Y3 Z9), (0.
↪0030327181316280945, X1 Z2 X3 Z9), (-0.0009545681042087755, Y1 Z2 Y3 Z11), (-0.
↪0009545681042087755, X1 Z2 X3 Z11), (-0.001303571630197791, Y1 Z2 Z3 Z4 Z6 Z7 Z8 Z9␣
↪Z10 Y11), (-0.001303571630197791, X1 Z2 Z3 Z4 Z6 Z7 Z8 Z9 Z10 X11), (0.
↪003797228697268315, Y1 Z2 Z3 Z4 Y5 Z7), (0.003797228697268315, X1 Z2 Z3 Z4 X5 Z7),␣
↪(0.003797228697268314, Y1 Z2 Z3 Z4 Y5 Z9), (0.003797228697268314, X1 Z2 Z3 Z4 X5␣
↪Z9), (0.0038911615108736, Y1 Z2 Z3 Z4 Y5 Z11), (0.0038911615108736, X1 Z2 Z3 Z4 X5␣
↪Z11), (-0.0015616085633067796, Y1 Z2 Z3 Z4 Z5 Z6 Z8 Z9 Z10 Y11), (-0.
↪0015616085633067796, X1 Z2 Z3 Z4 Z5 Z6 Z8 Z9 Z10 X11), (-0.00156160856330678, Y1 Z2␣
↪Z3 Z4 Z5 Z6 Z7 Z8 Z10 Y11), (-0.00156160856330678, X1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z10 X11),
↪ (0.05351096774236478, Z3 Z5), (0.004341885966178415, Z3 Y5 Z6 Z7 Z8 Z9 Z10 Y11),␣
↪(0.004341885966178415, Z3 X5 Z6 Z7 Z8 Z9 Z10 X11), (0.0626670334454797, Z3 Z7), (0.
↪0626670334454797, Z3 Z9), (0.08349262828889345, Z3 Z11), (0.004593195603389777, Y3␣
↪Z4 Z6 Z7 Z8 Z9 Z10 Y11), (0.004593195603389777, X3 Z4 Z6 Z7 Z8 Z9 Z10 X11), (0.
↪0035943814139258335, Y3 Z4 Y5 Z7), (0.0035943814139258335, X3 Z4 X5 Z7), (0.
↪0035943814139258335, Y3 Z4 Y5 Z9), (0.0035943814139258335, X3 Z4 X5 Z9), (0.
↪002787634890910415, Y3 Z4 Y5 Z11), (0.002787634890910415, X3 Z4 X5 Z11), (-0.
↪001998446536799233, Y3 Z4 Z5 Z6 Z8 Z9 Z10 Y11), (-0.001998446536799233, X3 Z4 Z5 Z6␣
↪Z8 Z9 Z10 X11), (-0.0019984465367992284, Y3 Z4 Z5 Z6 Z7 Z8 Z10 Y11), (-0.
↪0019984465367992284, X3 Z4 Z5 Z6 Z7 Z8 Z10 X11), (0.060221371942898415, Z5 Z7), (0.
↪060221371942898415, Z5 Z9), (0.053913460265503775, Z5 Z11), (-0.003837218718848247,␣
```

(continues on next page)

(continued from previous page)

```
↪Y5 Z6 Z8 Z9 Z10 Y11), (-0.003837218718848247, X5 Z6 Z8 Z9 Z10 X11), (-0.
↪0038372187188482473, Y5 Z6 Z7 Z8 Z10 Y11), (-0.0038372187188482473, X5 Z6 Z7 Z8 Z10␣
↪X11), (0.06558452315458399, Z7 Z9), (0.062418263064741734, Z7 Z11), (0.
↪062418263064741714, Z9 Z11), (0.41455563812354274, Z0 Z1), (0.02905842727522894, Y0␣
↪Y2), (0.02905842727522894, X0 X2), (0.03443716627000962, Y0 Z2 Z3 Y4), (0.
↪03443716627000962, X0 Z2 Z3 X4), (-0.011160628514297801, Y0 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9␣
↪Y10), (-0.011160628514297801, X0 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 X10), (0.09449972206657385,
↪ Z1 Z2), (-0.0029110354547010153, Z1 Y2 Z3 Y4), (-0.0029110354547010153, Z1 X2 Z3␣
↪X4), (0.007543317569441216, Z1 Y2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Y10), (0.007543317569441216,␣
↪Z1 X2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 X10), (0.09898367691353932, Z1 Z4), (-0.
↪004354579739183622, Z1 Y4 Z5 Z6 Z7 Z8 Z9 Y10), (-0.004354579739183622, Z1 X4 Z5 Z6␣
↪Z7 Z8 Z9 X10), (0.09907741186198736, Z1 Z6), (0.09907741186198735, Z1 Z8), (0.
↪09042755838943325, Z1 Z10), (0.02905842727522894, Z0 Y1 Z2 Y3), (0.
↪02905842727522894, Z0 X1 Z2 X3), (0.0036321607988931397, Y0 X1 X2 Y3), (-0.
↪0036321607988931397, X0 X1 Y2 Y3), (-0.0036321607988931397, Y0 Y1 X2 X3), (0.
↪0036321607988931397, X0 Y1 Y2 X3), (-0.002875950698098762, Y0 X1 X3 Y4), (-0.
↪002875950698098762, X0 X1 X3 X4), (-0.002875950698098762, Y0 Y1 Y3 Y4), (-0.
↪002875950698098762, X0 Y1 Y3 X4), (0.0020613506337876825, Y0 X1 X3 Z4 Z5 Z6 Z7 Z8␣
↪Z9 Y10), (0.0020613506337876825, X0 X1 X3 Z4 Z5 Z6 Z7 Z8 Z9 X10), (0.
↪0020613506337876825, Y0 Y1 Y3 Z4 Z5 Z6 Z7 Z8 Z9 Y10), (0.0020613506337876825, X0 Y1␣
↪Y3 Z4 Z5 Z6 Z7 Z8 Z9 X10), (-0.0017826496094914316, Y1 Y3), (-0.0017826496094914316,
↪ X1 X3), (-0.0009054432974223135, Y1 X2 X3 Y4), (0.0009054432974223135, X1 X2 Y3␣
↪Y4), (0.0009054432974223135, Y1 Y2 X3 X4), (-0.0009054432974223135, X1 Y2 Y3 X4),␣
↪(0.0014073859207274033, Y1 X2 X3 Z4 Z5 Z6 Z7 Z8 Z9 Y10), (-0.0014073859207274033,␣
↪X1 X2 Y3 Z4 Z5 Z6 Z7 Z8 Z9 Y10), (-0.0014073859207274033, Y1 Y2 X3 Z4 Z5 Z6 Z7 Z8␣
↪Z9 X10), (0.0014073859207274033, X1 Y2 Y3 Z4 Z5 Z6 Z7 Z8 Z9 X10), (0.
↪002899573622979062, Y1 Z2 Y3 Z4), (0.002899573622979062, X1 Z2 X3 Z4), (-0.
↪0010478648602212561, Y1 Z2 Y3 Y4 Z5 Z6 Z7 Z8 Z9 Y10), (-0.0010478648602212561, X1␣
↪Z2 X3 Y4 Z5 Z6 Z7 Z8 Z9 Y10), (-0.0010478648602212561, Y1 Z2 Y3 X4 Z5 Z6 Z7 Z8 Z9␣
↪X10), (-0.0010478648602212561, X1 Z2 X3 X4 Z5 Z6 Z7 Z8 Z9 X10), (0.
↪0011411758810398506, Y1 Z2 Y3 Z6), (0.0011411758810398506, X1 Z2 X3 Z6), (0.
↪0011411758810398493, Y1 Z2 Y3 Z8), (0.0011411758810398493, X1 Z2 X3 Z8), (-0.
↪0010475030435323413, Y1 Z2 Y3 Z10), (-0.0010475030435323413, X1 Z2 X3 Z10), (0.
↪03443716627000962, Z0 Y1 Z2 Z3 Z4 Y5), (0.03443716627000962, Z0 X1 Z2 Z3 Z4 X5), (0.
↪002875950698098762, Y0 X1 X2 Z3 Z4 Y5), (-0.002875950698098762, X0 X1 Y2 Z3 Z4 Y5),␣
↪(-0.002875950698098762, Y0 Y1 X2 Z3 Z4 X5), (0.002875950698098762, X0 Y1 Y2 Z3 Z4␣
↪X5), (0.0053828366360446685, Y0 X1 X4 Y5), (-0.0053828366360446685, X0 X1 Y4 Y5), (-
↪0.0053828366360446685, Y0 Y1 X4 X5), (0.0053828366360446685, X0 Y1 Y4 X5), (0.
↪00035058250223327027, Y0 X1 X5 Z6 Z7 Z8 Z9 Y10), (0.00035058250223327027, X0 X1 X5␣
↪Z6 Z7 Z8 Z9 X10), (0.00035058250223327027, Y0 Y1 Y5 Z6 Z7 Z8 Z9 Y10), (0.
↪00035058250223327027, X0 Y1 Y5 Z6 Z7 Z8 Z9 X10), (0.0042370793547975485, Y1 Z3 Z4␣
↪Y5), (0.0042370793547975485, X1 Z3 Z4 X5), (-5.6906190068689256e-05, Y1 X2 X4 Y5),␣
↪(-5.6906190068689256e-05, X1 X2 X4 X5), (-5.6906190068689256e-05, Y1 Y2 Y4 Y5), (-5.
↪6906190068689256e-05, X1 Y2 Y4 X5), (-0.00014495692030201056, Y1 X2 X5 Z6 Z7 Z8 Z9␣
↪Y10), (0.00014495692030201056, X1 X2 Y5 Z6 Z7 Z8 Z9 Y10), (0.00014495692030201056,␣
↪Y1 Y2 X5 Z6 Z7 Z8 Z9 X10), (-0.00014495692030201056, X1 Y2 Y5 Z6 Z7 Z8 Z9 X10), (-0.
↪0004952251157452342, Y1 Z2 Z3 Y5), (-0.0004952251157452342, X1 Z2 Z3 X5), (0.
↪0011230162982700607, Y1 Z2 Z3 X4 X5 Z6 Z7 Z8 Z9 Y10), (-0.0011230162982700607, X1␣
↪Z2 Z3 X4 Y5 Z6 Z7 Z8 Z9 Y10), (-0.0011230162982700607, Y1 Z2 Z3 Y4 X5 Z6 Z7 Z8 Z9␣
↪X10), (0.0011230162982700607, X1 Z2 Z3 Y4 Y5 Z6 Z7 Z8 Z9 X10), (0.
↪0012360588162335989, Y1 Z2 Z3 Z4 Y5 Z6), (0.0012360588162335989, X1 Z2 Z3 Z4 X5 Z6),
↪ (0.0012360588162335976, Y1 Z2 Z3 Z4 Y5 Z8), (0.0012360588162335976, X1 Z2 Z3 Z4 X5␣
↪Z8), (0.0028405394777676262, Y1 Z2 Z3 Z4 Y5 Z10), (0.0028405394777676262, X1 Z2 Z3␣
↪Z4 X5 Z10), (0.002454762003519232, Y0 X1 X6 Y7), (-0.002454762003519232, X0 X1 Y6␣
↪Y7), (-0.002454762003519232, Y0 Y1 X6 X7), (0.002454762003519232, X0 Y1 Y6 X7), (-0.
↪001891542250588245, Y1 X2 X6 Y7), (-0.001891542250588245, X1 X2 X6 X7), (-0.
↪001891542250588245, Y1 Y2 Y6 Y7), (-0.001891542250588245, X1 Y2 Y6 X7), (-0.
```

(continues on next page)

(continued from previous page)

```
↪00256116988103471 6, Y1 Z2 Z3 X4 X6 Y7), (-0.00256116988103471 6, X1 Z2 Z3 X4 X6 X7),␣
↪(-0.00256116988103471 6, Y1 Z2 Z3 Y4 Y6 Y7), (-0.00256116988103471 6, X1 Z2 Z3 Y4 Y6␣
↪X7), (-0.001502506566785411, Y1 Z2 Z3 Z4 Z5 X6 X7 Z8 Z9 Y10), (0.001502506566785411,
↪ X1 Z2 Z3 Z4 Z5 X6 Y7 Z8 Z9 Y10), (0.001502506566785411, Y1 Z2 Z3 Z4 Z5 Y6 X7 Z8 Z9␣
↪X10), (-0.001502506566785411, X1 Z2 Z3 Z4 Z5 Y6 Y7 Z8 Z9 X10), (0.
↪0024547620035192 32, Y0 X1 X8 Y9), (-0.0024547620035192 32, X0 X1 Y8 Y9), (-0.
↪0024547620035192 32, Y0 Y1 X8 X9), (0.0024547620035192 32, X0 Y1 Y8 X9), (-0.
↪0018915422505882 45, Y1 X2 X8 Y9), (-0.0018915422505882 45, X1 X2 X8 X9), (-0.
↪0018915422505882 45, Y1 Y2 Y8 Y9), (-0.0018915422505882 45, X1 Y2 Y8 X9), (-0.
↪00256116988103471 6, Y1 Z2 Z3 X4 X8 Y9), (-0.00256116988103471 6, X1 Z2 Z3 X4 X8 X9),␣
↪(-0.00256116988103471 6, Y1 Z2 Z3 Y4 Y8 Y9), (-0.00256116988103471 6, X1 Z2 Z3 Y4 Y8␣
↪X9), (-0.001502506566785411, Y1 Z2 Z3 Z4 Z5 Z6 Z7 X8 X9 Y10), (0.001502506566785411,
↪ X1 Z2 Z3 Z4 Z5 Z6 Z7 X8 Y9 Y10), (0.001502506566785411, Y1 Z2 Z3 Z4 Z5 Z6 Z7 Y8 X9␣
↪X10), (-0.001502506566785411, X1 Z2 Z3 Z4 Z5 Z6 Z7 Y8 Y9 X10), (-0.
↪0111606285142978 01, Z0 Y1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Y11), (-0.0111606285142978 01,␣
↪Z0 X1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 X11), (-0.0020613506337876825, Y0 X1 X2 Z3 Z4 Z5␣
↪Z6 Z7 Z8 Z9 Z10 Y11), (0.0020613506337876825, X0 X1 Y2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10␣
↪Y11), (0.0020613506337876825, Y0 Y1 X2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 X11), (-0.
↪0020613506337876825, X0 Y1 Y2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 X11), (-0.
↪0003505825022332 7027, Y0 X1 X4 Z5 Z6 Z7 Z8 Z9 Z10 Y11), (0.0003505825022332 7027, X0␣
↪X1 Y4 Z5 Z6 Z7 Z8 Z9 Z10 Y11), (0.0003505825022332 7027, Y0 Y1 X4 Z5 Z6 Z7 Z8 Z9 Z10␣
↪X11), (-0.0003505825022332 7027, X0 Y1 Y4 Z5 Z6 Z7 Z8 Z9 Z10 X11), (0.
↪0018499622983046825, Y0 X1 X10 Y11), (-0.0018499622983046825, X0 X1 Y10 Y11), (-0.
↪0018499622983046825, Y0 Y1 X10 X11), (0.0018499622983046825, X0 Y1 Y10 X11), (0.
↪0015287276788598636, Y1 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Y11), (0.0015287276788598636, X1␣
↪Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 X11), (-0.00032387272645111053, Y1 X2 X4 Z5 Z6 Z7 Z8 Z9␣
↪Z10 Y11), (-0.00032387272645111053, X1 X2 X4 Z5 Z6 Z7 Z8 Z9 Z10 X11), (-0.
↪00032387272645111053, Y1 Y2 Y4 Z5 Z6 Z7 Z8 Z9 Z10 Y11), (-0.00032387272645111053,␣
↪X1 Y2 Y4 Z5 Z6 Z7 Z8 Z9 Z10 X11), (-9.293493932356588e-05, Y1 X2 X10 Y11), (-9.
↪293493932356588e-05, X1 X2 X10 X11), (-9.293493932356588e-05, Y1 Y2 Y10 Y11), (-9.
↪293493932356588e-05, X1 Y2 Y10 X11), (-0.0024265879284678517, Y1 Z2 Z3 Z5 Z6 Z7 Z8␣
↪Z9 Z10 Y11), (-0.0024265879284678517, X1 Z2 Z3 Z5 Z6 Z7 Z8 Z9 Z10 X11), (-0.
↪0010506220331059734, Y1 Z2 Z3 X4 X10 Y11), (-0.0010506220331059734, X1 Z2 Z3 X4 X10␣
↪X11), (-0.0010506220331059734, Y1 Z2 Z3 Y4 Y10 Y11), (-0.0010506220331059734, X1 Z2␣
↪Z3 Y4 Y10 X11), (-5.9101996521368776e-05, Y1 Z2 Z3 Z4 Z5 Z7 Z8 Z9 Z10 Y11), (-5.
↪9101996521368776e-05, X1 Z2 Z3 Z4 Z5 Z7 Z8 Z9 Z10 X11), (-5.9101996521369155e-05,␣
↪Y1 Z2 Z3 Z4 Z5 Z6 Z7 Z9 Z10 Y11), (-5.9101996521369155e-05, X1 Z2 Z3 Z4 Z5 Z6 Z7 Z9␣
↪Z10 X11), (0.0005606190834798984, Y1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Y11), (0.
↪0005606190834798984, X1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 X11), (0.09449972206657385, Z0 Z3),␣
↪(-0.0017826496094914316, Y0 Z1 Y2 Z3), (-0.0017826496094914316, X0 Z1 X2 Z3), (0.
↪0042370793547975485, Y0 Z1 Z2 Y4), (0.0042370793547975485, X0 Z1 Z2 X4), (0.
↪0015287276788598636, Y0 Z1 Z2 Z4 Z5 Z6 Z7 Z8 Z9 Y10), (0.0015287276788598636, X0 Z1␣
↪Z2 Z4 Z5 Z6 Z7 Z8 Z9 X10), (0.12337832084798564, Z2 Z3), (0.011777702347089292, Y2␣
↪Y4), (0.011777702347089292, X2 X4), (-0.03290192982028284, Y2 Z4 Z5 Z6 Z7 Z8 Z9␣
↪Y10), (-0.03290192982028284, X2 Z4 Z5 Z6 Z7 Z8 Z9 X10), (0.05656920126071201, Z3␣
↪Z4), (0.012743636944600203, Z3 Y4 Z5 Z6 Z7 Z8 Z9 Y10), (0.012743636944600203, Z3 X4␣
↪Z5 Z6 Z7 Z8 Z9 X10), (0.06864790934686854, Z3 Z6), (0.06864790934686854, Z3 Z8), (0.
↪11425464668554039, Z3 Z10), (-0.0029110354547010153, Z0 Y3 Z4 Y5), (-0.
↪0029110354547010153, Z0 X3 Z4 X5), (-0.0009054432974223135, Y0 Z1 Y2 Y3 Z4 Y5), (-0.
↪0009054432974223135, X0 Z1 X2 Y3 Z4 Y5), (-0.0009054432974223135, Y0 Z1 Y2 X3 Z4␣
↪X5), (-0.0009054432974223135, X0 Z1 X2 X3 Z4 X5), (5.690619006868926e-05, Y0 Z1 Z2␣
↪X3 X4 Y5), (-5.690619006868926e-05, X0 Z1 Z2 X3 Y4 Y5), (-5.690619006868926e-05, Y0␣
↪Z1 Z2 Y3 X4 X5), (5.690619006868926e-05, X0 Z1 Z2 Y3 Y4 X5), (-0.
↪00032387272645111053, Y0 Z1 Z2 X3 X5 Z6 Z7 Z8 Z9 Y10), (-0.00032387272645111053, X0␣
↪Z1 Z2 X3 X5 Z6 Z7 Z8 Z9 X10), (-0.00032387272645111053, Y0 Z1 Z2 Y3 Y5 Z6 Z7 Z8 Z9␣
↪Y10), (-0.00032387272645111053, X0 Z1 Z2 Y3 Y5 Z6 Z7 Z8 Z9 X10), (0.
↪011777702347089292, Z2 Y3 Z4 Y5), (0.011777702347089292, Z2 X3 Z4 X5), (0.
```

(continues on next page)

(continued from previous page)

```
↪0030582335183472237, Y2 X3 X4 Y5), (-0.0030582335183472237, X2 X3 Y4 Y5), (-0.
↪0030582335183472237, Y2 Y3 X4 X5), (0.0030582335183472237, X2 Y3 Y4 X5), (0.
↪008401750978421788, Y2 X3 X5 Z6 Z7 Z8 Z9 Y10), (0.008401750978421788, X2 X3 X5 Z6␣
↪Z7 Z8 Z9 X10), (0.008401750978421788, Y2 Y3 Y5 Z6 Z7 Z8 Z9 Y10), (0.
↪008401750978421788, X2 Y3 Y5 Z6 Z7 Z8 Z9 X10), (-0.001577111222680453, Y3 Y5), (-0.
↪001577111222680453, X3 X5), (0.0021362024316156046, Y3 X4 X5 Z6 Z7 Z8 Z9 Y10), (-0.
↪0021362024316156046, X3 X4 Y5 Z6 Z7 Z8 Z9 Y10), (-0.0021362024316156046, Y3 Y4 X5␣
↪Z6 Z7 Z8 Z9 X10), (0.0021362024316156046, X3 Y4 Y5 Z6 Z7 Z8 Z9 X10), (-0.
↪0012095002061739725, Y3 Z4 Y5 Z6), (-0.0012095002061739725, X3 Z4 X5 Z6), (-0.
↪0012095002061739725, Y3 Z4 Y5 Z8), (-0.0012095002061739725, X3 Z4 X5 Z8), (0.
↪010573106696630573, Y3 Z4 Y5 Z10), (0.010573106696630573, X3 Z4 X5 Z10), (0.
↪001891542250588245, Y0 Z1 Z2 X3 X6 Y7), (-0.001891542250588245, X0 Z1 Z2 X3 Y6 Y7),␣
↪(-0.001891542250588245, Y0 Z1 Z2 Y3 X6 X7), (0.001891542250588245, X0 Z1 Z2 Y3 Y6␣
↪X7), (0.005980875901388835, Y2 X3 X6 Y7), (-0.005980875901388835, X2 X3 Y6 Y7), (-0.
↪005980875901388835, Y2 Y3 X6 X7), (0.005980875901388835, X2 Y3 Y6 X7), (-0.
↪004803881620099806, Y3 X4 X6 Y7), (-0.004803881620099806, X3 X4 X6 X7), (-0.
↪004803881620099806, Y3 Y4 Y6 Y7), (-0.004803881620099806, X3 Y4 Y6 X7), (-0.
↪004882988854490184, Y3 Z4 Z5 X6 X7 Z8 Z9 Y10), (0.004882988854490184, X3 Z4 Z5 X6␣
↪Y7 Z8 Z9 Y10), (0.004882988854490184, Y3 Z4 Z5 Y6 X7 Z8 Z9 X10), (-0.
↪004882988854490184, X3 Z4 Z5 Y6 Y7 Z8 Z9 X10), (0.001891542250588245, Y0 Z1 Z2 X3␣
↪X8 Y9), (-0.001891542250588245, X0 Z1 Z2 X3 Y8 Y9), (-0.001891542250588245, Y0 Z1␣
↪Z2 Y3 X8 X9), (0.001891542250588245, X0 Z1 Z2 Y3 Y8 X9), (0.005980875901388835, Y2␣
↪X3 X8 Y9), (-0.005980875901388835, X2 X3 Y8 Y9), (-0.005980875901388835, Y2 Y3 X8␣
↪X9), (0.005980875901388835, X2 Y3 Y8 X9), (-0.004803881620099806, Y3 X4 X8 Y9), (-0.
↪004803881620099806, X3 X4 X8 X9), (-0.004803881620099806, Y3 Y4 Y8 Y9), (-0.
↪004803881620099806, X3 Y4 Y8 X9), (-0.004882988854490183, Y3 Z4 Z5 Z6 Z7 X8 X9 Y10),
↪ (0.004882988854490183, X3 Z4 Z5 Z6 Z7 X8 Y9 Y10), (0.004882988854490183, Y3 Z4 Z5␣
↪Z6 Z7 Y8 X9 X10), (-0.004882988854490183, X3 Z4 Z5 Z6 Z7 Y8 Y9 X10), (0.
↪007543317569441216, Z0 Y3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Y11), (0.007543317569441216, Z0 X3␣
↪Z4 Z5 Z6 Z7 Z8 Z9 Z10 X11), (0.001407385920727403, Y0 Z1 Y2 Y3 Z4 Z5 Z6 Z7 Z8 Z9␣
↪Z10 Y11), (0.001407385920727403, X0 Z1 X2 Y3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Y11), (0.
↪001407385920727403, Y0 Z1 Y2 X3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 X11), (0.001407385920727403,␣
↪X0 Z1 X2 X3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 X11), (-0.00014495692030201056, Y0 Z1 Z2 X3 X4 Z5␣
↪Z6 Z7 Z8 Z9 Z10 Y11), (0.00014495692030201056, X0 Z1 Z2 X3 Y4 Z5 Z6 Z7 Z8 Z9 Z10␣
↪Y11), (0.00014495692030201056, Y0 Z1 Z2 Y3 X4 Z5 Z6 Z7 Z8 Z9 Z10 X11), (-0.
↪00014495692030201056, X0 Z1 Z2 Y3 Y4 Z5 Z6 Z7 Z8 Z9 Z10 X11), (9.293493932356588e-
↪05, Y0 Z1 Z2 X3 X10 Y11), (-9.293493932356588e-05, X0 Z1 Z2 X3 Y10 Y11), (-9.
↪293493932356588e-05, Y0 Z1 Z2 Y3 X10 X11), (9.293493932356588e-05, X0 Z1 Z2 Y3 Y10␣
↪X11), (-0.03290192982028284, Z2 Y3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Y11), (-0.
↪03290192982028284, Z2 X3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 X11), (-0.008401750978421788, Y2 X3␣
↪X4 Z5 Z6 Z7 Z8 Z9 Z10 Y11), (0.008401750978421788, X2 X3 Y4 Z5 Z6 Z7 Z8 Z9 Z10 Y11),
↪ (0.008401750978421788, Y2 Y3 X4 Z5 Z6 Z7 Z8 Z9 Z10 X11), (-0.008401750978421788,␣
↪X2 Y3 Y4 Z5 Z6 Z7 Z8 Z9 Z10 X11), (0.03076201839664694, Y2 X3 X10 Y11), (-0.
↪03076201839664694, X2 X3 Y10 Y11), (-0.03076201839664694, Y2 Y3 X10 X11), (0.
↪03076201839664694, X2 Y3 Y10 X11), (0.0024569931717741715, Y3 Z5 Z6 Z7 Z8 Z9 Z10␣
↪Y11), (0.0024569931717741715, X3 Z5 Z6 Z7 Z8 Z9 Z10 X11), (0.007785471805720158, Y3␣
↪X4 X10 Y11), (0.007785471805720158, X3 X4 X10 X11), (0.007785471805720158, Y3 Y4␣
↪Y10 Y11), (0.007785471805720158, X3 Y4 Y10 X11), (0.0028845423176909507, Y3 Z4 Z5␣
↪Z7 Z8 Z9 Z10 Y11), (0.0028845423176909507, X3 Z4 Z5 Z7 Z8 Z9 Z10 X11), (0.
↪0028845423176909555, Y3 Z4 Z5 Z6 Z7 Z9 Z10 Y11), (0.0028845423176909555, X3 Z4 Z5␣
↪Z6 Z7 Z9 Z10 X11), (-0.03495095588690905, Y3 Z4 Z5 Z6 Z7 Z8 Z9 Y11), (-0.
↪03495095588690905, X3 Z4 Z5 Z6 Z7 Z8 Z9 X11), (0.09898367691353932, Z0 Z5), (0.
↪002899573622979062, Y0 Z1 Y2 Z5), (0.002899573622979062, X0 Z1 X2 Z5), (-0.
↪0004952251157452342, Y0 Z1 Z2 Z3 Y4 Z5), (-0.0004952251157452342, X0 Z1 Z2 Z3 X4␣
↪Z5), (-0.0024265879284678517, Y0 Z1 Z2 Z3 Z4 Z6 Z7 Z8 Z9 Y10), (-0.
↪0024265879284678517, X0 Z1 Z2 Z3 Z4 Z6 Z7 Z8 Z9 X10), (0.05656920126071201, Z2 Z5),␣
↪(-0.001577111222680453, Y2 Z3 Y4 Z5), (-0.001577111222680453, X2 Z3 X4 Z5), (0.
```

(continues on next page)

(continued from previous page)

```
↪0024569931717741715, Y2 Z3 Z4 Z6 Z7 Z8 Z9 Y10), (0.0024569931717741715, X2 Z3 Z4 Z6␣
↪Z7 Z8 Z9 X10), (0.08468141134340954, Z4 Z5), (-0.009009636937375524, Y4 Z6 Z7 Z8 Z9␣
↪Y10), (-0.009009636937375524, X4 Z6 Z7 Z8 Z9 X10), (0.07054866291332548, Z5 Z6), (0.
↪07054866291332548, Z5 Z8), (0.06049115034651019, Z5 Z10), (0.002561169881034716, Y0␣
↪Z1 Z2 Z3 Z4 X5 X6 Y7), (-0.002561169881034716, X0 Z1 Z2 Z3 Z4 X5 Y6 Y7), (-0.
↪002561169881034716, Y0 Z1 Z2 Z3 Z4 Y5 X6 X7), (0.002561169881034716, X0 Z1 Z2 Z3 Z4␣
↪Y5 Y6 X7), (0.004803881620099806, Y2 Z3 Z4 X5 X6 Y7), (-0.004803881620099806, X2 Z3␣
↪Z4 X5 Y6 Y7), (-0.004803881620099806, Y2 Z3 Z4 Y5 X6 X7), (0.004803881620099806, X2␣
↪Z3 Z4 Y5 Y6 X7), (0.010327290970427058, Y4 X5 X6 Y7), (-0.010327290970427058, X4 X5␣
↪Y6 Y7), (-0.010327290970427058, Y4 Y5 X6 X7), (0.010327290970427058, X4 Y5 Y6 X7),␣
↪(-0.0034635998144360207, Y5 X6 X7 Z8 Z9 Y10), (0.0034635998144360207, X5 X6 Y7 Z8␣
↪Z9 Y10), (0.0034635998144360207, Y5 Y6 X7 Z8 Z9 X10), (-0.0034635998144360207, X5␣
↪Y6 Y7 Z8 Z9 X10), (0.002561169881034716, Y0 Z1 Z2 Z3 Z4 X5 X8 Y9), (-0.
↪002561169881034716, X0 Z1 Z2 Z3 Z4 X5 Y8 Y9), (-0.002561169881034716, Y0 Z1 Z2 Z3␣
↪Z4 Y5 X8 X9), (0.002561169881034716, X0 Z1 Z2 Z3 Z4 Y5 Y8 X9), (0.
↪004803881620099806, Y2 Z3 Z4 X5 X8 Y9), (-0.004803881620099806, X2 Z3 Z4 X5 Y8 Y9),␣
↪(-0.004803881620099806, Y2 Z3 Z4 Y5 X8 X9), (0.004803881620099806, X2 Z3 Z4 Y5 Y8␣
↪X9), (0.010327290970427058, Y4 X5 X8 Y9), (-0.010327290970427058, X4 X5 Y8 Y9), (-0.
↪010327290970427058, Y4 Y5 X8 X9), (0.010327290970427058, X4 Y5 Y8 X9), (-0.
↪0034635998144360207, Y5 Z6 Z7 X8 X9 Y10), (0.0034635998144360207, X5 Z6 Z7 X8 Y9␣
↪Y10), (0.0034635998144360207, Y5 Z6 Z7 Y8 X9 X10), (-0.0034635998144360207, X5 Z6␣
↪Z7 Y8 Y9 X10), (-0.004354579739183622, Z0 Y5 Z6 Z7 Z8 Z9 Z10 Y11), (-0.
↪004354579739183622, Z0 X5 Z6 Z7 Z8 Z9 Z10 X11), (-0.0010478648602212561, Y0 Z1 Y2␣
↪Y5 Z6 Z7 Z8 Z9 Z10 Y11), (-0.0010478648602212561, X0 Z1 X2 Y5 Z6 Z7 Z8 Z9 Z10 Y11),␣
↪(-0.0010478648602212561, Y0 Z1 Y2 X5 Z6 Z7 Z8 Z9 Z10 X11), (-0.0010478648602212561,␣
↪X0 Z1 X2 X5 Z6 Z7 Z8 Z9 Z10 X11), (0.0011230162982700607, Y0 Z1 Z2 Z3 Y4 Y5 Z6 Z7␣
↪Z8 Z9 Z10 Y11), (0.0011230162982700607, X0 Z1 Z2 Z3 X4 Y5 Z6 Z7 Z8 Z9 Z10 Y11), (0.
↪0011230162982700607, Y0 Z1 Z2 Z3 Y4 X5 Z6 Z7 Z8 Z9 Z10 X11), (0.0011230162982700607,
↪ X0 Z1 Z2 Z3 X4 X5 Z6 Z7 Z8 Z9 Z10 X11), (0.0010506220331059734, Y0 Z1 Z2 Z3 Z4 X5␣
↪X10 Y11), (-0.0010506220331059734, X0 Z1 Z2 Z3 Z4 X5 Y10 Y11), (-0.
↪0010506220331059734, Y0 Z1 Z2 Z3 Z4 Y5 X10 X11), (0.0010506220331059734, X0 Z1 Z2␣
↪Z3 Z4 Y5 Y10 X11), (0.012743636944600203, Z2 Y5 Z6 Z7 Z8 Z9 Z10 Y11), (0.
↪012743636944600203, Z2 X5 Z6 Z7 Z8 Z9 Z10 X11), (0.002136202431615605, Y2 Z3 Y4 Y5␣
↪Z6 Z7 Z8 Z9 Z10 Y11), (0.002136202431615605, X2 Z3 X4 Y5 Z6 Z7 Z8 Z9 Z10 Y11), (0.
↪002136202431615605, Y2 Z3 Y4 X5 Z6 Z7 Z8 Z9 Z10 X11), (0.002136202431615605, X2 Z3␣
↪X4 X5 Z6 Z7 Z8 Z9 Z10 X11), (-0.007785471805720157, Y2 Z3 Z4 X5 X10 Y11), (0.
↪007785471805720157, X2 Z3 Z4 X5 Y10 Y11), (0.007785471805720157, Y2 Z3 Z4 Y5 X10␣
↪X11), (-0.007785471805720157, X2 Z3 Z4 Y5 Y10 X11), (-0.009009636937375524, Z4 Y5␣
↪Z6 Z7 Z8 Z9 Z10 Y11), (-0.009009636937375524, Z4 X5 Z6 Z7 Z8 Z9 Z10 X11), (0.
↪006577690081006409, Y4 X5 X10 Y11), (-0.006577690081006409, X4 X5 Y10 Y11), (-0.
↪006577690081006409, Y4 Y5 X10 X11), (0.006577690081006409, X4 Y5 Y10 X11), (-0.
↪00037361890441222636, Y5 Z7 Z8 Z9 Z10 Y11), (-0.00037361890441222636, X5 Z7 Z8 Z9␣
↪Z10 X11), (-0.00037361890441222269, Y5 Z6 Z7 Z9 Z10 Y11), (-0.00037361890441222269,␣
↪X5 Z6 Z7 Z9 Z10 X11), (0.010904449891730138, Y5 Z6 Z7 Z8 Z9 Y11), (0.
↪010904449891730138, X5 Z6 Z7 Z8 Z9 X11), (0.09907741186198736, Z0 Z7), (0.
↪0011411758810398506, Y0 Z1 Y2 Z7), (0.0011411758810398506, X0 Z1 X2 Z7), (0.
↪0012360588162335989, Y0 Z1 Z2 Z3 Y4 Z7), (0.0012360588162335989, X0 Z1 Z2 Z3 X4 Z7),
↪ (-5.9101996521368776e-05, Y0 Z1 Z2 Z3 Z4 Z5 Z6 Z8 Z9 Y10), (-5.9101996521368776e-
↪05, X0 Z1 Z2 Z3 Z4 Z5 Z6 Z8 Z9 X10), (0.06864790934686854, Z2 Z7), (-0.
↪0012095002061739725, Y2 Z3 Y4 Z7), (-0.0012095002061739725, X2 Z3 X4 Z7), (0.
↪0028845423176909507, Y2 Z3 Z4 Z5 Z6 Z8 Z9 Y10), (0.0028845423176909507, X2 Z3 Z4 Z5␣
↪Z6 Z8 Z9 X10), (0.07054866291332548, Z4 Z7), (-0.00037361890441222636, Y4 Z5 Z6 Z8␣
↪Z9 Y10), (-0.00037361890441222636, X4 Z5 Z6 Z8 Z9 X10), (0.07823637778985214, Z6␣
↪Z7), (0.0698018080330067, Z7 Z8), (0.0672954012940464, Z7 Z10), (0.
↪004217284878422704, Y6 X7 X8 Y9), (-0.004217284878422704, X6 X7 Y8 Y9), (-0.
↪004217284878422704, Y6 Y7 X8 X9), (0.004217284878422704, X6 Y7 Y8 X9), (-0.
↪001502506566785411, Y0 Z1 Z2 Z3 Z4 Z5 Y6 Y7 Z8 Z9 Z10 Y11), (-0.001502506566785411,␣
```

(continues on next page)

(continued from previous page)

```
↪X0 Z1 Z2 Z3 Z4 Z5 X6 Y7 Z8 Z9 Z10 Y11), (-0.001502506566785411, Y0 Z1 Z2 Z3 Z4 Z5␣
↪Y6 X7 Z8 Z9 Z10 X11), (-0.001502506566785411, X0 Z1 Z2 Z3 Z4 Z5 X6 X7 Z8 Z9 Z10␣
↪X11), (-0.004882988854490184, Y2 Z3 Z4 Z5 Y6 Y7 Z8 Z9 Z10 Y11), (-0.
↪004882988854490184, X2 Z3 Z4 Z5 X6 Y7 Z8 Z9 Z10 Y11), (-0.004882988854490184, Y2 Z3␣
↪Z4 Z5 Y6 X7 Z8 Z9 Z10 X11), (-0.004882988854490184, X2 Z3 Z4 Z5 X6 X7 Z8 Z9 Z10␣
↪X11), (-0.0034635998144360207, Y4 Z5 Y6 Y7 Z8 Z9 Z10 Y11), (-0.0034635998144360207,␣
↪X4 Z5 X6 Y7 Z8 Z9 Z10 Y11), (-0.0034635998144360207, Y4 Z5 Y6 X7 Z8 Z9 Z10 X11), (-
↪0.0034635998144360207, X4 Z5 X6 X7 Z8 Z9 Z10 X11), (0.004877138229304677, Y6 X7 X10␣
↪Y11), (-0.004877138229304677, X6 X7 Y10 Y11), (-0.004877138229304677, Y6 Y7 X10␣
↪X11), (0.004877138229304677, X6 Y7 Y10 X11), (0.09907741186198735, Z0 Z9), (0.
↪0011411758810398493, Y0 Z1 Y2 Z9), (0.0011411758810398493, X0 Z1 X2 Z9), (0.
↪0012360588162335976, Y0 Z1 Z2 Z3 Y4 Z9), (0.0012360588162335976, X0 Z1 Z2 Z3 X4 Z9),
↪ (-5.9101996521369155e-05, Y0 Z1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Y10), (-5.9101996521369155e-
↪05, X0 Z1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 X10), (0.06864790934686854, Z2 Z9), (-0.
↪0012095002061739725, Y2 Z3 Y4 Z9), (-0.0012095002061739725, X2 Z3 X4 Z9), (0.
↪0028845423176909555, Y2 Z3 Z4 Z5 Z6 Z7 Z8 Y10), (0.0028845423176909555, X2 Z3 Z4 Z5␣
↪Z6 Z7 Z8 X10), (0.07054866291332548, Z4 Z9), (-0.0003736189044122269, Y4 Z5 Z6 Z7␣
↪Z8 Y10), (-0.0003736189044122269, X4 Z5 Z6 Z7 Z8 X10), (0.0698018080330067, Z6 Z9),␣
↪(0.07823637778985214, Z8 Z9), (0.06729540129404639, Z9 Z10), (-0.
↪0015025065667854107, Y0 Z1 Z2 Z3 Z4 Z5 Z6 Z7 Y8 Y9 Z10 Y11), (-0.
↪0015025065667854107, X0 Z1 Z2 Z3 Z4 Z5 Z6 Z7 X8 Y9 Z10 Y11), (-0.
↪0015025065667854107, Y0 Z1 Z2 Z3 Z4 Z5 Z6 Z7 Y8 X9 Z10 X11), (-0.
↪0015025065667854107, X0 Z1 Z2 Z3 Z4 Z5 Z6 Z7 X8 X9 Z10 X11), (-0.004882988854490183,
↪ Y2 Z3 Z4 Z5 Z6 Z7 Y8 Y9 Z10 Y11), (-0.004882988854490183, X2 Z3 Z4 Z5 Z6 Z7 X8 Y9␣
↪Z10 Y11), (-0.004882988854490183, Y2 Z3 Z4 Z5 Z6 Z7 Y8 X9 Z10 X11), (-0.
↪004882988854490183, X2 Z3 Z4 Z5 Z6 Z7 X8 X9 Z10 X11), (-0.0034635998144360207, Y4␣
↪Z5 Z6 Z7 Y8 Y9 Z10 Y11), (-0.0034635998144360207, X4 Z5 Z6 Z7 X8 Y9 Z10 Y11), (-0.
↪0034635998144360207, Y4 Z5 Z6 Z7 Y8 X9 Z10 X11), (-0.0034635998144360207, X4 Z5 Z6␣
↪Z7 X8 X9 Z10 X11), (0.004877138229304677, Y8 X9 X10 Y11), (-0.004877138229304677,␣
↪X8 X9 Y10 Y11), (-0.004877138229304677, Y8 Y9 X10 X11), (0.004877138229304677, X8␣
↪Y9 Y10 X11), (0.09042755838943325, Z0 Z11), (-0.0010475030435323413, Y0 Z1 Y2 Z11),␣
↪(-0.0010475030435323413, X0 Z1 X2 Z11), (0.0028405394777676262, Y0 Z1 Z2 Z3 Y4 Z11),
↪ (0.0028405394777676262, X0 Z1 Z2 Z3 X4 Z11), (0.0005606190834798984, Y0 Z1 Z2 Z3␣
↪Z4 Z5 Z6 Z7 Z8 Z9 Y10 Z11), (0.0005606190834798984, X0 Z1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9␣
↪X10 Z11), (0.11425464668554039, Z2 Z11), (0.010573106696630573, Y2 Z3 Y4 Z11), (0.
↪010573106696630573, X2 Z3 X4 Z11), (-0.03495095588690905, Y2 Z3 Z4 Z5 Z6 Z7 Z8 Z9␣
↪Y10 Z11), (-0.03495095588690905, X2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 X10 Z11), (0.
↪06049115034651019, Z4 Z11), (0.010904449891730138, Y4 Z5 Z6 Z7 Z8 Z9 Y10 Z11), (0.
↪010904449891730138, X4 Z5 Z6 Z7 Z8 Z9 X10 Z11), (0.0672954012940464, Z6 Z11), (0.
↪06729540129404639, Z8 Z11), (0.11404376960716622, Z10 Z11)
```

Unitary transformations may be applied to the integrals with the *rotate()* method, which takes a unitary matrix and transforms in-place:

```
from scipy.stats import ortho_group
u = ortho_group.rvs(lih_sto3g.n_orbital)  # Random, real unitary matrix
integral_operator.rotate(u)
print(integral_operator.df())
```

```
      Coefficients                Terms
0         1.050650
1        -1.520568              F0^ F0
2         0.364231              F0^ F2
3        -0.543298              F0^ F4
4         0.351219              F0^ F6
...            ...                 ...
```

(continues on next page)

(continued from previous page)

```
4460      0.008956   F11^ F10^ F9  F10
4461      0.008956   F11^ F10^ F9  F10
4462      0.164174  F11^ F10^ F10  F11
4463      0.164174  F11^ F10^ F10  F11
4464     -1.087628            F11^ F11

[4465 rows x 2 columns]
```

Integral operators support other utility methods including *approx_equal()*, for comparing Hamiltonians, *items()*, which iterates over terms, and *to_FermionOperator()*, for converting between operator and integral focused objects.

Alongside integral operators, the *operators* module also provides a set of classes for managing reduced density matrices (RDMs). These may be used in combination with integral operators to compute useful properties of the Hamiltonian. For example, with the *UnrestrictedOneBodyRDM* we may compute the total mean-field energy and effective potential matrices:

```
from inquanto.operators import UnrestrictedOneBodyRDM
import numpy as np

integral_operator = load_h5('h3_sto3g_m2_u.h5', as_tuple = True).hamiltonian_operator
rdm1 = UnrestrictedOneBodyRDM(rdm1_aa=np.diag([1, 1, 0]), rdm1_bb=np.diag([1, 0, 0]))

print("Mean-field energy:\n", integral_operator.energy(rdm1))
print("\nEffective potential a:\n", integral_operator.effective_potential(rdm1)[0])
print("\nEffective potential b:\n", integral_operator.effective_potential(rdm1)[1])
```

```
Mean-field energy:
 -1.51409740661877

Effective potential a:
 [[ 9.96146656e-01 -1.07552856e-16  2.25805701e-01]
 [-1.14491749e-16  9.24318718e-01  1.07552856e-16]
 [ 2.25805701e-01 -1.73472348e-17  1.63193773e+00]]

Effective potential b:
 [[1.16296545e+00 2.42861287e-17 6.87317394e-02]
 [6.93889390e-17 1.54136779e+00 1.90819582e-16]
 [6.87317394e-02 4.99600361e-16 1.71210367e+00]]
```

Above, RDM is initialized with `numpy` arrays in the same basis as the integral operator. The *energy()* method may also be provided with a 2-RDM (see *RestrictedTwoBodyRDM* and *UnrestrictedTwoBodyRDM*) to calculate the total, non-mean-field, energy.

In addition to the basic integral operator classes, InQuanto also includes compact integral operators: *ChemistryRestrictedIntegralOperatorCompact* and *ChemistryUnrestrictedIntegralOperatorCompact*, which exploit symmetries in the two-body integrals to reduce classical memory requirements. Compact integral operator classes support the same operations as discussed above, and are most naturally instantiated using the *inquanto-pyscf* extension.

9.5 Orbital Transformation and Optimization

Slater determinants, represented by vectors in Fock space, are functions of molecular orbitals. Typically the molecular orbitals are found by solving some other problem, such as the Hartree-Fock equations, but in some cases orbitals are chosen to be some other set of functions. For instance, they could be localized or optimized in some other way. InQuanto contains tools to help with procedures of this type, and to facilitate the transfer of these methods to quantum algorithms.

The *OrbitalTransformer* class contains several methods for common practices in molecular orbital manipulation. For instance, to Gram-Schmidt orthogonalize a set of molecular orbitals in an orthogonal atomic orbital basis,

```
from inquanto.operators import OrbitalTransformer
import numpy
orbitals = numpy.array([[1, 2], [3, 4]])
ot = OrbitalTransformer()
ot.gram_schmidt(v=orbitals, overlap=None)
```

```
array([[-0.31622777, -0.9486833 ],
       [-0.9486833 ,  0.31622777]])
```

or, equivalently, in a non-orthogonal atomic orbital basis,

```
s = numpy.array([[1.0, 0.66314574], [0.66314574, 1.0]])
ot.gram_schmidt(v=orbitals, overlap=s)
```

```
array([[-0.26746311, -1.30897671],
       [-0.80238934,  1.06823588]])
```

One can achieve the same goal of orthonormalization using the *orthonormalize()* method, which aims to find the closest orthonormal set using a single matrix transformation.

```
ot.orthonormalize(v=orbitals, overlap=s)
```

```
array([[-8.86633378e-01,  9.99418731e-01],
       [ 1.33602237e+00,  8.76208369e-04]])
```

The *OrbitalTransformer* object also holds a method for computing the unitary which relates two sets of molecular orbitals. For example, to find the unitary relating the MO coefficients in the matrix X with those in matrix C,

```
X = numpy.array([[1, 2], [3, 4]])
C = numpy.array([[2, 1], [4, 3]])
ot = OrbitalTransformer()
my_unitary = ot.compute_unitary(v_init=X, v_final=C)
print(my_unitary)
```

```
[[0.00000000e+00 1.00000000e+00]
 [1.00000000e+00 1.11022302e-16]]
```

Similarly, to transform orbitals X to orbitals C, if the unitary is known,

```
new_C = ot.transform(v=X, tu=my_unitary)
print(new_C)
```

```
[[2. 1.]
 [4. 3.]]
```

The majority of the time, the unitary is not known, and is the result of some optimisation process. The *OrbitalOptimizer* class in InQuanto is constructed with a black-box function which finds the rotational unitary, given some variational criteria. For instance, we can perform a localization which depends on some function `localize()`:

```
from inquanto.operators import OrbitalOptimizer
oo = OrbitalOptimizer(
     v_init=initial_orbitals,
     occ=[2, 2, 0, 0],
     split_rotation=False,
     functional=localize,
     minimizer=MinimizerScipy(),
     reduce_free_parameters=True
)
final_orbitals, minimising_unitary, final_value = oo.optimize()
```

The *OrbitalOptimizer* object will try to retain orbital symmetries if `point_group` and `orb_irreps` are both passed into the constructor. The function passed to the `functional` argument must be a function of a 2D array.

Note: The *OrbitalOptimizer* is also a `callable` object, but the callable execution of the optimisation returns only the optimised orbitals. This is for compatibility with the `transf` functionality in some extensions.

After the optimisation, a report can be generated with the *generate_report()* method.

CHAPTER

TEN

ANSATZES

Generally, an ansatz represents a state, with parameters to be optimized as part of a *variational algorithm* in order to find accurate approximations to ground and/or excited states. An ansatz class in InQuanto handles generation of a single circuit that can prepare the ansatz state on a quantum device, as well as manipulation of its parameters (such as symbol substitution or evaluation), keeping track of the number of qubits over which an ansatz is defined, managing reference state, and conversion of an underlying circuit into symbolic and numeric representations.

InQuanto provides a range of ansatzes for the simulation of molecules and solid state systems. The ansatz classes comprise the chemically-inspired approaches constituting the *Unitary Coupled Cluster (UCC)* ansatz family which includes the specially optimized *Chemically Aware Ansatz*; or, the less physically motivated *Hardware Efficient Ansatz (HEA)*, which has lower resource requirements when running on quantum hardware.

Many ansatzes (such as non-generalised *Unitary Coupled Cluster (UCC)*) are single-reference, as they must be applied to a simple product-type (Hartree-Fock) state (i.e. a single configuration in Hilbert space). InQuanto however also contains a method which can treat *Multiconfigurational States using Givens rotations*, allowing the user to prepare a multi-configurational state at the quantum circuit level with the configuration coefficients controlled by gate rotation angles. Finally, there is a *Real Basis Rotation Ansatz*, encoding a unitary rotation of an arbitrary basis - an important building block for many algorithms.

In addition, InQuanto provides tools for a user to construct their own ansatzes, either by using some of the intermediate base classes, such as *Trotter Ansatz* and *Fermionic Exponentiated Ansatz*, combining various ansatz classes using *Composed Ansatz*, wrapping an externally-provided circuit in a *Circuit Ansatz*, or even implementing their own ansatz class, using InQuanto's abstract `GeneralAnsatz` class.

Finally, InQuanto contains classes for representing the *Parameters* used to specify ansatzes – these contain methods for the easy construction and manipulation of parameter sets.

10.1 Trotter Ansatz

The `TrotterAnsatz` class represents a state built from a product of exponentiated Pauli strings and is at the core of the *UCC*: family of ansatzes.

The mathematical definition of `TrotterAnsatz` is as follows:

$$|\text{TrotterAnsatz}\rangle = \overleftarrow{\prod_k} e^{ip_k \sum_{l_k} \lambda_{l_k}^{(k)} \hat{P}_{l_k}^{(k)}} |\text{Ref}\rangle = \overleftarrow{\prod_k} \prod_{l_k} e^{ip_k \lambda_{l_k}^{(k)} \hat{P}_{l_k}^{(k)}} |\text{Ref}\rangle, \tag{10.1}$$

where it is assumed that for every k and $\{l_k, l'_k\}$, $\hat{P}_{l_k}^{(k)}$ and $\hat{P}_{l'_k}^{(k)}$ are mutually commuting Pauli strings, $\lambda_{l_k}^{(k)}$ is a real numerical value and p_k is a real numeric or symbolic expression. We also use the following notation for the reverse-ordered product of operators here:

$$\overleftarrow{\prod_k} \hat{O}_k = ...\hat{O}_{k+1}\hat{O}_k\hat{O}_{k-1}...\hat{O}_0. \tag{10.2}$$

When generating circuits for `TrotterAnsatz` in InQuanto, we leverage the pytket `PauliExpBox` class to prepare gates corresponding to the exponentiated Pauli strings. `PauliExpBox` constructor takes in a Pauli string object `hat{P}` and an expression object `t`, and encodes the following exponentiated expression:

$$\text{PauliExpBox}(\hat{P}, t) = e^{-\frac{i\pi}{2} t \hat{P}}. \tag{10.3}$$

We can thus re-write the definition of `TrotterAnsatz` in terms of `PauliExpBox` as follows:

$$|\text{TrotterAnsatz}\rangle = \overleftarrow{\prod_k} \prod_{l_k} \text{PauliExpBox}(\hat{P}_{l_k}^{(k)}, -\frac{2}{\pi} p_k \lambda_{l_k}^{(k)}) |\text{Ref}\rangle. \tag{10.4}$$

To construct it, one needs a reference *`QubitState`* object $|Ref\rangle$ and a *`QubitOperatorList`* object, which represents the expression to be exponentiated, and is defined as follows:

$$\begin{aligned} \text{QubitOperatorList} = [..., (p_{k-1}, \{..., (i\lambda_{l_{k-1}}^{(k-1)}, \hat{P}_{l_{k-1}}^{(k-1)}), ...\}), \\ (p_k, \{..., (i\lambda_{l_k}^{(k)}, \hat{P}_{l_k}^{(k)}), ...\}), (p_{k+1}, \{..., (i\lambda_{l_{k+1}}^{(k+1)}, \hat{P}_{l_{k+1}}^{(k+1)}), ...\}), ...] \end{aligned} \tag{10.5}$$

In this way, one should keep in mind, that only imaginary part of the supplied $\lambda_{l_k}^{(k)}$ (i.e. the coefficients of each of the *`QubitOperator`* object terms), will be taken on construction of `TrotterAnsatz`.

In InQuanto ansatzes class hierarchy, `TrotterAnsatz` is a base class of *`FermionSpaceStateExp`*, which in its turn forms a basis of all the *Unitary Coupled Cluster* ansatz classes. However, it can be used all by itself, in order to define a custom ansatz in terms of *`QubitOperator`* objects. In order to use it, one needs to provide bare-bones Pauli string terms, "wrapped" into `QubitOperator` objects, and construct a *`QubitOperatorList`* object out of them. As described above, one should think of it as of a product of exponents, with the provided Pauli strings, multiplied by symbolic expressions, each being exponentiated. Resulting `QubitOperatorList` object is then passed to a *`TrotterAnsatz`* constructor.

For instance, to recover the FCI energy of a hydrogen molecule in a minimal basis, a single Pauli word is needed:

```
from inquanto.operators import QubitOperator, QubitOperatorList
from inquanto.states import QubitState
from inquanto.ansatzes import TrotterAnsatz
from sympy import Symbol

q = QubitOperator("Y0 X1 X2 X3", 1j)
qlist = QubitOperatorList([(Symbol("x"), q)])
ansatz = TrotterAnsatz(qlist, QubitState([1, 1, 0, 0]))
```

10.2 Fermionic Exponentiated Ansatz

The *`FermionSpaceStateExp`* class is essentially a thin wrapper over *`TrotterAnsatz`*, accepting a *`FermionOperatorList`* object as an input, and mapping it to a corresponding *`QubitOperatorList`* object, serving as an input to `TrotterAnsatz`, as described in the previous chapter. This class in its turn serves as the basis for all the *Unitary Coupled Cluster* ansatzes.

If one has an idea for a chemically inspired unitary ansatz, which can be written in terms of fermionic operators, and would like to build and handle the corresponding circuit in InQuanto, this can be achieved by first instantiating a list of fermionic operators, and then providing it to a *`FermionSpaceStateExp`* constructor.

As an illustrative example, let us generate a reduced UCCD ansatz from scratch for a system of two electrons in six spin-orbitals. We first need to write the double excitations of interest:

```
from inquanto.spaces import FermionSpace
from inquanto.ansatzes import FermionSpaceAnsatzUCCD
from inquanto.operators import (
     FermionOperatorString,
     FermionOperator,
     FermionOperatorList
)
from sympy import Symbol

# Generate example state and space
space = FermionSpace(n_spin_orb=6)
state = space.generate_occupation_state(n_fermion=2)

# Construct a list of two double excitation operators
d0 = FermionOperatorString.from_string("2^ 0 3^ 1")
d1 = FermionOperatorString.from_string("4^ 0 5^ 1")
term_list = FermionOperatorList(
   [
     (Symbol("d0"), FermionOperator(d0, 1)),
     (Symbol("d1"), FermionOperator(d1, 1))
   ]
)
print(f"Fermion state: {state}")
print(f"Fermion operator list: \n{term_list}")
```

```
Fermion state: (1.0, [1, 1, 0, 0, 0, 0])
Fermion operator list:
d0         [(1, F2^ F0  F3^ F1 )],
d1         [(1, F4^ F0  F5^ F1 )]
```

Here, we should think of `term_list` as a product of exponents of single `FermionOperator` objects, constructed from a certain string of creation-annihilation fermionic operators, and pre-multiplied by symbolic terms d0 and d1. In order to ensure the ansatz operator is unitary, we make the expressions under the exponents anti-hermitian by taking the difference of each term with its adjoint. The resulting `FermionOperatorList` is then passed to the *FermionSpaceStateExp* costructor:

```
from inquanto.ansatzes import FermionSpaceStateExp
from inquanto.mappings import QubitMappingJordanWigner
anti_hermitian_term_list = FermionOperatorList(
   [(symbol, operator - operator.dagger()) for operator, symbol in term_list.items()]
)
my_ansatz = FermionSpaceStateExp(
   anti_hermitian_term_list,
   state,
   QubitMappingJordanWigner()
)
```

together with the `FermionState` object and the fermion-to-qubit mapping class of choice. The ansatz object thus constructed can generate a circuit and be used with any of the algorithms or computable objects of InQuanto.

10.3 The UCC Family

10.3.1 Unitary Coupled Cluster

This ansatz corresponds to the wavefunction representation in a variant of the (non-unitary) coupled cluster method [14], in which the operator acting on a reference state becomes unitary [15].

$$\begin{aligned} \hat{U}_{\mathrm{UCC}}(\boldsymbol{\theta}) &= e^{\hat{T}-\hat{T}^\dagger} = e^{\sum_\lambda \theta_\lambda (\hat{T}_\lambda - \hat{T}_\lambda^\dagger)} \\ |\Psi(\boldsymbol{\theta})\rangle &= \hat{U}_{\mathrm{UCC}}(\boldsymbol{\theta})|\mathrm{HF}\rangle \end{aligned} \tag{10.6}$$

Here, $\hat{T}$ is the excitation operator which has the same form as in the coupled cluster theory. In order for $\hat{U}_{\mathrm{UCC}}$ to be unitary, the exponent must be anti-hermitian, hence one subtracts the excitation operator self-adjoint under the exponent. This makes this approach natural to be implemented on a quantum circuit. By parameterising the excitations, this ansatz is suitable for *variational algorithms* in which the parameters $\boldsymbol{\theta} = \{\theta_1, \theta_2, ..., \theta_\lambda, ...\}$ are to be optimized such that the total energy is minimized according to the variational principle. The reference wavefunction is often the Hartree-Fock ground state, but excited configurations are also possible, as long as $\hat{U}_{\mathrm{UCC}}$ is applied to a single configuration. Truncating excitations to singles and doubles leads to the (*FermionSpaceAnsatzUCCSD*) ansatz. Doubles-only (*FermionSpaceAnsatzUCCD*) ansatz is also available.

Consider the following example for a system of 4 spin orbitals and 2 electrons (applicable to the H_2 molecule in minimal basis):

```
from inquanto.states import FermionState
from inquanto.spaces import FermionSpace
from inquanto.mappings import QubitMappingJordanWigner
from inquanto.ansatzes import FermionSpaceAnsatzUCCD, FermionSpaceAnsatzUCCSD

space = FermionSpace(4)
state = FermionState([1, 1, 0, 0])
jw_map = QubitMappingJordanWigner()

ansatz_uccsd = FermionSpaceAnsatzUCCSD(
    fermion_space=space, fermion_state=state, qubit_mapping=jw_map
)

ansatz_uccd = FermionSpaceAnsatzUCCD(
    fermion_space=space, fermion_state=state, qubit_mapping=jw_map
)

print("UCCSD parameters:", ansatz_uccsd.state_symbols)

print("\nUCCD parameters:", ansatz_uccd.state_symbols)

report_uccsd = ansatz_uccsd.generate_report()
report_uccd = ansatz_uccd.generate_report()

print("\nNumber of parameters: {} (UCCSD), {} (UCCD)".format(
    report_uccsd['n_parameters'],
    report_uccd['n_parameters'])
)
print("Number of qubits: {} (UCCSD), {} (UCCD)".format(
    report_uccsd['n_qubits'],
    report_uccd['n_qubits'])
)
print("Ansatz circuit depth: {} (UCCSD), {} (UCCD)".format(
    report_uccsd['ansatz_circuit_depth'],
```

(continues on next page)

(continued from previous page)

```
    report_uccd['ansatz_circuit_depth'])
)

uccsd_circuit = ansatz_uccsd.get_circuit(
ansatz_uccsd.state_symbols.construct_random()
)
uccd_circuit = ansatz_uccd.get_circuit(
ansatz_uccd.state_symbols.construct_random()
)
```

```
UCCSD parameters: [s0, s1, d0]

UCCD parameters: [d0]

Number of parameters: 3 (UCCSD), 1 (UCCD)
Number of qubits: 4 (UCCSD), 4 (UCCD)
Ansatz circuit depth: 58 (UCCSD), 29 (UCCD)
```

In the above script, we have built UCCSD and UCCD ansatzes objects using the *fermionic orbital space* (`FermionSpace`) and *occupation state* (`FermionState`) objects, as well as our choice of *fermion-to-qubit mapping* (`QubitMappingJordanWigner`). The symbols corresponding to the ansatzes parameters can be accessed by the `state_symbols` attribute, which returns an InQuanto `SymbolSet` object that stores the `SymPy` symbols. For more information on how to access and manipulate symbolic parameters in InQuanto please see the *corresponding* section. Useful information about the ansatz circuit such as number of qubits, number of parameters, and circuit depth is available from the `dict` returned by the `generate_report()` method of each ansatz object. The tket`Circuit` object can be generated using the `get_circuit()` method (which needs numeric values for the parameters, and we provide random values here using the `state_symbols.construct_random()` method).

For this system, we see three parameters for UCCSD: 2 single (alpha-beta single excitations are not allowed in order to preserve spin symmetry), and 1 double excitation. Exclusion of the singles in the UCCD ansatz leads to only one excitation – the double. This is consistent with the minimal basis H_2. As we can see, fewer excitations in UCCD leads to equally less parameters and a reduced circuit depth compared to the UCCSD ansatz.

InQuanto generates the excitation operators for UCC using the `FermionOperatorList` class (see section *Fermion Operators & States*). To construct a UCC ansatz object, anti-hermitian UCC excitations must be transformed to qubit operators via a provided *fermion to qubit mapping* method and exponentiated. To facilitate its implementation on a quantum circuit, the UCC ansatz must have a trotterised form (facilitated by the `FermionOperatorList` internal structure), i.e. an operator is approximated e.g. as

$$e^{\hat{T}-\hat{T}^\dagger} \approx \prod_\lambda e^{\theta_\lambda(\hat{T}_\lambda-\hat{T}_\lambda^\dagger)} \tag{10.7}$$

which corresponds to the first order Trotter decomposition. In general this is an approximation, which leads to (usually small) variations of the energy that depend on the order of terms in the product $\prod_\lambda e^{\theta_\lambda(\hat{T}_\lambda-\hat{T}_\lambda^\dagger)}$. For variational algorithms, it is expected that this error will be absorbed by the variational procedure; however, this must be considered – particularly when using non-variational algorithms. All the UCC-family ansatzes in InQuanto use first-order trotterisation. In order to use higher-order approximation, one needs to use their base `FermionSpaceStateExp` class, as explained in the corresponding *section*.

10.3.2 Generalised Unitary Coupled Cluster

The UCC ansatz discussed in the previous section refers to set of coupled cluster operators with a well defined separation between occupied and unoccupied (virtual) orbital spaces, such that all excitations are transitions between occupied and virtual spin orbitals. Lee et al [16] proposed variations of UCC in which occupied-to-occupied and virtual-to-virtual excitations are also included. The direct extension of UCC(S)D ansatzes to include these generalised transitions is referred to as UCCG(S)D. Here:

$$\begin{aligned}\hat{T}^{(1G)} &= \sum_{p,q} t_p^q \hat{f}_q^\dagger \hat{f}_p \\ \hat{T}^{(2G)} &= \sum_{p,q,r,s} t_{p,q}^{r,s} \hat{f}_r^\dagger \hat{f}_s^\dagger \hat{f}_p \hat{f}_q \\ \hat{T}_{\text{GSD}} &= \hat{T}^{(1G)} + \hat{T}^{(2G)} \\ \hat{U}_{\text{UCCGSD}}(\boldsymbol{\theta}) &= e^{\hat{T}_{\text{GSD}} - \hat{T}_{\text{GSD}}^\dagger},\end{aligned} \tag{10.8}$$

where p, q, r, s run over both occupied and virtual spin orbital spaces. Note there is a one-to-one mapping between the set of parameters (labelled by generic indexes for cluster operators) and the set of excitation coefficients for singles and doubles (labelled by orbitals involved in the transition), i.e. $\boldsymbol{\theta} = \{\theta_1, \theta_2, ..., \theta_\lambda, ...\} \mapsto \{t_p^q, ..., t_{p,q}^{r,s}, ...\}$. The UCCG(S)D ansatzes are instantiated in the same way as UCC(S)D (see the code snippet in *Unitary Coupled Cluster*), by simply replacing `FermionSpaceAnsatzUCCSD` (`FermionSpaceAnsatzUCCD`) with *`FermionSpaceAnsatzUCCGSD`* (*`FermionSpaceAnsatzUCCGD`*) for singles+doubles (doubles only).

A more compact form of generalised UCC has also been proposed in which the double excitations are restricted to pair-doubles, i.e. transitions of a pair of electrons between the two spatial orbitals (but spatial orbital indexes still span the general range as above):

$$\hat{T}^{(2pG)} = \sum_{(p\alpha,p\beta),(q\alpha,q\beta)} t_{p\alpha,p\beta}^{q\alpha,q\beta} \hat{f}_{q\alpha}^\dagger \hat{f}_{q\beta}^\dagger \hat{f}_{p\alpha} \hat{f}_{p\beta} \tag{10.9}$$

where α, β label spins and p, q - spatial orbitals, and the singles are as in $T^{(1G)}$. This ansatz, referred to as k-UpCCGSD [16] (*`FermionSpaceAnsatzkUpCCGSD`*), is expressed as a product of k factors of cluster operators, each one with an independent set of parameters:

$$\begin{aligned}\hat{T}^{(k)} &= \hat{T}^{(1G)} + \hat{T}^{(2pG)} \\ \hat{U}_{k\text{UpCCGSD}}(\boldsymbol{\theta}) &= \prod_k e^{\hat{T}^{(k)} - \hat{T}^{(k)\dagger}}.\end{aligned} \tag{10.10}$$

The following snippet demonstrates the initialization of the UCCGSD and k-UpCCGSD ansatzes:

```
from inquanto.states import FermionState
from inquanto.spaces import FermionSpace
from inquanto.mappings import QubitMappingJordanWigner
from inquanto.ansatzes import FermionSpaceAnsatzkUpCCGSD, FermionSpaceAnsatzUCCGSD

space = FermionSpace(4)
state = FermionState([1, 1, 0, 0])
jw_map = QubitMappingJordanWigner()

ansatz_uccgsd = FermionSpaceAnsatzUCCGSD(
    fermion_space=space, fermion_state=state, qubit_mapping=jw_map
)

ansatz_kupccsd = FermionSpaceAnsatzkUpCCGSD(
    fermion_space=space, fermion_state=state, k_input=2, qubit_mapping=jw_map
)
```

(continues on next page)

(continued from previous page)

```
print("UCCGSD parameters:", ansatz_uccgsd.state_symbols)

print("\nkUpCCGSD parameters:", ansatz_kupccsd.state_symbols)
```

```
UCCGSD parameters: [gs0, gs1, gd0]

kUpCCGSD parameters: [gs0k0, gs1k0, gd0k0, gs0k1, gs1k1, gd0k1]
```

In this case, there are 2 duplicates for the set of singles and doubles of k-UpCCGSD, as the input value of k is set to 2.

10.4 Chemically Aware Ansatz

Chemically aware ansatzes (*FermionSpaceStateExpChemicallyAware* and *FermionSpaceAnsatzChemicallyAwareUCCSD*) benefit from a special excitation regrouping strategy, aimed at reducing the two-qubit gate count.

The method consists of:

- Regrouping excitations to arrange the spatial-to-spatial excitations in a consecutive sequence.
- Mapping spatial-to-spatial excitations directly to a circuit, to reduce from 64 to 2 two-qubit gates per excitation (+ a small overhead of a handful of CX-gates).
- Spin-orbitals encoded to qubits with Jordan–Wigner mapping.
- Ansatz circuit synthesis is by exploiting commuting sets of excitations.

For more details on the method please refer to [17].

The ansatz can be used similarly to other ansatzes in the UCC family. Once a `space` and a reference `state` objects are available, the `FermionSpaceAnsatzChemicallyAwareUCCSD` can be instantiated in the same way as `FermionSpaceAnsatzUCCSD`:

```
from pytket.circuit import OpType

from inquanto.ansatzes import FermionSpaceAnsatzUCCSD,␣
↪FermionSpaceAnsatzChemicallyAwareUCCSD

from inquanto.spaces import FermionSpace
from inquanto.states import FermionState

space = FermionSpace(8)
state = FermionState([1, 1, 0, 0, 0, 0, 0, 0])

ansatz = FermionSpaceAnsatzUCCSD(space, state)
chemically_aware_ansatz = FermionSpaceAnsatzChemicallyAwareUCCSD(space, state)

print(ansatz.state_circuit.depth_by_type(OpType.CX))
print(chemically_aware_ansatz.state_circuit.depth_by_type(OpType.CX))
```

```
187
```

```
156
```

The depth of the resulting state circuit of `chemically_aware_ansatz` is smaller than that of the `ansatz`. However, the user should be aware that the rearrangement of excitations may impact the Trotter error - either positively or negatively.

For even better circuit depth, reduction symmetry filtering is recommended. For example, for H2 molecule, basis set `631g`, we can use the symmetry aware `FermionSpace`:

```
from inquanto.symmetry import PointGroup

point_group = PointGroup("D2h")
orb_irreps = ["Ag", "Ag", "B1u", "B1u", "Ag", "Ag", "B1u", "B1u"]
space_pg = FermionSpace(8, point_group=point_group, orb_irreps=orb_irreps)

ansatz = FermionSpaceAnsatzUCCSD(space_pg, state)
chemically_aware_ansatz = FermionSpaceAnsatzChemicallyAwareUCCSD(space_pg, state)

print(ansatz.state_circuit.depth_by_type(OpType.CX))
print(chemically_aware_ansatz.state_circuit.depth_by_type(OpType.CX))
```

```
96
62
```

The `FermionSpaceStateExpChemicallyAware` class can be used for any sequence of single and double excitations in the same way as `FermionSpaceStateExp`, and it will give greater flexibility to compose an excitation list with more spatial-to-spatial excitations.

10.5 Hardware Efficient Ansatz

The hardware efficient ansatz (HEA, *`HardwareEfficientAnsatz`*) [18] is a lower-depth alternative to chemistry-inspired ansatzes, such as UCCSD. It is "physics-agnostic" in the sense that couplings/entanglements between qubits do not take into account chemical/physical information. Rather, the circuit structure consists of layers of rotations and entangling operations; each layer contains blocks of single qubit rotation gates separated by a block of 2-qubit entanglers. While this leads to circuits that are more efficient in depth compared to UCC, the HEA ansatz does not consider spin or particle number symmetries, so unwanted symmetry breaking is more likely for HEA during a variational algorithm. Hence, this ansatz should be used with caution.

The unitary operator corresponding to HEA, with one block of R_x rotations per layer, can be expressed as:

$$\begin{aligned}\hat{U}_{\mathrm{HEA}}(\boldsymbol{\theta}) = &\left(\prod_i^{N_\mathrm{q}} \hat{U}_{R_x}(\theta_{i,N_\mathrm{L}})\right) \hat{U}_{\mathrm{Ent}} \left(\prod_i^{N_\mathrm{q}} \hat{U}_{R_x}(\theta_{i,N_\mathrm{L}-1})\right) \hat{U}_{\mathrm{Ent}} \ldots \times \\ \times \ldots &\left(\prod_i^{N_\mathrm{q}} \hat{U}_{R_x}(\theta_{i,l})\right) \hat{U}_{\mathrm{Ent}} \ldots \left(\prod_i^{N_\mathrm{q}} \hat{U}_{R_x}(\theta_{i,1})\right) \hat{U}_{\mathrm{Ent}} \left(\prod_i^{N_\mathrm{q}} \hat{U}_{R_x}(\theta_{i,\mathrm{cap}})\right)\end{aligned} \tag{10.11}$$

where N_q is the number of qubits, N_L is the number of layers, and index l labels the layers. The final layer ($l = 1$) is terminated with another set of single qubit rotations (the cap or "capping" block).

Here, a parameter $\theta_{i,l}$ corresponds to a single qubit rotation applied to the i^{th} qubit in the l^{th} HEA layer (the capping block also has independent rotation angles).

Each rotation block can consist of R_x, R_y, and/or, R_z rotations (hence each layer can have up to 3 blocks of rotations - the selection of which is also reflected in the number of capping blocks at the end of the circuit), while each entangling block consists of controlled-X (CNOT) 2-qubit gates. The following example shows how to create a HEA ansatz using InQuanto. This example is for 2 HEA layers, each one consisting of a block of R_x rotations, a block of R_y rotations, and

a block of CNOT 2-qubit entanglers (with the final capping blocks of R_x and R_y rotations applied after the second HEA layer)

```
from inquanto.ansatzes import HardwareEfficientAnsatz
from inquanto.states import QubitState

from inquanto.core._tket import OpType

ansatz = HardwareEfficientAnsatz(
    [OpType.Rx, OpType.Ry], reference=QubitState([1, 1, 0, 0]), n_layers=2
)

report_hea = ansatz.generate_report()

print("Number of parameters: {}".format(
    report_hea['n_parameters'])
)
print("Number of qubits: {}".format(
    report_hea['n_qubits'])
)
print("Ansatz circuit depth: {}".format(
    report_hea['ansatz_circuit_depth'])
)

hea_circuit = ansatz.get_circuit(
ansatz.state_symbols.construct_random()
)
```

```
Number of parameters: 24
Number of qubits: 4
Ansatz circuit depth: 12
```

As with all other InQuanto Ansatz objects, the `generate_report()` method can be used to extract information about the quantum circuit representing the HEA ansatz, and the circuit object itself can be accessed by the `get_circuit()` method. Unlike UCC, this is an ansatz that is built at the circuit level directly, hence a fermion-to-qubit mapping is not needed to construct the HEA ansatz object.

10.6 Circuit Ansatz

One might wish to provide a completely custom circuit, and use it as an InQuanto ansatz (e.g. in order to use it in InQuanto *Algorithms*). This is possible by simply passing a tket `Circuit` object to a *`CircuitAnsatz`* constructor.

For example, for the minimal basis H_2, one could recover the FCI energy with the following custom circuit:

```
from pytket import circuit
from math import pi

circ = circuit.Circuit(4)
circ.X(0)
circ.X(1)
circ.CX(0, 1)
circ.CX(0, 2)
circ.CX(0, 3)
circ.V(0)
```

(continues on next page)

(continued from previous page)

```
# Add custom block, of redefined Rz
theta = Symbol('\theta')
subcirc = circuit.Circuit(1)
subcirc.Rz(theta+pi, 0)
R = circuit.CustomGateDef.define("R", subcirc, [theta])
theta_index = circuit.fresh_symbol(r't_0')
circ.add_custom_gate(R, [theta_index], [0])

circ.Vdg(0)
circ.CX(0, 3)
circ.CX(0, 2)
circ.CX(0, 1)
circ.V(0); circ.V(1); circ.V(2); circ.V(3)
circ.S(0); circ.S(1); circ.S(2); circ.S(3)
circ.H(0); circ.H(1); circ.H(2); circ.H(3)
circ.S(0); circ.S(1); circ.S(2); circ.S(3)

from inquanto.ansatzes import CircuitAnsatz

my_ansatz = CircuitAnsatz(circ)
```

10.7 Multiconfiguration States using Givens rotations

This ansatz uses multicontrolled Givens rotations to prepare a quantum circuit corresponding to a linear combination of determinants, where the latter are selected by the user and represented by the terms (occupation configurations) of an InQuanto *FermionState* or *QubitState*. The resulting object can be used as i) an ansatz itself, or ii) a multiconfigurational generalisation of a mean-field single reference, to which another ansatz (that allows for multireference initial states) can be applied. In both of these cases, InQuanto allows for fixed and variational configuration coefficients.

The methodology of preparing these circuits is based on proofs that multicontrolled Givens rotations are universal for quantum chemistry [19]. A Givens rotation is a unitary operation that linearly mixes between two configurations. Restricting to real coefficients, a 2-qubit (1-body) Givens unitary can be written in the 2-qubit computational basis as

$$G_1(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{10.12}$$

which corresponds to a rotation between the $|10\rangle$ and $|01\rangle$ basis states. This has a straight-forward generalisation to n qubits [19]. For example, if one chooses to mix $|1100\rangle$ and $|0011\rangle$, this can be done using a 4-qubit (2-body) Givens rotation G_2 applied to either $|1100\rangle$ or $|0011\rangle$ (and where G_2 is a 4×4 generalisation of G_1, with appropriate lexicographical ordering)

$$\begin{aligned} G_2(\theta)|1100\rangle &= \cos(\theta)|1100\rangle + \sin(\theta)|0011\rangle \\ G_2(\theta)|0011\rangle &= \cos(\theta)|0011\rangle - \sin(\theta)|1100\rangle \end{aligned} \tag{10.13}$$

Hence the configuration coefficients correspond to elements of the unitary, are tied to the rotation angles of the corresponding gates that implement the unitary, and must be normalised to unity.

The following example shows how to use InQuanto to prepare a circuit which generates a state corresponding to a linear mixing of these 2 basis configurations

```
from inquanto.ansatzes import MultiReferenceState
from inquanto.states import QubitState, QubitStateString

from pytket.extensions.qiskit import AerStateBackend

from math import sqrt

config1, c_1 = QubitStateString([1, 1, 0, 0]), 1/sqrt(2)
config2, c_2 = QubitStateString([0, 0, 1, 1]), 1/sqrt(2)
qubit_states = QubitState({config1: c_1, config2: c_2})

multi_state = MultiReferenceState(qubit_states)
multi_state_circuit = AerStateBackend().get_compiled_circuit(multi_state.get_
↪circuit())
print("Circuit depth:", multi_state_circuit.depth())
print("Number of gates:", multi_state_circuit.n_gates)
```

```
Circuit depth: 21
Number of gates: 34
```

Here we have chosen the coefficients of both configurations $c_1 = c_2 = 1/\sqrt{2}$, but any real values c_i are accepted as long as $\sum_i |c_i|^2 = 1$.

When linearly combining more than two determinants using a series of unitaries applied to the circuit, all rotations must be maintained within the desired subspace of the full Hilbert space. This guarantees, for example, that spin and particle numbers are invariant with respect to the transformation corresponding to the mixing unitaries, and prevents unwanted configurations from entering the full expansion of the rotated state vector. One way to guarantee this is to control (or in general multicontrol) the application of some Givens unitaries on the qubits of the "initial" term (i.e. the first configuration in the sequence of configurations supplied by *FermionState* or *QubitState*), depending on the position of a particular Givens unitary in the circuit [19]. In InQuanto, this is implemented in a general way that guarantees the circuit will be in the state corresponding precisely to a linear combination of the user-defined basis configurations, regardless of how many configurations are chosen. However, the size of the circuits can increase dramatically when multicontrolled operations are present. For example, adding just one basis configuration to the example above significantly increases the circuit size when compiled to a backend gate set, largely because of additional multicontroled gates:

```
from inquanto.ansatzes import MultiReferenceState
from inquanto.states import QubitState, QubitStateString

from pytket.extensions.qiskit import AerStateBackend

from math import sqrt

config1, c_1 = QubitStateString([1, 1, 0, 0]), 1/sqrt(3)
config2, c_2 = QubitStateString([0, 0, 1, 1]), 1/sqrt(3)
config3, c_3 = QubitStateString([1, 0, 0, 1]), 1/sqrt(3)
qubit_states = QubitState({config1: c_1, config2: c_2, config3: c_3,})

multi_state = MultiReferenceState(qubit_states)
multi_state_circuit = AerStateBackend().get_compiled_circuit(multi_state.get_
↪circuit())
print("Circuit depth:", multi_state_circuit.depth())
print("Number of gates:", multi_state_circuit.n_gates)
```

```
Circuit depth: 146
Number of gates: 166
```

Basis states separated by more than two-body rotations (e.g. a rotation corresponding to a triples excitation) can also

be linearly combined. For this, a sequence of multicontrolled SWAPs combined with a multicontrolled 2-qubit (1-body) Givens rotation [19] is utilized; such a multicontrolled "SWAP+rotation ladder" can be used to represent a general chemical excitation to any order on a quantum circuit, and it is used to simulate (> 2)-body rotations in InQuanto. However, again due to the use of multicontrols, a large increase in circuit depth is observed when comparing a 2-body mixing to e.g. 3-body mixing, after compiling to a native gate set. The following example shows this by first preparing $\psi\rangle = \frac{1}{\sqrt{2}}|111000\rangle + \frac{1}{\sqrt{2}}|100011\rangle$ (separated by a 2-body excitation), then preparing $\psi\rangle = \frac{1}{\sqrt{2}}|111000\rangle + \frac{1}{\sqrt{2}}|000111\rangle$ (separated by a 3-body excitation), and then comparing the size of the corresponding circuits:

```
from inquanto.ansatzes import MultiReferenceState
from inquanto.states import QubitState, QubitStateString

from pytket.extensions.qiskit import AerStateBackend

from math import sqrt

backend = AerStateBackend()

config1, c_1 = QubitStateString([1, 1, 1, 0, 0, 0]), 1/sqrt(2)
config2, c_2 = QubitStateString([1, 0, 0, 0, 1, 1]), 1/sqrt(2)
qs_2bodyrot = QubitState({config1: c_1, config2: c_2})

ms_2bodyrot = MultiReferenceState(qs_2bodyrot)
ms_circ_2bodyrot = backend.get_compiled_circuit(ms_2bodyrot.get_circuit())
print("Circuit depth, configs separated by 2-body excitation:", ms_circ_2bodyrot.
↪depth())
print("Number of gates, configs separated by 2-body excitation:", ms_circ_2bodyrot.n_
↪gates)

config1, c_1 = QubitStateString([1, 1, 1, 0, 0, 0]), 1/sqrt(2)
config2, c_2 = QubitStateString([0, 0, 0, 1, 1, 1]), 1/sqrt(2)
qs_3bodyrot = QubitState({config1: c_1, config2: c_2})

ms_3bodyrot = MultiReferenceState(qs_3bodyrot)
ms_circ_3bodyrot = backend.get_compiled_circuit(ms_3bodyrot.get_circuit())
print("\nCircuit depth, configs separated by 3-body excitation:", ms_circ_3bodyrot.
↪depth())
print("Number of gates, configs separated by 3-body excitation:", ms_circ_3bodyrot.n_
↪gates)
```

```
Circuit depth, configs separated by 2-body excitation: 21
Number of gates, configs separated by 2-body excitation: 35

Circuit depth, configs separated by 3-body excitation: 1935
Number of gates, configs separated by 3-body excitation: 2790
```

How much the circuit size increases due to the presence of multicontrols depends crucially on their decomposition to a particular gate set. The well-known approach based on recursive decomposition of Toffoli gates yields quadratic scaling with respect to the number of control qubits [20]. Currently, tket utilizes a linear depth scaling approach for decomposing multicontrols [21] during circuit compilation for intermediate ($6 \leq n \leq 50$) numbers of qubits. However, the user should in general be cautious of large circuits resulting from the mixing of >2 basis configurations and/or mixing configurations that are separated by (> 2)-body rotations.

In all the examples shown above, the resulting ansatz object corresponds to a multiconfiguration state with fixed coefficients. In the next example we show how to prepare an ansatz object with variational coefficients. This ansatz can then be used in an InQuanto computable which in turn can be plugged into an InQuanto algorithm in the usual ways (see *Algorithms* and *Computables* sections) to optimize the coefficients and obtain the ground state.

```
from inquanto.ansatzes import GivensAnsatz
from inquanto.spaces import FermionSpace
from inquanto.states import FermionState, FermionStateString

fspace = FermionSpace(4)

fss_ref = FermionStateString([1, 1, 0, 0])
fss_2 = FermionStateString([0, 0, 1, 1])
fock_states = FermionState({fss_ref: 1.0, fss_2: 1.0})    # coefficients ignored here
ansatz = GivensAnsatz(fock_states.terms)
# |psi> = a|1100> + b|0011> , b = SQRT(1 - |a|^2) => only 1 parameter in VQE

print("Ansatz parameter info")
print("symbols:", ansatz.state_symbols.symbols)
print("N_parameters:", ansatz.n_symbols)
```

```
Ansatz parameter info
symbols: [theta0]
N_parameters: 1
```

Then, by optimizing the rotation angles of the Givens unitaries via VQE, a quantum circuit version of configuration interaction is achieved in which the determinants are selected *a-priori* followed by a variational optimisation of their coefficients.

Note that while coefficients need to be supplied to the *FermionState*, they are not used in the construction of the ansatz class so the user does not have to choose values that normalise to unity (this is at variance to *MultiReferenceState* in which the gate angles are calculated for a given set of coefficients during instantiation) - in this case the rotation angles of the object returned by *GivensAnsatz* are symbolic. However, the symbols representing these coefficients are still related by normalisation, hence in this example there is only 1 variational parameter.

10.8 Real Basis Rotation Ansatz

Orbital basis transformation can be represented on a quantum circuit with Givens rotation gates (https://arxiv.org/abs/1711.04789). This circuit representation allows us to define the *RealBasisRotationAnsatz*,

$$|\Psi(\theta)\rangle = \exp\left[\sum_{ij} \theta_{ij} a_i^\dagger a_j\right] |\text{Ref}\rangle \tag{10.14}$$

where the relationship between a basis transformation matrix and the variatonal parameters is $\theta_{ij} = [\ln R]_{ij}$. This ansatz can be used in variatonal algorithms to find for example the mean-field solution of a chemistry Hamiltonian on quantum computer:

```
from pytket.extensions.qiskit import AerStateBackend

from inquanto.ansatzes import RealBasisRotationAnsatz
from inquanto.express import load_h5, run_vqe
from inquanto.states import QubitState

reference = QubitState([1, 1, 0, 0])
ra = RealBasisRotationAnsatz(reference=reference)

h2_sto3g = load_h5("h2_sto3g.h5", as_tuple=True)
```

(continues on next page)

(continued from previous page)

```
hamiltonian_lowdin = h2_sto3g.hamiltonian_operator_lowdin.qubit_encode()

print("HF energy (ref):  ", h2_sto3g.energy_hf)
print("<REF|H|REF>:      ", reference.vdot(hamiltonian_lowdin.dot_state(reference)))

vqe = run_vqe(ra, hamiltonian_lowdin, AerStateBackend(), initial_parameters=ra.state_
↪symbols.construct_random() )

print("VQE energy:       ", vqe.final_value)
```

```
HF energy (ref):   -1.1175058842043315
<REF|H|REF>:       -0.12440620192256766
# TIMER BLOCK-0 BEGINS AT 2023-05-02 11:27:34.609809
```

```
# TIMER BLOCK-0 ENDS - DURATION (s): 30.8161050 [0:00:30.816105]
VQE energy:        -1.117505884195662
```

If R is a real, unitary matrix, we can also compute the corresponding ansatz parameters:

```
import numpy

# unitary matrix for which the QR gives R with diagonals [1,1,1,-1]
R = numpy.array(
    [
        [0.04443313, -0.95103332, 0.1990091, -0.2322858],
        [-0.14642999, -0.16713913, -0.96063852, -0.16672251],
        [0.98783978, 0.02520736, -0.14785981, -0.04092234],
        [-0.02750505, 0.25877542, 0.1253255, -0.95737781],
    ]
)

p = ra.ansatz_parameters_from_unitary(R)
print(f"Rotation parameters:\n{p}")
```

```
Rotation parameters:
{theta0: 0.8624009177850535, theta1: -0.31117806797812503, theta2: -1.010574433602697,
↪ theta3: -1.526348563129208, theta4: -1.717901058621577, theta5: 0.
↪027836442642734202, phi0: 0.0, phi1: 0.0, phi2: 0.0, phi3: 3.141592653589793}
```

10.9 Composed Ansatz

Finally, let us explain how to merge two ansatzes into a single one in InQuanto, by using the *ComposedAnsatz* class: one just needs to pass the two ansatz objects to its constructor, with the order of parameters being the inverse of how their circuits are to be ordered with respect to each other.

For example, if one would like to perform a multi-reference generalised UCC calculation for a system of two electrons in four spin-orbitals, one could consider employing *GivensAnsatz* to define a singly-excited multi-configuration reference state with variable parameters (CI coefficients), and combining it with an empty-reference *FermionSpaceAnsatzUCCGD*:

```
from inquanto.spaces import FermionSpace
from inquanto.states import FermionStateString, FermionState
from inquanto.ansatzes import (
```

(continues on next page)

(continued from previous page)

```
    GivensAnsatz,
    FermionSpaceAnsatzUCCGD,
    ComposedAnsatz
)
from numpy import sqrt

space = FermionSpace(4)
fss_ref = FermionStateString([1, 1, 0, 0])
fss_1001 = FermionStateString([1, 0, 0, 1])
fss_0110 = FermionStateString([0, 1, 1, 0])
# Note: coefficients don't actually matter as we are building a symbolic ansatz.
fstate_multiconf = FermionState({fss_ref: sqrt(0.8), fss_1001: sqrt(0.1), fss_0110:␣
↪sqrt(0.1)})

ansatz_givens = GivensAnsatz(fstate_multiconf.terms)
ansatz_uccgd = FermionSpaceAnsatzUCCGD(space, FermionState([0, 0, 0, 0]))
ansatz_multiref = ComposedAnsatz(ansatz_uccgd, ansatz_givens)
```

In this particular case the resulting multireference asnatz will be identical to a UCCGSD ansatz, constructed with the same Fock space and HF reference state, but for more complicated Fock spaces one could cherry-pick some important reference states and use them in conjunction with a shallower UCC ansatz. Other use cases of combining various ansatzes can be easily envisaged.

10.10 Parameters

As discussed above, ansatzes are typically defined by a set of parameters. These parameters are handled in InQuanto by means of the two classes: `SymbolSet` and `SymbolDict`. Both represent symbolic parameters – the difference between them is that `SymbolDict` associates each parameter with a specific value, whereas `SymbolSet` does not. In essence, both are wrappers over a Python `dict` data structure (with the `SymbolSet` values set to `None` and non-accessible), ensuring consistency in the order in which symbols are added. The keys of a `SymbolSet` or `SymbolDict` are SymPy `Symbol` objects, and thus may be manipulated or created using the SymPy library if required. Additional convenience methods for `SymbolSet` and `SymbolDict` are also included. In general, when an ansatz is being constructed, its `.state_symbols` member variable represents a `SymbolSet` object - just a set of symbolic parameters, without any values assigned. One can also retrieve a `SymbolSet` object from the symbolic operator objects using the `.free_symbols()` method:

```
operator = FermionOperatorList.from_string("d0 [(1.0, F2^ F0 F3^ F1)], d1 [(1.0, F4^␣
↪F0 F5^ F1)]")
symbols = operator.free_symbols()
print(symbols)
```

```
{d0, d1}
```

and substitute free symbols with the numeric values using the `.subs()` method:

```
operator.subs({Symbol("d0"): 0.9, Symbol("d1"): 0.1})
print(operator)
```

```
d0        [(1.0, F2^ F0  F3^ F1 )],
d1        [(1.0, F4^ F0  F5^ F1 )]
```

When initiating an `Algorithm` object, it is common to pass initial values for ansatz parameters. Here, convenience methods such as `.construct_from_array()`, `construct_zeros()` and `construct_random()` can be

used to convert a `SymbolSet` object to a `SymbolDict` object, with each symbol being mapped to its value. The order of symbols is maintained in this conversion. For example, with the custom circuit constructed above:

```
random_param = my_ansatz.state_symbols.construct_random()
print(random_param)
```

```
{t_0: 0.9417154046806644}
```

If necessary, a `SymbolDict` object can be manipulated in the same way as an ordinary Python `dict`, i.e. by updating, adding or removing elements:

```
from inquanto.core import SymbolDict
sd = SymbolDict(a=1, b=2)
sd["b"] = 3
sd["c"] = 2
sd.discard("a")
print(sd)
```

```
{b: 3, c: 2}
```

CHAPTER

ELEVEN

SYMMETRY

The use of symmetry to simplify computational problems in quantum chemistry is well-established. A symmetry of an operator $\hat{O}$ can be described by an operator $\hat{S}$ on the same Hilbert space, where the two operators $\hat{O}$ and $\hat{S}$ commute. InQuanto contains special purpose classes to represent symmetry operators in both the fermionic (*inquanto.operators.SymmetryOperatorFermionic*) and qubit (*inquanto.operators.SymmetryOperatorPauli*) spaces. These subclass their respective operator classes, and as such may otherwise be constructed and used in the same way. Both classes provide additional functionality related to symmetry, above the normal *inquanto.operators.FermionOperator* and *inquanto.operators.QubitOperator classes*. *is_symmetry_of()* can verify (by commutation) that a given symmetry operator is indeed a symmetry of a given operator within the same space. *symmetry_sector()* yields the symmetry sector that a provided state is in - i.e. the expectation value of the symmetry operator with the provided state.

```
from inquanto.operators import SymmetryOperatorPauli
from inquanto.operators import QubitOperator
from inquanto.states import QubitState

symmetry = SymmetryOperatorPauli("Z0 Z1 Z2 Z3")
operator = QubitOperator("X0 X1 X2 X3")
state = QubitState([1,1,0,0])

print("Is it a symmetry? ", symmetry.is_symmetry_of(operator))
print("Symmetry sector with state: ", symmetry.symmetry_sector(state))
```

```
Is it a symmetry?  True
Symmetry sector with state:  1.0
```

Warning: Calculating a sector is performed by determining the expectation value of the state with the symmetry operator. For typical purposes (for instance, calculating the sector of a Hartree-Fock state with Z2 symmetry operators) this will be both efficient and produce predictable results. However, validity checking is not performed and the use of symmetry-inconsistent states may lead to inconsistent results.

11.1 Finding symmetries

InQuanto is capable of automatically generating symmetry operators on both fermionic and qubit spaces; however, the method of generation is different in each case. For fermionic Hilbert spaces, it is common to have some prior knowledge regarding the physical point group, electron number and spin symmetries. This information is typically provided by a precursor classical calculation and is associated with the *FermionSpace* class. InQuanto uses this information to generate the Z2 symmetry operators corresponding to the Z2 point group symmetries (using orbital irreducible representation information), the electron number parity and spin parity symmetries. This can be performed using the *symmetry_operators_z2()* method:

```
from inquanto.spaces import FermionSpace
point_group_label = "D2h"
irrep_labels = ["Ag", "Ag", "B1u", "B1u"]
fermion_space = FermionSpace(4, point_group_label, irrep_labels)
symmetry_operators_fermion = fermion_space.symmetry_operators_z2()
for x in symmetry_operators_fermion:
    print(x)
```

```
(1.0, ), (-2.0, F3^ F3 ), (-2.0, F2^ F2 ), (4.0, F2^ F2  F3^ F3 )
(1.0, ), (-2.0, F3^ F3 ), (-2.0, F2^ F2 ), (4.0, F2^ F2  F3^ F3 ), (-2.0, F1^ F1 ),␣
↪(4.0, F1^ F1  F3^ F3 ), (4.0, F1^ F1  F2^ F2 ), (-8.0, F1^ F1  F2^ F2  F3^ F3 ), (-
↪2.0, F0^ F0 ), (4.0, F0^ F0  F3^ F3 ), (4.0, F0^ F0  F2^ F2 ), (-8.0, F0^ F0  F2^␣
↪F2  F3^ F3 ), (4.0, F0^ F0  F1^ F1 ), (-8.0, F0^ F0  F1^ F1  F3^ F3 ), (-8.0, F0^␣
↪F0  F1^ F1  F2^ F2 ), (16.0, F0^ F0  F1^ F1  F2^ F2  F3^ F3 )
(1.0, ), (-2.0, F2^ F2 ), (-2.0, F0^ F0 ), (4.0, F0^ F0  F2^ F2 )
```

FermionSpace objects without point group symmetry information will generate symmetry operators corresponding to only electron number parity and spin parity.

Warning: Symmetry operators generated in this way may exponentially blow up in the number of terms. For advanced usage, the parameter `return_factorized` can be set to True to return operators as a *inquanto.operators.SymmetryOperatorFermionicFactorised* (a subclass of *FermionOperatorList*), giving a product of component operators.

Conversely, *QubitSpace* objects in InQuanto do not store any information with regards to the physical symmetries of the system. Here, Z2 symmetries can be determined by providing an operator which preserves the relevant symmetries – typically the Hamiltonian. Symmetries are found by finding a set of independent generators for operators that commute with the provided operator.

```
from inquanto.spaces import QubitSpace
from inquanto.express import load_h5
h2_sto3g = load_h5("h2_sto3g.h5", as_tuple=True)
h2_hamiltonian = h2_sto3g.hamiltonian_operator.to_FermionOperator().qubit_encode()
qubit_space = QubitSpace(4)
symmetry_operators_qubit = qubit_space.symmetry_operators_z2(h2_hamiltonian)
for x in symmetry_operators_qubit:
    print(x)
```

```
(1.0, I0 I1 Z2 Z3)
(1.0, Z0 I1 I2 Z3)
(1.0, I0 Z1 I2 Z3)
```

11.2 Point Group Symmetry

In quantum chemistry, it is common to explicitly consider molecular point group symmetries. While the *symmetry_operators_z2()* method of a *FermionSpace* forms a convenient way to generate Z2 symmetries (including Z2 point group symmetries), InQuanto contains tools for the explicit analysis of the molecular point group. The spatial symmetries of molecules and their molecular orbitals can be manipulated in using the *inquanto.symmetry.PointGroup* object. A list of supported point groups can be obtained:

```
from inquanto.symmetry import PointGroup
print(PointGroup.supported_groups())
```

```
['C1', 'C2', 'C2h', 'C2v', 'C3v', 'Ci', 'Cs', 'D2', 'D2h', 'D3h', 'Oh', 'Td']
```

If the point group of interest is supported, the corresponding character table can be printed in a human-readable format:

```
from inquanto.symmetry import PointGroup
pg = PointGroup("C2v")
pg.print_character_table()
```

```
       C2v          E     C2 (z)   σ_v (xz)   σ_v (yz)
        A1          1          1          1          1
        A2          1          1         -1         -1
        B1          1         -1          1         -1
        B2          1         -1         -1          1
```

Components of representations expressible by the group can also be evaluated:

```
from inquanto.symmetry import PointGroup
pg = PointGroup("C2v")
print("[1, -1, -1, 1] components:", pg.compute_representation_components([1, -1, -1,␣
↪1]))
```

```
[1, -1, -1, 1] components: [(0, 'A1'), (0, 'A2'), (0, 'B1'), (1, 'B2')]
```

Similarly, the results of direct products of irreps can be obtained, both as a list of characters and as a decomposition into irreducible components.

```
from inquanto.symmetry import  PointGroup
pg = PointGroup("D2h")
components, direct_product = pg.irrep_direct_product(["Ag", "B1u"])
print("Ag, B1u direct product:", direct_product)
print("Ag, B1u direct product components:", components)
```

```
Ag, B1u direct product: [ 1  1 -1 -1 -1 -1  1  1]
Ag, B1u direct product components: [(0, 'Ag'), (0, 'B1g'), (0, 'B2g'), (0, 'B3g'), (0,
↪ 'Au'), (1, 'B1u'), (0, 'B2u'), (0, 'B3u')]
```

11.3 Z2 Tapering

Z2 qubit tapering [22] is a method of reducing the number of qubits needed to simulate a given fermionic Hamiltonian. Each independent Z2 symmetry divides the full 2^N dimensional qubit state space into two 2^{N-1} dimensional sectors - one where the expectation value of the symmetry operator is 1, and one where it is -1. As the symmetry operator commutes with the Hamiltonian, each eigenstate of the Hamiltonian must have support exclusively on states within one of these sectors. By transforming the Hamiltonian such that the symmetry operators are mapped to single qubit operators (note that this is a Clifford operation and as such can be performed classically efficiently), the state of N_s qubits is fixed and can be inferred from the symmetry sector that the desired eigenstate is in. In ground state problems, this symmetry sector is known a priori – it will be the symmetry sector of the Hartree-Fock state, assuming that the true eigenstate has nonzero overlap with the Hartree-Fock state. As the state of these N_s qubits is fixed and known, there is no need to include them in the quantum calculation.

Z2 qubit tapering is implemented in InQuanto by the *inquanto.symmetry.TapererZ2* class. To taper an operator, symmetry operators and the symmetry sectors of a known reference state must be provided. Symmetry operators may be obtained from either the fermionic or qubit space; however, in the former case the operators must be mapped to qubit operators using the same fermion-to-qubit encoding scheme as the qubit operator. Once the *TapererZ2* object is instantiated for a given set of symmetry operators and sectors, it may be used to find tapered operators and states as below. The object may be reused, if tapering multiple operators with the same symmetries is required.

```
from inquanto.symmetry import TapererZ2
hf_state = QubitState([1,1,0,0])
symmetry_sectors = [x.symmetry_sector(hf_state) for x in symmetry_operators_qubit]

taperer = TapererZ2(symmetry_operators_qubit, symmetry_sectors)
tapered_hamiltonian = taperer.tapered_operator(h2_hamiltonian)
print(tapered_hamiltonian)
```

```
(-0.29264467227722535, I3), (-0.8248612119271057, Z3), (0.17966867956301558, X3)
```

Note that if using a tapered Hamiltonian in a variational algorithm, the ansatz state must also be tapered. This is performed using additional functionality of the ansatz classes.

```
from inquanto.ansatzes import FermionSpaceAnsatzUCCSD
from inquanto.states import FermionState
reference_state = FermionState([1,1,0,0])
ansatz = FermionSpaceAnsatzUCCSD(
fermion_space,
reference_state,
taperer=taperer,
tapering_exponent_check_behaviour="discard",
)
```

Tip: For this to work, ansatz excitations must preserve symmetry. By default, *FermionSpaceAnsatzUCCSD* will check for symmetry violating excitations and throw an error if any are found. This behaviour can be changed using the `tapering_exponent_check_behaviour` parameter, which is described in the API reference documentation for *FermionSpaceAnsatzUCCSD*.

CHAPTER

TWELVE

GEOMETRY

The first step of any quantum chemistry calculation is specification of the system geometry. The InQuanto *Geometry* objects exist to standardise geometry formats and provide convenient functionality for building and manipulating molecular and periodic structures.

12.1 Initializing Structures

Geometry objects may be constructed in several ways. Atom-by-atom construction is possible with the *add_atom()* method:

```
from inquanto.geometries import GeometryMolecular

g = GeometryMolecular()
g.add_atom("H", [0, 0, 0])
g.add_atom("H", [0, 0, 0.735])
g.add_atom("H", [0, 0, 1.])
print(g.df)
```

```
   element  x  y      z
id
0        H  0  0  0.000
1        H  0  0  0.735
2        H  0  0  1.000
```

where the `df` attribute is a pandas dataframe of the geometric structure. If the atom position is left empty in *add_atom()*, random coordinates between ±1 Angstroms are generated:

```
g = GeometryMolecular()
g.add_atom("H")
print(g.df)
```

```
   element         x         y         z
id
0        H  0.816777  0.578812 -0.754763
```

Note: InQuanto *Geometry* objects support Angstrom (default) and Bohr as distance units, specified by the `distance_units` constructor argument. If another unit is used, the user can convert the geometry to Bohrs or Angstroms using the *rescale_position_vectors()* class method.

Alternatively, one may instantiate a *GeometryMolecular* object from z-matrices or Cartesian coordinates provided they are appropriately formatted. Examples for both are given below.

```
water_zmatrix = """h
o 1 1.0
h 2 1.0 1 104.5
"""

g = GeometryMolecular(water_zmatrix)
print("Geometry from z-matrix:\n", g.df)

water_xyz = [
     ["O", [0.0000000, 0.0000000, 0.1271610]],
     ["H", [0.0000000, 0.7580820, -0.5086420]],
     ["H", [0.0000000, -0.7580820, -0.5086420]],
]
g = GeometryMolecular(water_xyz)
print("\nGeometry from xyz:\n", g.df)
```

```
Geometry from z-matrix:
    element    x         y        z
id
0         h  0.0  0.000000  0.00000
1         o  0.0  0.000000  1.00000
2         h  0.0  0.968148  1.25038

Geometry from xyz:
    element    x         y         z atom
id
0         O  0.0  0.000000  0.127161   O1
1         H  0.0  0.758082 -0.508642   H2
2         H  0.0 -0.758082 -0.508642   H3
```

It is also possible to instantiate/write from files for a limited number of formats without the use of any extensions. These include:

- Standard format .XYZ files using the *load_xyz()* and *save_xyz()* methods.
- .csv files generated by InQuanto *Geometry* objects using the *load_csv()* and *save_csv()* methods.
- .json files generated by InQuanto *Geometry* objects using the *load_json()* and *save_json()* methods.
- Z-matrix files in the Gaussian format using the *load_zmatrix()* and *save_zmatrix()* methods.

12.2 Calculating and modifying properties

A variety of geometrical quantities, including bond lengths, angles and dihedrals, can be calculated with *Geometry* class methods:

```
ethane_xyz = [
     ["C", [0.0000000, 0.0000000, 0.7688350]],
     ["C", [0.0000000, 0.0000000, -0.7688350]],
     ["H", [0.0000000, 1.0157000, 1.1533240]],
     ["H", [-0.8796220, -0.5078500, 1.1533240]],
     ["H", [0.8796220, -0.5078500, 1.1533240]],
     ["H", [0.0000000, -1.0157000, -1.1533240]],
```

(continues on next page)

(continued from previous page)

```
    ["H", [-0.8796220, 0.5078500, -1.1533240]],
    ["H", [0.8796220, 0.5078500, -1.1533240]],
]
g = GeometryMolecular(ethane_xyz)
print("C=C bond length:", g.bond_length([0, 1]))
print("H-C-C bond angle:", g.bond_angle([2, 0, 1]))
print("2-0-1-5 dihedral angle:", g.dihedral_angle([2, 0, 1, 5]))

g = GeometryMolecular(water_xyz)
print("\n water distance matrix:\n", g.compute_distance_matrix())
```

```
C=C bond length: 1.53767
H-C-C bond angle: 110.73395111184878
2-0-1-5 dihedral angle: 180.0

 water distance matrix:
 [[0.         0.98941082 0.98941082]
 [0.98941082 0.         1.516164  ]
 [0.98941082 1.516164   0.        ]]
```

These quantities can also be modified:

```
g = GeometryMolecular(water_xyz)
print("Equilibrium structure 0-1 bond length:", g.bond_length([0, 1]))
print("Equilibrium structure 0-1 bond angle:", g.bond_angle([1, 0, 2]))

# modifying a bond length
g.modify_bond_length(atom_ids=[0, 1], new_bond_length=2)
print("Stretched structure 0-1 bond length:", g.bond_length([0, 1]))

# modifying a bond angle
g.modify_bond_angle(atom_ids=[1, 0, 2], theta=90, units="deg")
print("Closed bond angle 2-0-2:", g.bond_angle([1, 0, 2]))
```

```
Equilibrium structure 0-1 bond length: 0.989410821414947
Equilibrium structure 0-1 bond angle: 100.0269116572392
Stretched structure 0-1 bond length: 2.0
Closed bond angle 2-0-2: 90.0
```

Note: Only the last specified atom in the `atom_ids` has its position changed when modifying geometric properties as above. The position of all other atoms remains unchanged.

To transform the geometry by moving more than one atom at a time, one may define a grouping scheme within a structure. A chemically relevant example is switching between the staggered and eclipsed geometries of ethane. Below we define a `"methyl"` grouping scheme which contains two sub-groups:

```
g = GeometryMolecular(ethane_xyz)
g.set_groups("methyl", {"ch3_1": [0, 2, 3, 4], "ch3_2": [1, 5, 6, 7]})
print(g.df)
```

```
   element         x        y         z atom methyl
id
0        C  0.000000  0.00000  0.768835   C1  ch3_1
```

(continues on next page)

(continued from previous page)

```
1         C  0.000000  0.00000 -0.768835   C2  ch3_2
2         H  0.000000  1.01570  1.153324   H3  ch3_1
3         H -0.879622 -0.50785  1.153324   H4  ch3_1
4         H  0.879622 -0.50785  1.153324   H5  ch3_1
5         H  0.000000 -1.01570 -1.153324   H6  ch3_2
6         H -0.879622  0.50785 -1.153324   H7  ch3_2
7         H  0.879622  0.50785 -1.153324   H8  ch3_2
```

Transformations may then be performed "by group", so that all atoms within a sub-group are modified simultaneously, preserving internal structure. For example, below we stretch the C-C bond, then rotate the atoms in `"ch3_2"` such that the ethane is in an eclipsed configuration:

```
g.modify_bond_length_by_group(atom_ids=[0, 1], bond_length=1.8, group="methyl")
g.modify_dihedral_angle_by_group(atom_ids=[2, 0, 1, 7], theta=0.0, group="methyl")
print(g.df)
```

```
   element             x        y         z atom methyl
id
0         C  0.000000e+00  0.00000  0.768835   C1  ch3_1
1         C  0.000000e+00  0.00000 -1.031165   C2  ch3_2
2         H  0.000000e+00  1.01570  1.153324   H3  ch3_1
3         H -8.796220e-01 -0.50785  1.153324   H4  ch3_1
4         H  8.796220e-01 -0.50785  1.153324   H5  ch3_1
5         H  8.796220e-01 -0.50785 -1.415654   H6  ch3_2
6         H -8.796220e-01 -0.50785 -1.415654   H7  ch3_2
7         H  2.372829e-16  1.01570 -1.415654   H8  ch3_2
```

Similarly to single atom modifications, the sub-group that gets moved when transformations are applied is that which contains the last specified atom in `atom_ids`.

Convenience methods for generating a series of structures with varying geometric properties are also available. For example, below we generate a range of *GeometryMolecular* objects for water with different H-O-H bond angles:

```
water_geom = GeometryMolecular(water_xyz)
geometries = water_geom.scan_bond_angle([1, 0, 2], [100, 101, 103, 104, 105, 106, 107,
↪ 108])
print([g.bond_angle([1,0,2]) for g in geometries])
```

```
[100.00000000000003, 101.0, 103.0, 103.99999999999997, 105.0, 106.0, 107.0, 108.0]
```

Similar functions are implemented for scanning over bond lengths and dihedrals (*scan_bond_length()* and *scan_dihedral_angle()*), and equivalently for varying properties by group (*scan_bond_angle_by_group()* for example).

12.3 Periodic systems

For periodic systems there exists the *GeometryPeriodic* class, which holds unit cell lattice vectors in addition to atomic positions:

```
from inquanto.geometries import GeometryPeriodic
import numpy as np

# Diamond fcc
```

(continues on next page)

(continued from previous page)

```
d = 3.576

a, b, c = [
   np.array([0, d/2, d/2]),
   np.array([d/2, 0, d/2]),
   np.array([d/2, d/2, 0])
]
atoms = [
   ["C", [0, 0, 0]],
   ["C", a/4 + b/4 + c/4]
]

cfcc_geom = GeometryPeriodic(geometry=atoms, unit_cell=[a, b, c])
print("Diamond fcc lattice vectors:\n", cfcc_geom.unit_cell)
print("\nCartesian atomic positions:\n", cfcc_geom.df)
```

```
Diamond fcc lattice vectors:
 [[0.    1.788 1.788]
 [1.788 0.    1.788]
 [1.788 1.788 0.   ]]

Cartesian atomic positions:
    element      x      y      z atom
id
0         C  0.000  0.000  0.000   C1
1         C  0.894  0.894  0.894   C2
```

GeometryPeriodic supports all of the methods discussed above, excluding instantiation by z-matrices. In addition, one may easily construct supercells with the *build_supercell()* method, which edits the geometry in-place:

```
cfcc_geom.build_supercell(dimensions=[2, 2, 1])
print("Supercell atomic positions:\n", cfcc_geom.df)
```

```
Supercell atomic positions:
    element      x      y      z atom
id
0         C  0.000  0.000  0.000   C1
1         C  0.894  0.894  0.894   C2
2         C  1.788  0.000  1.788   C3
3         C  2.682  0.894  2.682   C4
4         C  0.000  1.788  1.788   C5
5         C  0.894  2.682  2.682   C6
6         C  1.788  1.788  3.576   C7
7         C  2.682  2.682  4.470   C8
```

To visualize both molecular and periodic structures, see the *inquanto-nglview* extension.

CHAPTER

THIRTEEN

CLASSICAL MINIMIZERS

In the near-term many quantum computational chemistry algorithms use a combination of classical and quantum computational resources. Typically, hybrid quantum-classical approaches to finding Hamiltonian eigenvalues and eigenstates involve the minimization of a cost function, which is often the energy. The variables of the cost function are updated on a classical device, and the value of the cost function at each iteration is evaluated on a quantum device. To facilitate algorithms of this type, a variety of minimization methods are available in the InQuanto package. Each of the *minimizers* contains a *minimize()* method, which can be called by the user or by `algorithms` objects as part of the workflow. In this section, we will implement a bespoke state-vector VQE routine with gradients to showcase a few of the available minimizers.

13.1 MinimizerScipy

The simplest minimizer option available in InQuanto is the InQuanto *MinimizerScipy* class, which wraps the SciPy suite of minimizer classes into an InQuanto object. Below, we implement and optimize a VQE objective function using the conjugate gradient method to demonstrate the functionality of the minimizer classes, and the additional control that familiarity with these objects can provide. First, we load in a Hamiltonian from the *express module*:

```
from inquanto.express import load_h5
from inquanto.spaces import FermionSpace
from inquanto.mappings import QubitMappingJordanWigner

h2_sto3g = load_h5("h2_sto3g.h5")
hamiltonian = h2_sto3g["hamiltonian_operator"]
space = FermionSpace(4)
state = space.generate_occupation_state_from_list([1, 1, 0, 0])
qubit_hamiltonian = QubitMappingJordanWigner().operator_map(hamiltonian)
```

We now choose an ansatz (UCCSD), and prepare functions to compute the energy and energy gradient, which will be the VQE objective and VQE gradient functions respectively:

```
from inquanto.ansatzes import FermionSpaceAnsatzUCCSD
from inquanto.computables import ExpectationValue
from inquanto.protocols import ProtocolStateVectorSparse
from pytket.extensions.qiskit import AerStateBackend
import numpy as np

ansatz = FermionSpaceAnsatzUCCSD(space, state)
parameters = ansatz.state_symbols.construct_zeros()

ev = ExpectationValue(kernel=qubit_hamiltonian, state=ansatz)
ev.build([ProtocolStateVectorSparse()])
```

(continues on next page)

(continued from previous page)

```
symbols = ansatz.state_symbols

evg = ev.gradient_real()
evg.build([ProtocolStateVectorSparse()])
backend = AerStateBackend()

def vqe_objective(variables):
    parameters = ansatz.state_symbols.construct_from_array(variables)
    ev.run(backend, parameters)
    return ev.evaluate().real

def vqe_gradient(variables):
    parameters = ansatz.state_symbols.construct_from_array(variables)
    evg.run(backend, parameters)
    return evg.evaluate()

print("VQE energy with parameters at [0, 0, 0]:", vqe_objective(np.zeros(3)))
print("Gradients of parameters at [0, 0, 0]:", vqe_gradient(np.zeros(3)))
```

```
VQE energy with parameters at [0, 0, 0]: -1.1175058842043315
Gradients of parameters at [0, 0, 0]: [0.         0.         0.35933736]
```

It should be noted that the minimizers work with NumPy arrays, not InQuanto *SymbolDict* or *SymbolSet* parameter objects. The parameter objects compatible with the *computables* objects can be constructed with the *construct_from_array()* method as shown above.

We now initialize and execute the conjugate gradient minimizer. The underlying SciPy object can be configured by passing an options `dict` to the *MinimizerScipy* constructor, according to the solver-specific guidance in the SciPy user manual. With this, one may define, for example, a maximum number of iterations. With the *minimize()* method, we can leave the `gradient` argument empty to compute the gradient numerically, or pass the gradient function we defined above:

```
from inquanto.minimizers import MinimizerScipy

minimizer = MinimizerScipy("CG", disp=True)
minimizer.minimize(function=vqe_objective, initial=np.zeros(3))
```

```
Optimization terminated successfully.
         Current function value: -1.136847
         Iterations: 2
         Function evaluations: 20
         Gradient evaluations: 5
```

```
(-1.1368465754720543, array([ 0.        ,  0.        , -0.10723349]))
```

```
minimizer = MinimizerScipy("CG", disp=True)
min, loc = minimizer.minimize(function=vqe_objective, initial=np.zeros(3),␣
↪gradient=vqe_gradient)
print("Objective function minimum is {}, located at {}".format(min, loc))
```

```
Optimization terminated successfully.
         Current function value: -1.136847
         Iterations: 2
         Function evaluations: 5
         Gradient evaluations: 5
```

(continues on next page)

(continued from previous page)

```
Objective function minimum is -1.1368465754720547, located at [ 0.          0.          -
↪0.1072335]
```

As one might expect, we observe that the optimizer converges to the same value in both cases, but requires less evaluations of the objective function when gradient information is provided.

13.2 MinimizerRotosolve

The Rotosolve minimizer [23], is a gradient-free optimizer designed for minimization of VQE-like objective functions, and is available in the *MinimizerRotosolve* class. With this minimizer, one may define a maximum number of iterations and convergence threshold on initialization.

A short example using rotosolve with an algorithm object is shown below.

```
from inquanto.algorithms import AlgorithmVQE
from inquanto.minimizers import MinimizerRotosolve

minimizer=MinimizerRotosolve(max_iterations=10, tolerance=1e-6, disp=True)

vqe = AlgorithmVQE(
    objective_expression=ev,
    minimizer=minimizer,
    initial_parameters=ansatz.state_symbols.construct_zeros()
)
vqe.build(backend=backend, protocol_expression=[ProtocolStateVectorSparse()])
vqe.run()
print("VQE Energy:", vqe.generate_report()["final_value"])
```

```
# TIMER BLOCK-0 BEGINS AT 2023-05-02 11:28:16.084203

ROTOSOLVER - A gradient-free optimizer for parametric circuits
```

```
Iteration 1
fun = -1.1368465754720547          variance = 0.011499023526666175          p-norm =␣
↪0.1072335000205914
```

```
Iteration 2
fun = -1.1368465754720547          variance = 4.930380657631324e-32          p-norm =␣
↪0.10723350002059162

Optimizer Converged
nit = 2          nfun = 19
final fun = -1.1368465754720547          final variance = 4.930380657631324e-32␣
↪          final p-norm = 0.10723350002059162

# TIMER BLOCK-0 ENDS - DURATION (s):  0.4214757 [0:00:00.421476]
VQE Energy: -1.1368465754720547
```

13.3 MinimizerSGD

The Stochastic Gradient Descent (SGD) approach to functional optimization is available in the *MinimizerSGD* class. This minimizer takes bespoke input arguments for the `learning_rate` and `decay_rate` parameters, defined in [24].

A short example using *MinimizerSGD* is shown below.

```
from inquanto.minimizers import MinimizerSGD

minimizer=MinimizerSGD(learning_rate=0.25, decay_rate=0.5, max_iterations=10,␣
↪disp=True)

vqe = AlgorithmVQE(
    objective_expression=ev,
    minimizer=minimizer,
    initial_parameters=ansatz.state_symbols.construct_zeros(),
    gradient_expression=evg
)
vqe.build(backend=backend, protocol_expression=[ProtocolStateVectorSparse()])
vqe.run()
print("VQE Energy:", vqe.generate_report()["final_value"])
```

```
# TIMER BLOCK-1 BEGINS AT 2023-05-02 11:28:16.527475

Optimizer Stochastic Gradient Descent

Iteration 0

                fun = -1.1175058842043315
                p-norm = 0.0
                g-norm = 0.3593373591260314
```

```
Iteration 1

                fun = -1.1321535146012713
                p-norm = 0.05448728137252682
                g-norm = 0.17778365523875406

Iteration 2

                fun = -1.1357239648371478
                p-norm = 0.08144509079704804
                g-norm = 0.08704389270781043
```

```
Iteration 3

                fun = -1.136578976183476
                p-norm = 0.09464378821405402
                g-norm = 0.04250854257884021

Iteration 4

                fun = -1.1367828405791487
```

(continues on next page)

(continued from previous page)

```
               p-norm = 0.10108947180749564
               g-norm = 0.02074667912184902
```

```
Iteration 5

               fun = -1.1368313985785883
               p-norm = 0.10423534605115099
               g-norm = 0.010124128478002314
```

```
Iteration 6

               fun = -1.1368429616408466
               p-norm = 0.10577049463234554
               g-norm = 0.004940280683154885

Iteration 7

               fun = -1.1368457149778977
               p-norm = 0.10651960255782542
               g-norm = 0.002410693567937411
```

```
Iteration 8

               fun = -1.1368463705791985
               p-norm = 0.10688514244785693
               g-norm = 0.0011763364084180494

Iteration 9

               fun = -1.1368465266849068
               p-norm = 0.10706351347231738
               g-norm = 0.0005740118592306442

Optimizer Converged
nit = 9          nfun = 9          njac = 9

            final fun = -1.1368465266849068
            final p-norm = 0.10715055242023289
            final g-norm = 0.0005740118592306442

# TIMER BLOCK-1 ENDS - DURATION (s):  0.5100899 [0:00:00.510090]
VQE Energy: -1.1368465266849068
```

CHAPTER

FOURTEEN

EXPRESS

The *express* module is a database of pre-computed, example data sets generated with classical compute methods. These may be used for testing and benchmarking algorithms. A list of the available data sets is given *below*, or can be queried with the *list_h5()* function. Each data set is stored in a structured `.h5` file which can be loaded as a Python `namedtuple` collection as follows:

```
from inquanto.express import load_h5, list_h5
print("Available express files:\n", list_h5())

h2_sto3g_data = load_h5("h2_sto3g.h5", as_tuple=True)
hamiltonian = h2_sto3g_data.hamiltonian_operator

print("\nFile description:\n", h2_sto3g_data.description)
print("\nHamiltonian operator:\n", hamiltonian.to_FermionOperator())
print("\nH2 STO-3G CCSD Energy:\n" + str(h2_sto3g_data.energy_ccsd) +' Ha')
```

```
Available express files:
 ['h2_4_ring_sto3g.h5', 'lih_sto3g.h5', 'h3_sto3g_m2_u.h5', 'h2_4_pbc_sto3g.h5', 'h2_
↪1_ring_sto3g.h5', 'ch2_sto3g_m3_u.h5', 'h2_5_ring_sto3g.h5', 'lih_sto3g_symmetry.h5
↪', 'h2_2_ring_sto3g.h5', 'h2_1_pbc_631g.h5', 'h2_1_ring_631g.h5', 'h2_sto3g.h5',
↪'h2_sto3g_long.h5', 'h2_1_pbc_sto3g.h5', 'h2_3_pbc_sto3g.h5', 'h4_square_sto3g_m3.h5
↪', 'h2_2_pbc_631g.h5', 'h2_sto3g_symmetry.h5', 'h2_2_pbc_sto3g.h5', 'h3_chain_sto3g.
↪h5', 'h2_5_pbc_sto3g.h5', 'heh_sto3g_u.h5', 'h2_631g.h5', 'beh2_sto3g_symmetry.h5',
↪'lih_631g.h5', 'h2_631g_symmetry.h5', 'h2_2_ring_631g.h5', 'hehp_sto3g.h5', 'h2o_
↪sto3g.h5', 'beh2_sto3g.h5', 'ch2_sto3g_m3.h5', 'nh3_sto3g.h5', 'h2o_sto3g_symmetry.
↪h5', 'h2_3_ring_sto3g.h5', 'h3p_chain_sto3g.h5', 'ch4_sto3g_symmetry.h5', 'nh3_
↪sto3g_symmetry.h5', 'ch4_sto3g.h5', 'h5_sto3g_m2.h5', 'h5_sto3g_m2_u.h5', 'h3p_
↪sto3g_c2v.h5', 'h3_sto3g_m2.h5']

File description:
 H2 molecule, calculated with the STO-3G basis set in a spin-restricted formalism.

Hamiltonian operator:
 (0.7430177069924179, ), (-1.2702927243904387, F0^ F0 ), (-0.45680735030940944, F2^␣
↪F2 ), (-1.2702927243904387, F1^ F1 ), (-0.45680735030940944, F3^ F3 ), (0.
↪4889085974504739, F2^ F0^ F0  F2 ), (0.4889085974504739, F3^ F1^ F1  F3 ), (0.
↪680061857584128, F1^ F0^ F0  F1 ), (0.6685772770134896, F2^ F1^ F1  F2 ), (0.
↪1796686795630157, F1^ F0^ F2  F3 ), (-0.1796686795630157, F2^ F1^ F0  F3 ), (-0.
↪1796686795630157, F3^ F0^ F1  F2 ), (0.17966867956301566, F3^ F2^ F0  F1 ), (0.
↪6685772770134896, F3^ F0^ F0  F3 ), (0.7028135332762816, F3^ F2^ F2  F3 )

H2 STO-3G CCSD Energy:
-1.1368465754747643 Ha
```

In addition to the data sets listed below, the express module contains additional tools for easily running simple calculations.

The *run_rhf()* and *run_rohf()* functions allow for quick SCF calculations providing basic data:

```
from inquanto.express import run_rhf
e_total, mo_energy, mo_coeff, rdm1 = run_rhf(h2_sto3g_data.hamiltonian_operator, h2_
↪sto3g_data.n_electron)
print("Hartree-Fock energy: {}".format(e_total))
```

```
Hartree-Fock energy: -1.1175058842043315
```

Similarly, the *run_vqe()* function performs a simple, state-vector VQE calculation, as shown in the *quick-start guide*:

```
from inquanto.express import run_vqe
from inquanto.states import FermionState
from inquanto.spaces import FermionSpace
from inquanto.ansatzes import FermionSpaceAnsatzUCCSD
from pytket.extensions.qiskit import AerStateBackend

backend = AerStateBackend()
state = FermionState([1, 1, 0, 0])
space = FermionSpace(4)
ansatz = FermionSpaceAnsatzUCCSD(fermion_space=space, fermion_state=state)
qubit_hamiltonian = hamiltonian.qubit_encode()
vqe = run_vqe(ansatz, qubit_hamiltonian, backend)
print("VQE Energy: ", round(vqe.final_value, 8))
```

```
# TIMER BLOCK-0 BEGINS AT 2023-05-02 11:28:09.063642
# TIMER BLOCK-0 ENDS - DURATION (s):  0.1457706 [0:00:00.145771]
VQE Energy:  -1.13684658
```

Finally, the *express* module also contains drivers for generating simple Hubbard Hamiltonians, such as the dimer:

```
from inquanto.express import DriverHubbardDimer

hubbard_dimer_driver = DriverHubbardDimer(t=0.2, u=2.0)

dimer_ham, dimer_space, dimer_hf_state = hubbard_dimer_driver.get_system()

print('Dimer Hamiltonian:\n', dimer_ham.normal_ordered().compress())
```

```
Dimer Hamiltonian:
 (-0.2, F2^ F0 ), (-0.2, F0^ F2 ), (-0.2, F3^ F1 ), (-0.2, F1^ F3 ), (-2.0, F1^ F0^␣
↪F1  F0 ), (-2.0, F3^ F2^ F3  F2 )
```

As well as the Hamiltonians for chain (finite number of sites with end sites not connected) and ring (finite number of sites with end sites connected) topologies:

```
from inquanto.express import DriverGeneralizedHubbard

hubbard_ring_driver = DriverGeneralizedHubbard(t=-0.2, u=2.0, n=3, ring=True)
hubbard_chain_driver = DriverGeneralizedHubbard(t=-0.2, u=2.0, n=3, ring=False)

ring_ham, ring_space, ring_hf_state = hubbard_ring_driver.get_system()
chain_ham, chain_space, chain_hf_state = hubbard_chain_driver.get_system()

print('\nRing Hamiltonian:\n', ring_ham.normal_ordered().compress())
print('\nChain Hamiltonian:\n', chain_ham.normal_ordered().compress())
```

```
Ring Hamiltonian:
 (-0.2, F0^ F2 ), (-0.2, F0^ F4 ), (-0.2, F2^ F0 ), (-0.2, F2^ F4 ), (-0.2, F4^ F0 ),␣
↪(-0.2, F4^ F2 ), (-0.2, F1^ F3 ), (-0.2, F1^ F5 ), (-0.2, F3^ F1 ), (-0.2, F3^ F5 ),
↪ (-0.2, F5^ F1 ), (-0.2, F5^ F3 ), (-2.0, F1^ F0^ F1  F0 ), (-2.0, F3^ F2^ F3  F2 ),
↪ (-2.0, F5^ F4^ F5  F4 )

Chain Hamiltonian:
 (-0.2, F0^ F2 ), (-0.2, F2^ F0 ), (-0.2, F2^ F4 ), (-0.2, F4^ F2 ), (-0.2, F1^ F3 ),␣
↪(-0.2, F3^ F1 ), (-0.2, F3^ F5 ), (-0.2, F5^ F3 ), (-2.0, F1^ F0^ F1  F0 ), (-2.0,␣
↪F3^ F2^ F3  F2 ), (-2.0, F5^ F4^ F5  F4 )
```

Note the minus sign for the `t` argument for *DriverGeneralizedHubbard* (whereas for *DriverHubbardDimer* `t` is negated when the Hamiltonian is constructed). This reflects the different conventions used in the literature; in some conventions `t` is kept negative, while in general the sign can be absorbed into the coefficients. Hence for a more "general" approach, the sign of the user-provided `t` is not changed in *DriverGeneralizedHubbard*. To generate the same Hubbard dimer Hamiltonian as above with *DriverGeneralizedHubbard*, use:

```
from inquanto.express import DriverGeneralizedHubbard

hubbard_dimer_driver = DriverGeneralizedHubbard(t=-0.2, u=2.0, n=2, ring=False)

dimer_ham, dimer_space, dimer_hf_state = hubbard_dimer_driver.get_system()

print('Dimer Hamiltonian:\n', dimer_ham.normal_ordered().compress())
```

```
Dimer Hamiltonian:
 (-0.2, F0^ F2 ), (-0.2, F2^ F0 ), (-0.2, F1^ F3 ), (-0.2, F3^ F1 ), (-2.0, F1^ F0^␣
↪F1  F0 ), (-2.0, F3^ F2^ F3  F2 )
```

14.1 List of Express files

The full list of chemical data sets included in the *express* module are given below. Files are labelled by the system atoms and structure. Charges are indicated by `p` for + and `n` for - appended to the atom list, and basis sets are labelled (e.g. `sto3g`). A spin multiplicity greater than 1 is labelled as `m#`, where `2` labels a doublet state. Lastly, `u` indicates the unrestricted Hartree-Fock formalism, and no such label always implies restricted Hartree-Fock.

File name	Description
beh2_sto3g.h5	BeH2 molecule, calculated with the STO-3G basis set, with a spin-restricted formalism.
beh2_sto3g_symmetry.h5	BeH2 molecule, calculated with point group symmetries in the STO-3G basis, with a spin-restricted formalism.
ch2_sto3g_m3.h5	CH2 molecule with spin multiplicity 3. Calculated with the STO-3G basis set in a spin-restricted formalism.
ch2_sto3g_m3_u.h5	CH2 molecule with spin multiplicity 3. Calculated with the STO-3G basis set in a spin-unrestricted formalism.
ch4_sto3g.h5	CH4 molecule, calculated with the STO-3G basis set in a spin-restricted formalism.

continues on next page

Table 14.1 – continued from previous page

File name	Description
ch4_sto3g_symmetry.h5	CH4 molecule, calculated with point group symmetries in the STO-3G basis, with a spin-restricted formalism.
h2_1_pbc_631g.h5	H2 molecule in a tetragonal unit cell with periodic boundary consitions. Calculated with the 631G basis set in a spin-restricted formalism.
h2_1_pbc_sto3g.h5	H2 molecule in a tetragonal unit cell with periodic boundary consitions. Calculated with the STO-3G basis set in a spin-restricted formalism.
h2_1_ring_631g.h5	A single H2 molecule, calculated with the 6-31G basis in a spin-restricted formalism.
h2_1_ring_sto3g.h5	A single H2 molecule, calculated with the STO-3G basis in a spin-restricted formalism.
h2_2_pbc_631g.h5	2x1x1 supercell of the periodic H2 system in h2_1_pbc_631g.h5.
h2_2_pbc_sto3g.h5	2x1x1 supercell of the periodic H2 system in h2_1_pbc_sto3g.h5.
h2_2_ring_631g.h5	Two H2 molecules in a ring geometry, calculated with the 6-31G basis in a spin-restricted formalism.
h2_2_ring_sto3g.h5	Two H2 molecules in a ring geometry, calculated with the STO-3G basis in a spin-restricted formalism.
h2_3_pbc_sto3g.h5	3x1x1 supercell of the periodic H2 system in h2_1_pbc_sto3g.h5.
h2_3_ring_sto3g.h5	Three H2 molecules in a ring geometry, calculated with the STO-3G basis in a spin-restricted formalism.
h2_4_pbc_sto3g.h5	4x1x1 supercell of the periodic H2 system in h2_1_pbc_sto3g.h5.
h2_4_ring_sto3g.h5	Four H2 molecules in a ring geometry, calculated with the STO-3G basis in a spin-restricted formalism.
h2_5_pbc_sto3g.h5	5x1x1 supercell of the periodic H2 system in h2_1_pbc_sto3g.h5.
h2_5_ring_sto3g.h5	Five H2 molecules in a ring geometry, calculated with the STO-3G basis in a spin-restricted formalism.
h2_631g.h5	H2 molecule, calculated with the 6-31G basis set in a spin-restricted formalism.
h2_631g_symmetry.h5	H2 molecule, calculated with point group symmetries in the 6-31G basis, with a spin-restricted formalism.
h2_sto3g.h5	H2 molecule, calculated with the STO-3G basis set in a spin-restricted formalism.
h2_sto3g_long.h5	H2 molecule with a longer bond length compared to h2_sto3g.h5.
h2_sto3g_symmetry.h5	H2 molecule, calculated with point group symmetries in the STO-3G basis, with a spin-restricted formalism.

continues on next page

Table 14.1 – continued from previous page

File name	Description
h2o_sto3g.h5	H2O molecule, calculated with the STO-3G basis set in a spin-restricted formalism.
h2o_sto3g_symmetry.h5	H2O molecule, calculated with point group symmetries in the STO-3G basis, with a spin-restricted formalism.
h3_chain_sto3g.h5	Linear chain of three H atoms, calculated with the STO-3G basis in a spin-restricted formalism.
h3_sto3g_m2.h5	Linear chain of three H atoms with spin multiplicity 2, calculated with the STO-3G basis in a spin-restricted formalism.
h3_sto3g_m2_u.h5	Linear chain of three H atoms with spin multiplicity 2, calculated with the STO-3G basis in a spin-unrestricted formalism.
h3p_chain_sto3g.h5	Linear H3+ chain, calculated with the STO-3G basis in a spin-restricted formalism.
h3p_sto3g_c2v.h5	H3+ chain with C2v symmetry, calculated with point group symmetries in the STO-3G basis, with a spin-restricted formalism.
h4_square_sto3g_m3.h5	Four H atoms in a square geometry with spin multiplicity 3, calculated with the STO-3G basis in a spin-restricted formalism.
h5_sto3g_m2.h5	H5 molecule with spin multiplicity 2. Calculated with the STO-3G basis set in a spin-restricted formalism.
h5_sto3g_m2_u.h5	H5 molecule with spin multiplicity 2. Calculated with the STO-3G basis set in a spin-unrestricted formalism.
heh_sto3g_u.h5	HeH molecule with spin multiplicity 2. Calculated with the STO-3G basis set in a spin-unrestricted formalism.
hehp_sto3g.h5	HeH+ molecule. Calculated with the STO-3G basis set in the spin-restricted formalism.
lih_631g.h5	LiH molecule, calculated with the 6-31G basis in a spin-restricted formalism.
lih_sto3g.h5	LiH molecule, calculated with the STO-3G basis set in a spin-restricted formalism.
lih_sto3g_symmetry.h5	LiH molecule, calculated with point group symmetries in the STO-3G basis, with a spin-restricted formalism.
nh3_sto3g.h5	NH3 molecule, calculated in the STO-3G basis set with a spin-restricted formalism.
nh3_sto3g_symmetry.h5	NH3 molecule, calculated with point group symmetries in the STO-3G basis, with a spin-restricted formalism.

CHAPTER

FIFTEEN

DENSITY MATRIX EMBEDDING THEORY

Density Matrix Embedding Theory (DMET) is a quantum embedding method developed to address the challenges of solving strongly correlated systems in quantum chemistry and condensed matter physics [25, 26]. This technique effectively partitions a large quantum system into smaller fragments and their corresponding environments, which allows for more manageable calculations.

The full DMET implementation involves nested iterations and fitting procedures to obtain a self-consistent one-body density matrix and an effective one-body correlation potential, which accounts for the correlation effects in the fragments. The accurate effective one-body correlation potential is found by solving the fragment problems using high-level classical or quantum methods and by comparing against results obtained through an effective Hamiltonian.

InQuanto supports various versions of DMET, including the full DMET, which utilizes a chemical potential and an arbitrary parametrization of the one-body correlation potential, one-shot DMET, which only optimizes the chemical potential, and single impurity DMET, which divides the total system into a single fragment and its environment.

Typically prior to the embedding algorithm one needs to generate a Hamiltonian with a localized and orthonormal basis or atomic orbitals, in which spatial fragments can be specified. In addition, DMET requires an initial density matrix of the total system, which is usually one-electron reduced density matrix (1-RDM), calculated with a lower level quantum chemical method, in the respective localized basis. In InQuanto this can be computed by an extension chemistry driver. Moreover, traditional chemistry methods can also be used as high level fragment solvers and inquanto-pyscf extension has a number of fragment solvers for the user to combine.

15.1 Impurity DMET

InQuanto's Impurity DMET method is the simplest of its kind because it only deals with one selected fragment of a larger system. The initial 1-RDM in the localized basis is used to find new orbitals (fragment orbitals) such that the environment block of the 1-RDM in the new orbitals becomes diagonal. The Schmidt decomposition and DMET guarantee that no more diagonal elements will be fractional in the environment block than the size of the fragment block. Those orbitals associated with fractional diagonals in the environment block are called the bath orbitals, and the remaining orbitals, which are associated with integer diagonal elements (i.e., fully occupied or unoccupied orbitals), are called environment orbitals.

In the new orbitals, the active space is chosen as the orbitals for the fragment and the bath part, and the fragment calculation is reduced to the active space Hamiltonian, which without the constant terms takes the form of:

$$\hat{H}_{\text{emb}} = \sum_{pq \in A \cup B} (h_{pq} + \sum_{rs \in E} ((pq|rs) - (ps|rq))\Gamma^{\text{env}}_{rs})\hat{a}^\dagger_p \hat{a}_q + \frac{1}{2} \sum_{pqrs \in A \cup B} (pq|rs)\hat{a}^\dagger_p \hat{a}^\dagger_r \hat{a}_s \hat{a}_q \tag{15.1}$$

where A is the set of orbitals on the fragment, B is the set of bath orbitals, $E = \overline{A \cup B}$ is the set of environment orbitals, $\hat{a}^\dagger$ and $\hat{a}$ are creation and annihilation operators in the fragment basis, respectively, and Γ^{env}_{rs} is the 1-RDM constructed from the fully occupied environment orbitals only.

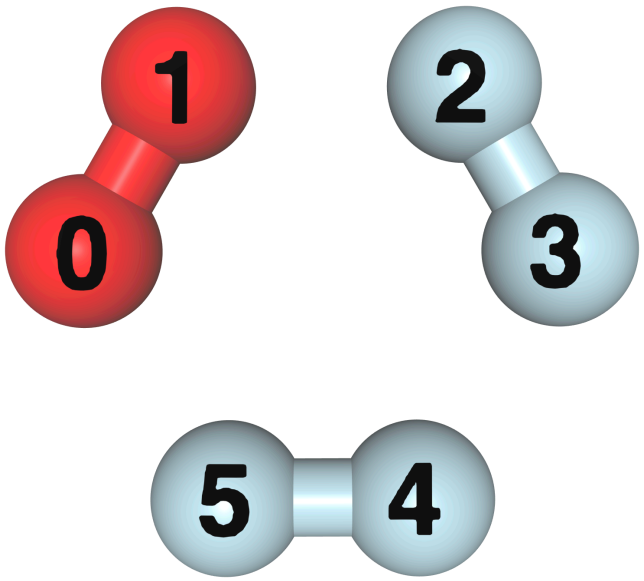


Fig. 15.1: Three H_2 molecules in a ring arrangement, the H_2 molecule in red color is selected as the fragment.

To demonstrate a simple calculation, the following example will load the necessary inputs for a 6-atom H ring from the `express` module. The 6 atoms are arranged as 3 pairs of H_2 molecules forming a hexagon, with the atoms indexed as shown in Fig. 15.1. The Hamiltonian and the Hartree-Fock (HF) 1-RDM were computed with the STO3G basis set and with the Löwdin transformation, resulting in 6 spatial orbitals, one for each hydrogen atom. The Löwdin transformation ensures that the spatial orbitals are localized and orthonormal, making them suitable for DMET calculations. After acquiring the input data, the Impurity DMET method can be initialized:

```
from inquanto.express import load_h5
from inquanto.embeddings import ImpurityDMETROHF

h2_3_ring_sto3g = load_h5("h2_3_ring_sto3g.h5", as_tuple=True)

dmet = ImpurityDMETROHF(
    h2_3_ring_sto3g.hamiltonian_operator_lowdin, h2_3_ring_sto3g.one_body_rdm_hf_
↪lowdin
)
```

The Impurity DMET method class *`ImpurityDMETROHF`* is initialized with the Hamiltonian operator and the 1-RDM in the localized basis. The name of the class indicates that this method assumes the Hamiltonian operator and the 1-RDM are spin-restricted.

In this specific example, we select the first H_2 molecule as a fragment, which consists of the first two consecutive H atoms in the ring. The fragment and its bath will be treated with a high-level solver. In InQuanto, the fragment is defined by a spatial orbital `mask` and a fragment solver as follows:

```
from numpy import array
mask = array([True, True, False, False, False, False])
```

(continues on next page)

(continued from previous page)

```
from inquanto.embeddings import ImpurityDMETROHFFragmentED
fragment = ImpurityDMETROHFFragmentED(dmet, mask)
```

The `mask` is a boolean array with a length equal to the number of localized spatial orbitals. The `True` values in the mask select the indices of the localized spatial orbitals that belong to the fragment. The elaborate classname *ImpurityDMETROHFFragmentED* defines a high-level solver for the fragment, which is responsible for calculating the ground state of the fragment+bath system during the DMET procedure. In this specific example, the solver employs an exact diagonalisation method.

To perform the DMET calculation, call the `run` method:

```
result = dmet.run(fragment)
```

The advantage of this simple single-fragment version of DMET is its variational nature, with no iterative part outside the solvers, and this makes it more suitable for present noisy quantum algorithms.

To include more than one fragment in the DMET method, use the *DMETRHF* class, which is described in the next subsection.

15.2 One-shot DMET

One-shot DMET allows a total system to be decomposed into fragments, and each fragment will have its own bath and will be treated with its own fragment solver. The method introduces a global chemical potential as a self-consistency parameter to ensure that fractional charges on the fragments eventually sum up exactly to the total charge of the system. The Hamiltonian of a fragment plus bath (embedding system), which the fragment solver needs to handle, is

$$\hat{H}_{\text{emb}}(\mu) = \hat{H}_{\text{emb}} - \mu \sum_{p \in A} a_p^\dagger \hat{a}_p \tag{15.2}$$

where μ is the introduced global chemical potential. The value of μ is determined by the global constraint

$$\sum_f \langle \Psi_f(\mu) | \hat{N}_f | \Psi_f(\mu) \rangle = N_{\text{tot}}, \tag{15.3}$$

where $\Psi_f(\mu)$ is the ground state of fragment f, calculated by a high-level method, $\hat{N}_f = \sum_{p \in A_f} a_p^\dagger \hat{a}_p$ is the particle number operator of fragment f and N_{tot} is the total number of electrons in the system. In InQuanto, this constraint is satisfied by employing an iterative solver.

The ground state energy of the total system is calculated from the individual ground state calculations of the fragments based on the democratic mixing of density matrix elements, and to avoid double counting of the environmental terms, the fragment's energy contribution to the total energy is calculated with the fragment energy operator and not the Hamiltonian:

$$\begin{aligned}\hat{E}_{\text{emb}} = \sum_{p \in A} \Bigg(\sum_{q \in A \cup B} \Bigg(h_{pq} + \frac{\sum_{rs \in E} [(pq|rs) - (ps|rq)] \Gamma^{\text{env}}_{rs}}{2} \Bigg) \hat{a}_p^\dagger \hat{a}_q \\ + \frac{1}{2} \sum_{qrs \in A \cup B} (pq|rs) \hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_r \hat{a}_s \Bigg).\end{aligned} \tag{15.4}$$

The total electronic energy is calculated as a sum of the fragment contributions, $\sum_f \langle \Psi_f(\mu) | \hat{E}_{\text{emb},f} | \Psi_f(\mu) \rangle$, where $E_{\text{emb},f}$ denotes the fragment energy operator from the equation above for fragment f, and Ψ_f denotes the high level solution for fragment f.

To demonstrate the one-shot DMET on the 6-atom ring toy system, we need to instantiate the class `DMETRHF` first:

```
from inquanto.embeddings import DMETRHF

h2_3_ring_sto3g = load_h5("h2_3_ring_sto3g.h5", as_tuple=True)

dmet = DMETRHF(
    h2_3_ring_sto3g.hamiltonian_operator_lowdin, h2_3_ring_sto3g.rdm1_hf_lowdin
)
```

The fragments are similarly defined as in the impurity DMET case. In this particular example, we decompose the total system into three H_2 fragments and use a black-box UCCSD VQE statevector fragment solver:

```
mask1 = array([True, True, False, False, False, False])
mask2 = array([False, False, True, True, False, False])
mask3 = array([False, False, False, False, True, True])

from inquanto.embeddings import DMETRHFFragmentUCCSDVQE
from pytket.extensions.qiskit import AerStateBackend

backend=AerStateBackend()
fragment1 = DMETRHFFragmentUCCSDVQE(dmet, mask1, backend=backend)
fragment2 = DMETRHFFragmentUCCSDVQE(dmet, mask2, backend=backend)
fragment3 = DMETRHFFragmentUCCSDVQE(dmet, mask3, backend=backend)

fragments = [fragment1, fragment2, fragment3]
```

It is essential that the set of fragments completely covers the entire system. To run the one-shot DMET procedure, call the `run` method:

```
result = dmet.run(fragments)
```

15.3 Full DMET with correlation matrix

In the full DMET simulation, the effect of a one-body correlation potential from the environment is also added to the embedding Hamiltonian, as follows:

$$\hat{H}_{\text{emb}}(\mu, C) = \hat{H}_{\text{emb}}(\mu) + V_{\text{emb}}(\sum_{p,q\notin A} C_{pq} a_p^\dagger \hat{a}_q). \tag{15.5}$$

By computing the ground state of this Hamiltonian with a high-level method, an accurate 1-RDM, Γ^A_{pq}, is obtained for the fragment. In turn, the full DMET method finds the effective one-body correlation for the fragment, that is, $C_{pq} a_p^\dagger \hat{a}_q$ for $p, q \in A$ such that a mean-field (MF) solution 1-RDM, $\Gamma^{A,\text{MF}}_{pq}$, of

$$\hat{H}_{\text{emb}}(\mu, C)' = \hat{H}_{\text{emb}}(\mu) + V_{\text{emb}}(\sum_{p,q\notin A} C_{pq} a_p^\dagger \hat{a}_q) + \sum_{p,q\in A} C_{pq} a_p^\dagger \hat{a}_q \tag{15.6}$$

minimally differs from Γ^A_{pq}, that is, $|\Gamma^A_{pq} - \Gamma^{A,\text{MF}}_{pq}|$ is minimum. In practice, since the mean-field solution is usually already available for $\hat{H}_{\text{emb}}(\mu, C)$, the correlation potential is added to its effective mean-field version, and this is solved instead of $\hat{H}_{\text{emb}}(\mu, C)'$. DMET performs this procedure for every fragment and, with the updated values of μ and C_{pq}, repeats the procedure again until self-consistency is reached. At this point, the fragment energies are computed to calculate the total electronic energy.

The technical difference from the one-shot DMET is that we also need to input a parametrization for the correlation potential C_{pq}. The `DMETRHF` class can take this parametrized correlation potential matrix, in pattern matrix format, as a constructor parameter. For example, for the 6-atom H ring, if we have two independent parameters in the correlation potential matrix, we prepare a pattern matrix that defines where the two independent parameters are:

```
pattern = numpy.array(
    [
        [None, 0, None, None, None, None],
        [0, None, None, None, None, None],
        [None, None, None, 1, None, None],
        [None, None, 1, None, None, None],
        [None, None, None, None, None, 0],
        [None, None, None, None, 0, None],
    ]
)
```

In this pattern matrix, the `0` and `1` indices refer to the first and second independent parameters. The `None` matrix elements in the pattern indicate that the corresponding values of C_{pq} will be kept zero. A parameter can appear in more than one place in the pattern matrix, which could reflect the symmetry of the system. In this example, there are three 2-atom fragments, which are all equivalent. Therefore, due to symmetry, we could use just a single independent parameter and replace the index 1 with index 0 in the pattern matrix. However, to demonstrate the use of independent parameters, in this example we introduce this symmetry-breaking parametrization; i.e. we choose a parameterization that is not required to follow the system's symmetry. At the end of the DMET run, we expect that the two independent parameters converge to the same value.

To perform the full DMET, one should call:

```
energy, chemical_potential, parameters = dmet.run(fragments, pattern)
```

15.4 Custom fragments

When running a hybrid quantum-classical algorithm as part of the DMET method flow, it is important to have the ability to create custom fragment solver classes. There are various ansatzes and strategies for performing a VQE. In InQuanto, only one pre-implemented VQE fragment solver, *DMETRHFFragmentUCCSDVQE*, is provided. To ensure greater versatility, we offer an easy way to define a custom fragment solver. To do this, one can subclass the *DMETRHFFragmentActive*, which requires implementing the `solve_active(...)` method. This method assumes that the Hamiltonian, Fragment Energy operator and other arguments are only provided for the active space, which can be specified when the fragment solver is constructed. The mandatory return values are the expectation value of the Hamiltonian and the Fragment Energy operator with the ground state (`energy` and `fragment_energy` below), as well as the 1-RDM with the ground state:

```
class MyFragment(DMETRHFFragmentActive):

    def solve_active(
         self,
         hamiltonian_operator: ChemistryRestrictedIntegralOperator,
         fragment_energy_operator: ChemistryRestrictedIntegralOperator,
         fermion_space: FermionSpace,
         fermion_state: FermionState,
    ):

        # ... your VQE solution ...

        return energy, fragment_energy, vqe_rdm1
```

You can find complete executable examples in the examples/embeddings folder. The active space can be specified with the `frozen` argument at instantiation, by listing the HF orbital indices for the embedding system. One should be aware that since the embedding system is composed of the fragment and bath orbitals, and the number of bath orbitals is at most the number of fragment orbitals, the maximum number of orbitals is twice the number of fragment orbitals. In the

6-atom ring example, the H_2 fragment has 2 spatial orbitals, therefore the total number of orbitals in a fragment solver is 4, allowing us to freeze, for example, the lowest and the highest energy HF orbitals with indices `0` and `3`, respectively:

```
fr = MyFragment(dmet, mask, frozen=[0, 3])
```

The fragment solvers for the Impurity DMET are simpler because there is only a single fragment, so it is not necessary to calculate the fragment energy. In this case, it is recommended to subclass the *ImpurityDMETROHFFragmentActive* to make a custom class and implement the `solve_active(...)` method.

```
class MyFragment(ImpurityDMETROHFFragmentActive):

    def solve_active(
        self,
        hamiltonian_operator: ChemistryRestrictedIntegralOperator,
        fermion_space: FermionSpace,
        fermion_state: FermionState,
    ):

        # ... your VQE solution ...

        return energy
```

The mandatory return value is the ground state energy of the embedding system. Again, there are complete running examples in the examples/embedding folder.

15.5 DMET for model systems and other Hamiltonians

InQuanto's DMET embedding methods are designed to be flexible and compatible with various types of Hamiltonians as long as they meet specific requirements:

- The Hamiltonian must be in a localized orthonormal basis.
- The Hamiltonian type should be *ChemistryRestrictedIntegralOperator*.

With these requirements met, you can use DMET-based embedding methods with Hamiltonians from various sources, for example:

- Periodic Systems: You can construct a Hamiltonian for a periodic system and use it with the API, enabling the study of crystalline or other periodic structures.
- Model Hamiltonians: Using the Hubbard driver, you can generate model Hamiltonians that can be used with the DMET-based embedding methods API. This capability allows you to explore simplified systems and gain insights into more complex ones.
- COSMO-generated Hamiltonians: By using the inquanto-pyscf extension, you can generate Hamiltonians with COSMO and use them with the API of DMET-based embedding methods. This feature enables the study of solvent effects and other environmental influences on your system of interest.

CHAPTER

SIXTEEN

TUTORIALS

These tutorial notebooks aim to teach the basics and demonstrate the fundamentals of using InQuanto to simulate quantum chemistry on quantum computers. Approximately they increase in complexity, and are available to download (right-click and 'Save as' on download link) or viewable in browser.

Basic VQE	The basics of running VQE simulations using InQuanto	download
Advanced VQE	Several optimization techniques and examining potential energy surfaces	download
Fragmentation	Tackling larger systems with fragmentation: using Density Matrix Embedding Theory (DMET) in InQuanto	download
NGLView	Visualisation of molecular geometries and orbitals with InQuanto-NGLView	download
VQD	Finding electronic excited state energies with Variational Quantum Deflation in InQuanto	download

InQuanto also includes many other *examples* of specific functionality, mostly in the form of scripts that can be adapted.

16.1 Tutorial 1 - A basic VQE simulation with InQuanto

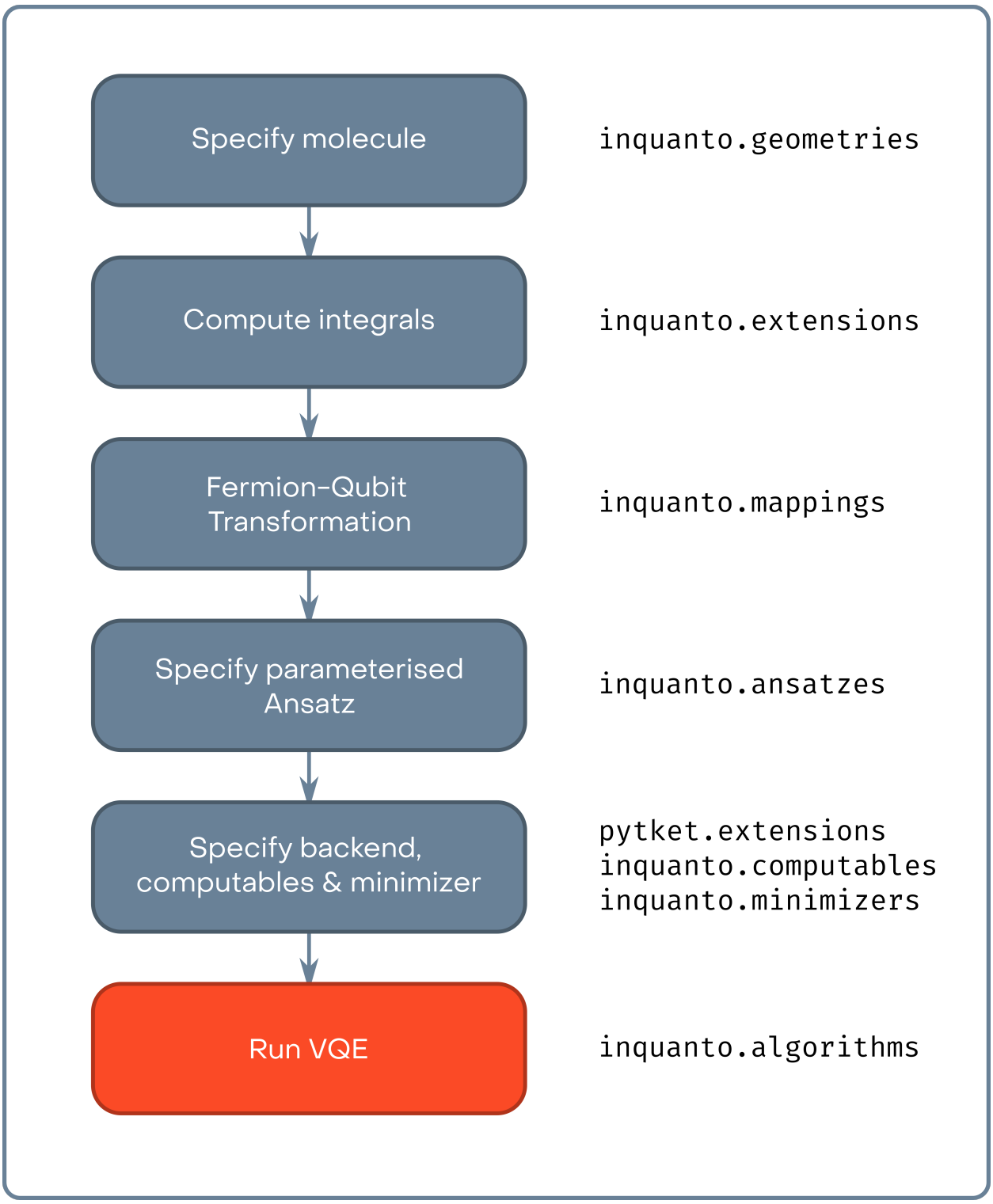
This file will introduce the basic methodology of performing quantum chemistry calculations on quantum computers using InQuanto. We focus here on a practical guide for computation using InQuanto. A discussion of the theory of VQE and a HTML version of this tutorial are available in the InQuanto docs. For this example, we will consider the calculation of the ground state electronic energy of the hydrogen molecule in a single geometric configuration, using a standard VQE methodology.

The goal of any form of electronic structure calculation is to find the eigenvalues and eigenvectors (for the ground state, the lowest eigenvalue/vector) of the second quantized electronic Hamiltonian:

$$\hat{H} = \sum_{i,j=0}^{N} h_{ij} a_i^\dagger a_j + \frac{1}{2} \sum_{i,j,k,l=0}^{N} h_{ijkl} a_i^\dagger a_j^\dagger a_k a_l \tag{16.1}$$

where N is the number of fermionic spin-orbitals, $a_i^\dagger$ and a_i are creation and annihilation operators acting on fermionic spin-orbitals, and h_{ij} and h_{ijkl} are classically precomputable integrals giving the strengths of the one and two electron interactions, respectively.

In the figure below we show the computational steps required in a canonical VQE calculation. The initial five steps here are classical preprocessing steps, whereas the final step may utilize either an actual quantum device, or a classical simulator. In order to demonstrate the simulation process, we subdivide the overall algorithm here to demonstrate the various components of InQuanto.



In the cell below we start by specifying the molecular system, which here is H_2 in the minimal basis set - STO-3G.

[ ]:
```
# ########################
# MOLECULE SPECIFICATION #
# ########################
```

(continues on next page)

(continued from previous page)

```
basis = "sto-3g"
geometry = [["H", [0, 0, 0]], ["H", [0, 0, 0.7122]]]
charge = 0
```

As is the case of classical computational chemistry we specify the atomic orbital basis, the molecular geometry, and the system charge. In InQuanto these are obtained from a `driver`, which runs classical electronic structure calculation to determine the reference molecular spin-orbitals and the h_{ij} and h_{ijkl} integrals in the electronic Hamiltonian (eq 1.). For example a user could run a Hartree-Fock calculation using Psi4, generate an FCIDump file, and use that to instantiate the system. However, inquanto.extensions streamlines this process.

Here we will utilize inquanto-pyscf which is an interface to the PySCF code. This code will steer PySCF calculations and collect the results necessary to build InQuanto objects (fermion operator, state, space). The choice of extension/driver will determine the availability of methods, basis sets, etc. Don't worry if you don't have inquanto.extensions or PySCF - InQuanto provides some data for small test systems.

In the cell below we use a restricted Hartree-Fock calculation to build our system, running a PySCF calculation using the inquanto.extensions.pyscf.ChemistryDriverPySCFMolecularRHF driver. Some useful parameters to the PySCF drivers are `frozen` (to specify frozen spatial atomic orbitals) and `point_group_symmetry` (to enable the use of point group symmetry to reduce computational cost).

[ ]:
```
# ########################## #
# PRELIMINARY CALCULATIONS  #
# ########################## #
from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF

driver = ChemistryDriverPySCFMolecularRHF(basis=basis, geometry=geometry,␣
↪charge=charge)
chemistry_hamiltonian, fock_space, hartree_fock_state = driver.get_system()
hartree_fock_energy = driver.mf_energy

print('HARTREE FOCK ENERGY: {}\n'.format(hartree_fock_energy))
```

```
HARTREE FOCK ENERGY: -1.1175058842043306
```

The .get_system() method uses PySCF to run the restricted Hartree-Fock calculation, returning the integrals in the electronic Hamiltonian as a ChemistryRestrictedIntegralOperator, a fock_space object describing the fermionic Fock space, and the fermionic Hartree-Fock reference state.

In the cell below, we show how the ChemistryRestrictedIntegralOperator, which internally stores the h_{ij} and h_{ijkl} integrals, can be converted to a FermionOperator that represents the electronic Hamiltonian as a sum of terms. These terms can be viewed in a data frame table or printed. The FermionOperator contains a description of both the molecular orbital integrals, and the fermionic creation and annihilation operators. As shown in the snippet above, we can also extract the Hartree-Fock energy from the driver - this gives us an upper bound as to the electronic ground-state energy (no electron correlation).

[ ]:
```
fermionic_hamiltonian = chemistry_hamiltonian.to_FermionOperator()

print('SECOND QUANTIZED HAMILTONIAN PRINTED:\n{}\n'.format(fermionic_hamiltonian))
print('SECOND QUANTIZED HAMILTONIAN AS DATAFRAME:')
fermionic_hamiltonian.df()
```

```
SECOND QUANTIZED HAMILTONIAN PRINTED:
(0.7430177069924179, ), (-1.270292724390438, F0^ F0 ), (-0.45680735030941033, F2^ F2␣
↪), (-1.270292724390438, F1^ F1 ), (-0.45680735030941033, F3^ F3 ), (0.
↪48890859745047327, F2^ F0^ F0  F2 ), (0.48890859745047327, F3^ F1^ F1  F3 ), (0.
```

(continues on next page)

(continued from previous page)

```
↪6800618575841273, F1^ F0^ F0  F1 ), (0.6685772770134888, F2^ F1^ F1  F2 ), (0.
↪1796686795630157, F1^ F0^ F2  F3 ), (-0.17966867956301558, F2^ F1^ F0  F3 ), (-0.
↪17966867956301558, F3^ F0^ F1  F2 ), (0.1796686795630155, F3^ F2^ F0  F1 ), (0.
↪6685772770134888, F3^ F0^ F0  F3 ), (0.7028135332762804, F3^ F2^ F2  F3 )

SECOND QUANTIZED HAMILTONIAN AS DATAFRAME:
```

	Coefficient	Term	Coefficient Type
0	0.743018		<class 'numpy.float64'>
1	-1.270293	F0^ F0	<class 'numpy.float64'>
2	0.680062	F1^ F0^ F0 F1	<class 'numpy.float64'>
3	0.179669	F1^ F0^ F2 F3	<class 'numpy.float64'>
4	-1.270293	F1^ F1	<class 'numpy.float64'>
5	0.488909	F2^ F0^ F0 F2	<class 'numpy.float64'>
6	-0.179669	F2^ F1^ F0 F3	<class 'numpy.float64'>
7	0.668577	F2^ F1^ F1 F2	<class 'numpy.float64'>
8	-0.456807	F2^ F2	<class 'numpy.float64'>
9	0.668577	F3^ F0^ F0 F3	<class 'numpy.float64'>
10	-0.179669	F3^ F0^ F1 F2	<class 'numpy.float64'>
11	0.488909	F3^ F1^ F1 F3	<class 'numpy.float64'>
12	0.179669	F3^ F2^ F0 F1	<class 'numpy.float64'>
13	0.702814	F3^ F2^ F2 F3	<class 'numpy.float64'>
14	-0.456807	F3^ F3	<class 'numpy.float64'>

The `fock_space` and `hartree_fock_state` can be also printed to inspect which orbitals or spin-orbitals are occupied.

[ ]:
```
print('FOCK SPACE AND THE HARTREE-FOCK STATE OCCUPATIONS:')
fock_space.print_state(hartree_fock_state)

#FOCK SPACE AND THE HARTREE-FOCK STATE OCCUPATIONS:
# 0 0a          :  1
# 1 0b          :  1
# 2 1a          :  0
# 3 1b          :  0
```

```
FOCK SPACE AND THE HARTREE-FOCK STATE OCCUPATIONS:
 0 0a          :  1
 1 0b          :  1
 2 1a          :  0
 3 1b          :  0
```

As mentioned above, InQuanto also provides a set of small test systems for where use of a full extension is undesirable. We access these using the inquanto.express module. In the cell below we repeat the above example which used InQuanto-PySCF, but instead of running the Hartree-Fock we load in the precomputed data from express.

[ ]:
```
from inquanto.express import load_h5
from inquanto.spaces import FermionSpace
from inquanto.states import FermionState
h2_sto3g_data = load_h5('h2_sto3g.h5')

integrals = h2_sto3g_data['hamiltonian_operator']
fermionic_hamiltonian = integrals.to_FermionOperator()

hartree_fock_energy = h2_sto3g_data['energy_hf']
num_electrons = h2_sto3g_data['n_electron']
num_spin_orbitals = h2_sto3g_data['n_orbital'] * 2
```

(continues on next page)

(continued from previous page)

```
fock_space = FermionSpace(num_spin_orbitals)
hartree_fock_state = FermionState([1] * num_electrons + [0] * (num_spin_orbitals-num_
↪electrons))

print('SECOND QUANTIZED HAMILTONIAN:\n{}\n'.format(fermionic_hamiltonian))
print('HARTREE FOCK ENERGY: {}\n'.format(hartree_fock_energy))
print('FOCK SPACE AND THE HARTREE-FOCK STATE OCCUPATIONS:')
fock_space.print_state(hartree_fock_state)
```

```
SECOND QUANTIZED HAMILTONIAN:
(0.7430177069924179, ), (-1.2702927243904387, F0^ F0 ), (-0.45680735030940944, F2^ F2␣
↪), (-1.2702927243904387, F1^ F1 ), (-0.45680735030940944, F3^ F3 ), (0.
↪4889085974504739, F2^ F0^ F0  F2 ), (0.4889085974504739, F3^ F1^ F1  F3 ), (0.
↪680061857584128, F1^ F0^ F0  F1 ), (0.6685772770134896, F2^ F1^ F1  F2 ), (0.
↪1796686795630157, F1^ F0^ F2  F3 ), (-0.1796686795630157, F2^ F1^ F0  F3 ), (-0.
↪1796686795630157, F3^ F0^ F1  F2 ), (0.17966867956301566, F3^ F2^ F0  F1 ), (0.
↪6685772770134896, F3^ F0^ F0  F3 ), (0.7028135332762816, F3^ F2^ F2  F3 )

HARTREE FOCK ENERGY: -1.1175058842043315

FOCK SPACE AND THE HARTREE-FOCK STATE OCCUPATIONS:
 0 0a          :  1
 1 0b          :  1
 2 1a          :  0
 3 1b          :  0
```

We now have our data gathered directly from inquanto.express instead of via an external quantum chemistry package. Note that here we needed to manually create `FermionSpace` and `FermionState` objects instead of deriving them from an external driver. The `inquanto.express` also stores the integrals as a `ChemistryRestrictedIntegralOperator` which we need to transform to `FermionOperator`.

Having found the fermionic Hamiltonian, we now must transform it into a form implementable on a quantum computer. As discussed in the documentation, the fermionic creation and annihilation operators must be mapped to strings of Pauli operators. In this example, we use the Jordan-Wigner transformation for this purpose.

[ ]:
```
# ########################## #
# QUBIT MAPPING HAMILTONIAN #
# ########################## #

from inquanto.mappings import QubitMappingJordanWigner

jw_map = QubitMappingJordanWigner()
qubit_hamiltonian = jw_map.operator_map(fermionic_hamiltonian)
print('QUBIT HAMILTONIAN:\n{}'.format(qubit_hamiltonian))
```

```
QUBIT HAMILTONIAN:
(-0.05962058276034621, ), (0.1757594291831965, Z0), (-0.23667117678035654, Z2), (0.
↪1757594291831965, Z1), (-0.23667117678035654, Z3), (0.12222714936261847, Z0 Z2), (0.
↪12222714936261847, Z1 Z3), (0.17001546439603 2, Z0 Z1), (0.1671443192533724, Z1 Z2),␣
↪(0.04491716989075392, Y0 X1 X2 Y3), (-0.04491716989075392, X0 X1 Y2 Y3), (-0.
↪04491716989075392, Y0 Y1 X2 X3), (0.04491716989075392, X0 Y1 Y2 X3), (0.
↪1671443192533724, Z0 Z3), (0.1757033833190704, Z2 Z3)
```

Here, we use the inquanto.mapping.QubitMappingJordanWigner class to perform the mapping, yielding a qubit Hamiltonian as a weighted sum of strings of Pauli X, Y and Z operators acting on qubits. inquanto.mapping contains several fermion-qubit mappings, which all utilise the `.operator_map()` method to map fermionic `FermionOperators` to InQuanto `QubitOperators`. This qubit Hamiltonian will be used to calculate the ground state energy by determining the expectation value of each term in the sum. While the Hamiltonian alone is sufficient for some quantum algorithms (e.g. phase estimation), here we consider a VQE calculation where the preparation of an ansatz state is required.

In the cell below, we use the canonical UCCSD Ansatz – `inquanto.ansatz.FermionSpaceAnsatzUCCSD`. When instantiated, the Ansatz object contains a tket circuit object corresponding to the generation of the parametrized Ansatz state. We can use the `.generate_report()` method of the Ansatz object to give a quick report on some of the quantum resource costs associated with generating the Ansatz – the circuit depth, the overall gate count, the number of Ansatz parameters and the number of qubits required. Further diagnostics can be obtained by examining the tket circuit object itself, `ansatz.state_circuit`. Here we give an example of finding the number of CNOT gates in the circuit. Further information about how to analyze tket circuits can be found in the tket documentation. Two final steps are needed before a VQE simulation can be run - determining how the quantum computer itself will be simulated and setting up the classical optimizer.

[ ]:
```
# ##################### #
# CREATE A UCCSD ANSATZ #
# ##################### #

from inquanto.ansatzes import FermionSpaceAnsatzUCCSD
from pytket import Circuit, OpType

ansatz = FermionSpaceAnsatzUCCSD(fock_space, hartree_fock_state)
print('ANSATZ REPORT:')
print(ansatz.generate_report())
print('\nCNOT GATES:  {}'.format(ansatz.state_circuit.n_gates_of_type(OpType.CX)))
print("\nANSATZ GENERATION CIRCUIT:")
ansatz.state_circuit
```

```
ANSATZ REPORT:
{'ansatz_circuit_depth': 58, 'ansatz_circuit_gates': 110, 'n_parameters': 3, 'n_qubits
↪': 4}

CNOT GATES:  22

ANSATZ GENERATION CIRCUIT:
```

```
[X q[0]; X q[1]; V q[2]; V q[3]; Sdg q[0]; S q[2]; S q[3]; Vdg q[0]; CX q[1], q[2]; H␣
↪q[3]; Sdg q[2]; S q[3]; CX q[2], q[0]; V q[3]; Rz(1.0*s0/pi) q[0]; H q[2]; S q[3];␣
↪Rz(-1.0*s0/pi) q[2]; H q[2]; CX q[2], q[0]; V q[0]; S q[2]; S q[0]; CX q[1], q[2];␣
↪V q[0]; V q[1]; Sdg q[2]; S q[0]; S q[1]; Vdg q[2]; H q[0]; H q[1]; V q[2]; S q[0];␣
↪S q[1]; S q[2]; V q[0]; Sdg q[1]; H q[2]; S q[0]; Vdg q[1]; S q[2]; H q[0]; CX q[2],
↪ q[3]; S q[0]; Sdg q[3]; CX q[3], q[1]; Rz(1.0*s1/pi) q[1]; H q[3]; Rz(-1.0*s1/pi)␣
↪q[3]; H q[3]; CX q[3], q[1]; V q[1]; S q[3]; S q[1]; CX q[2], q[3]; V q[1]; V q[2];␣
↪Sdg q[3]; S q[1]; S q[2]; Vdg q[3]; H q[1]; H q[2]; V q[3]; S q[1]; S q[2]; S q[3];␣
↪CX q[0], q[1]; H q[3]; CX q[0], q[2]; S q[3]; CX q[0], q[3]; V q[0]; Rz(-0.25*d0/
↪pi) q[0]; CX q[3], q[0]; Rz(0.25*d0/pi) q[0]; CX q[2], q[0]; Rz(-0.25*d0/pi) q[0];␣
↪CX q[3], q[0]; Rz(0.25*d0/pi) q[0]; CX q[1], q[0]; Rz(0.25*d0/pi) q[0]; CX q[3],␣
↪q[0]; Rz(-0.25*d0/pi) q[0]; CX q[2], q[0]; Rz(0.25*d0/pi) q[0]; CX q[3], q[0]; Rz(-
↪0.25*d0/pi) q[0]; CX q[1], q[0]; Vdg q[0]; CX q[0], q[3]; CX q[0], q[2]; V q[3]; CX␣
↪q[0], q[1]; V q[2]; S q[3]; V q[0]; V q[1]; S q[2]; H q[3]; S q[0]; S q[1]; H q[2];␣
↪S q[3]; H q[0]; H q[1]; S q[2]; S q[0]; S q[1]; ]
```

To simulate the quantum computer, we connect to a backend using a pytket extension. The backends are where the quantum circuit is run or simulated. In this case, we are using the Qiskit state vector simulator through the use of

`pytket.extensions.qiskit`.

There are two broad approaches to simulating the action of a quantum circuit which differ in how measurement is treated. Firstly, the full state of the qubits may be tracked and returned as a vector of complex amplitudes – *state vector* simulation. Alternatively, we can build up the probability distribution of through repeated measurement of the qubit register – *shot based* simulation. This latter approach is a more faithful simulation of how current quantum computation is performed, but requires many repetitions to obtain accurate statistics of the desired quantity. The type of simulator influences how InQuanto handles the result returned by the backend. The Qiskit AerStateBackend here performs a state vector simulation.

Finally, we also need to choose how the classical optimization of Ansatz parameters is performed. This functionality is provided by the inquanto.minimizers module. In this case, we use Scipy to perform the minimization, which is interfaced through `inquanto.minimizers.scipy`. The choice of optimization algorithm and the settings can be passed through to Scipy. Here, we have requested an L-BFGS-B optimizer.

[ ]:
```
# ############################### #
# SIMULATOR AND OPTIMIZER DETAILS #
# ############################### #

# install pytket-qiskit using e.g. pip install pytket-qiskit if necessary


from pytket.extensions.qiskit import AerStateBackend
from inquanto.minimizers import MinimizerScipy

backend = AerStateBackend()
minimizer = MinimizerScipy(method="L-BFGS-B")
```

To make the first VQE simulation simple we use the `run_vqe(...)` function from the `inquanto.express` that is suitable to run with Qiskit state vector simulator. In order to perform much more customized VQE experiments, it is recommended to use `AlgorithmVQE` and the corresponding `Computables` and `Protocols`.

After all these, we can call the `run_vqe(...)` function that also executes the simulation:

[ ]:
```
# ############ #
# OPTIMIZATION #
# ############ #

from inquanto.express import run_vqe

vqe = run_vqe(ansatz, qubit_hamiltonian, backend=backend, with_gradient=True,␣
↪minimizer=minimizer)

report = vqe.generate_report()

print('\nVQE ENERGY: {}'.format(report['final_value']))
print('\nVQE REPORT: ')
vqe.generate_report()
```

```
# TIMER BLOCK-0 BEGINS AT 2023-03-28 15:45:39.429128
# TIMER BLOCK-0 ENDS - DURATION (s):  0.1461670 [0:00:00.146167]

VQE ENERGY: -1.1368465754720536

VQE REPORT:
```

```
{'minimizer': {'final_value': -1.1368465754720536,
  'final_parameters': array([ 0.        ,  0.        , -0.10723347])},
 'final_value': -1.1368465754720536,
```

(continues on next page)

(continued from previous page)

```
 'initial_parameters': [{'ordering': 0, 'symbol': 's0', 'value': 0.0},
  {'ordering': 1, 'symbol': 's1', 'value': 0.0},
  {'ordering': 2, 'symbol': 'd0', 'value': 0.0}],
 'final_parameters': [{'ordering': 0, 'symbol': 's0', 'value': 0.0},
  {'ordering': 1, 'symbol': 's1', 'value': 0.0},
  {'ordering': 2, 'symbol': 'd0', 'value': -0.10723347230091546}]}
```

The `.generate_report()` provides the details of the result. We can see here that the VQE simulation has successfully converged, giving an energy of -1.1368 – a big improvement over the Hartree-Fock energy of -1.1175! This tutorial forms the basic workflow of running simple VQE calculations in InQuanto. From here, a few things can be very easily switched up - for instance, you could try:

- Changing the molecule to another small example (such as LiH)
- Changing the qubit mapping (for instance, to `inquanto.fock_space.QubitMappingBravyiKitaev`) to see the impact on circuit gate counts.
- Changing the Ansatz state (for instance, to `inquanto.ansatz.FockSpaceAnsatzUCCGD`) to see the impact on the energy.
- Changing the optimizer method.
- Restricting the active space with the `frozen` parameter in the driver, or enabling point group symmetry with `point_group_symmetry=True` to see the impact on the number of Ansatz parameters.

Other topics – for instance, using other quantum algorithms, or fragmentation methods to look at larger systems – will be covered in the following tutorials.

16.2 Tutorial 2 - Further VQE Topics

In Tutorial 1, we considered a canonical VQE calculation at a single geometry with no resource optimization. However, in general, this will only be the first step in an analysis of the quantum algorithm. We may wish to expand on this analysis by considering more molecular geometries or systems – for example, looking at the energetics of bond dissociation. We also may wish to compare optimization methods in order to assess their effectiveness at reducing the overall cost with regards to quantum computational resources.

In this tutorial, we will look at how to achieve these goals using inquanto. We start by examining bond dissociation in molecular hydrogen using a canonical VQE approach. Then, we will look at a slightly larger system – the bending and stretching of water. As this is a larger system, we will have to introduce optimizations to enable the simulations to run on a standard laptop. Specifically, we introduce how to reduce the active space (and thus the number of qubits in the quantum computation) by freezing orbitals using the PySCF driver. Finally, we look at one optimization strategy in inquanto - Ansatz parameter reduction by point group symmetry. The tutorial described here is available in HTML format in the inquanto documentation.

[ ]:
```
from pytket.extensions.qiskit import AerStateBackend
from inquanto.express import run_vqe
from inquanto.minimizers import MinimizerScipy
from inquanto.ansatzes import FermionSpaceAnsatzUCCSD
from inquanto.mappings import QubitMappingJordanWigner
from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF
import datetime
import matplotlib.pyplot as plt
import numpy as np
```

16.2.1 H2 Bond Stretching

After imports, we start by examining bond dissociation in molecular hydrogen in order to present a general workflow.

```
[ ]: def hydrogen_vqe_energy(bond_length):
         basis = 'STO-3G'
         geometry = [["H", [0, 0, 0]], ["H", [0, 0, bond_length]]]
         charge = 0

         driver = ChemistryDriverPySCFMolecularRHF(basis=basis, geometry=geometry,␣
     ↪charge=charge)
         fermionic_hamiltonian, fock_space, fock_state = driver.get_system()
         jw = QubitMappingJordanWigner
         qubit_hamiltonian = jw.operator_map(fermionic_hamiltonian)
         ansatz = FermionSpaceAnsatzUCCSD(fock_space, fock_state, jw)
         backend = AerStateBackend()
         minimizer = MinimizerScipy(method="L-BFGS-B")
         vqe = run_vqe(ansatz, qubit_hamiltonian, backend=backend, with_gradient=True,␣
     ↪minimizer=minimizer)

         ground_state_energy = vqe.generate_report()["final_value"]
         hartree_fock_energy = driver.mf_energy
         return ground_state_energy, hartree_fock_energy

     print(hydrogen_vqe_energy(0.741))
```

```
# TIMER BLOCK-0 BEGINS AT 2023-03-23 12:08:38.357489
# TIMER BLOCK-0 ENDS - DURATION (s):  0.3170135 [0:00:00.317014]
(-1.1372744055294364, -1.116706137236105)
```

The code here does all that is necessary to generate a ground state energy using canonical VQE for the hydrogen molecule, as in Tutorial 1. Here, we have wrapped it in a function to allow us to easily view the change with bond length:

```
[ ]: h2_bond_lengths = np.linspace(0.4,2.0,20)
     h2_results = [hydrogen_vqe_energy(x) for x in h2_bond_lengths]
     print(h2_results)
```

```
# TIMER BLOCK-1 BEGINS AT 2023-03-23 12:12:13.497597
# TIMER BLOCK-1 ENDS - DURATION (s):  0.3516882 [0:00:00.351688]
# TIMER BLOCK-2 BEGINS AT 2023-03-23 12:12:13.989042
# TIMER BLOCK-2 ENDS - DURATION (s):  0.2756270 [0:00:00.275627]
# TIMER BLOCK-3 BEGINS AT 2023-03-23 12:12:14.397974
# TIMER BLOCK-3 ENDS - DURATION (s):  0.2672877 [0:00:00.267288]
# TIMER BLOCK-4 BEGINS AT 2023-03-23 12:12:14.798315
# TIMER BLOCK-4 ENDS - DURATION (s):  0.2691947 [0:00:00.269195]
# TIMER BLOCK-5 BEGINS AT 2023-03-23 12:12:15.198835
# TIMER BLOCK-5 ENDS - DURATION (s):  0.2670629 [0:00:00.267063]
# TIMER BLOCK-6 BEGINS AT 2023-03-23 12:12:15.623292
# TIMER BLOCK-6 ENDS - DURATION (s):  0.3118640 [0:00:00.311864]
# TIMER BLOCK-7 BEGINS AT 2023-03-23 12:12:16.151589
# TIMER BLOCK-7 ENDS - DURATION (s):  0.2882644 [0:00:00.288264]
# TIMER BLOCK-8 BEGINS AT 2023-03-23 12:12:16.572625
# TIMER BLOCK-8 ENDS - DURATION (s):  0.2683956 [0:00:00.268396]
# TIMER BLOCK-9 BEGINS AT 2023-03-23 12:12:16.973561
# TIMER BLOCK-9 ENDS - DURATION (s):  0.2653925 [0:00:00.265392]
# TIMER BLOCK-10 BEGINS AT 2023-03-23 12:12:17.370130
# TIMER BLOCK-10 ENDS - DURATION (s):  0.2674740 [0:00:00.267474]
# TIMER BLOCK-11 BEGINS AT 2023-03-23 12:12:17.769096
# TIMER BLOCK-11 ENDS - DURATION (s):  0.2684676 [0:00:00.268468]
```

(continues on next page)

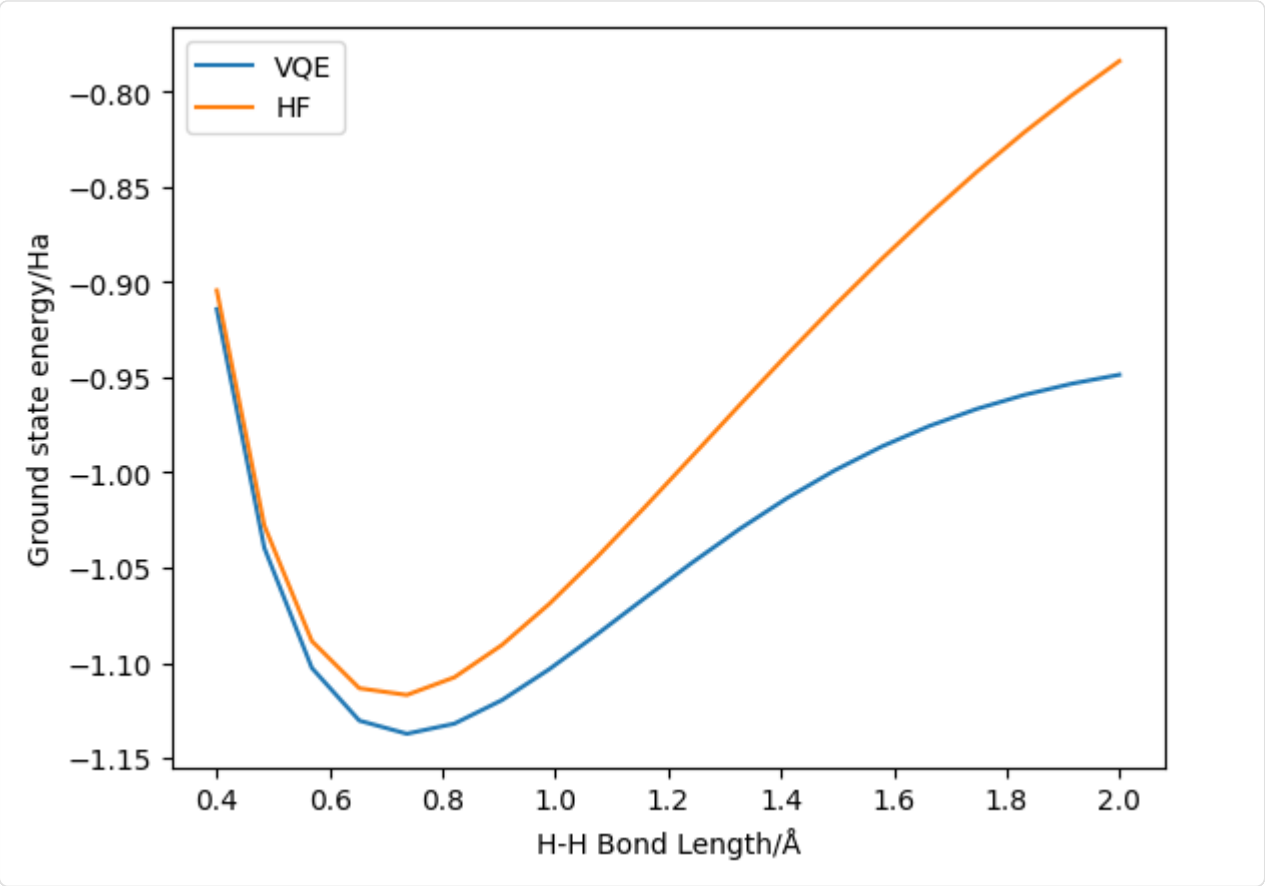
(continued from previous page)

```
# TIMER BLOCK-12 BEGINS AT 2023-03-23 12:12:18.173184
# TIMER BLOCK-12 ENDS - DURATION (s):  0.2746285 [0:00:00.274628]
# TIMER BLOCK-13 BEGINS AT 2023-03-23 12:12:18.586528
# TIMER BLOCK-13 ENDS - DURATION (s):  0.2667404 [0:00:00.266740]
# TIMER BLOCK-14 BEGINS AT 2023-03-23 12:12:19.024038
# TIMER BLOCK-14 ENDS - DURATION (s):  0.3045242 [0:00:00.304524]
# TIMER BLOCK-15 BEGINS AT 2023-03-23 12:12:19.480942
# TIMER BLOCK-15 ENDS - DURATION (s):  0.2770261 [0:00:00.277026]
# TIMER BLOCK-16 BEGINS AT 2023-03-23 12:12:19.889404
# TIMER BLOCK-16 ENDS - DURATION (s):  0.2695249 [0:00:00.269525]
# TIMER BLOCK-17 BEGINS AT 2023-03-23 12:12:20.290592
# TIMER BLOCK-17 ENDS - DURATION (s):  0.2632620 [0:00:00.263262]
# TIMER BLOCK-18 BEGINS AT 2023-03-23 12:12:20.684036
# TIMER BLOCK-18 ENDS - DURATION (s):  0.2651360 [0:00:00.265136]
# TIMER BLOCK-19 BEGINS AT 2023-03-23 12:12:21.081271
# TIMER BLOCK-19 ENDS - DURATION (s):  0.2633803 [0:00:00.263380]
# TIMER BLOCK-20 BEGINS AT 2023-03-23 12:12:21.476280
# TIMER BLOCK-20 ENDS - DURATION (s):  0.2662574 [0:00:00.266257]
[(-0.9141497046270836, -0.9043613941635398), (-1.0396441933684184, -1.
↪0278952240485908), (-1.102723451217336, -1.0885821110334752), (-1.1303984654811357,␣
↪-1.1133931546411708), (-1.1373027360323478, -1.1169158055488677), (-1.
↪1320031440770353, -1.1076586023375483), (-1.1196476526566383, -1.0906923776851998),␣
↪(-1.1033573201316123, -1.0690432214496197), (-1.0850885605499418, -1.
↪04456492118589), (-1.0661536707290793, -1.0184750668009017), (-1.0474923241943452, -
↪0.9916426659510071), (-1.029790070531814, -0.9647203151980028), (-1.01352559104073,␣
↪-0.9382014307919535), (-0.998995920249617, -0.9124510069368059), (-0.
↪9863403143934911, -0.8877296363816177), (-0.9755678355211703, -0.8642149983652552),␣
↪(-0.9665878877681977, -0.8420201240601859), (-0.9592413973101763, -0.
↪8212077601664908), (-0.953330033975642, -0.8018012103709642), (-0.9486411121756364,␣
↪-0.7837926542773532)]
```

We have successfully generated a potential energy curve for the dissociation of the H2 molecule using VQE and, as a reference, Hartree-Fock. We can now separate the VQE and HF results and plot them:

[ ]:
```
h2_vqe_results,h2_hf_results = zip(*h2_results)
plt.plot(h2_bond_lengths,h2_vqe_results,label='VQE')
plt.plot(h2_bond_lengths,h2_hf_results,label='HF')
plt.xlabel('H-H Bond Length/Å')
plt.ylabel('Ground state energy/Ha')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fdf8301fa60>
```



Here we can see the improvement obtained by using UCCSD VQE – indeed, this system is sufficiently low in number of spin-orbitals that UCCSD is exact (i.e. FCI-level). On the other hand, we can also observe the increasing inaccuracy of (restricted) Hartree-Fock at higher bond lengths.

16.2.2 H2O Bending - active space reduction

H2O in an STO-3G basis is a 14 spin-orbital (and thus 14 qubit for conventional qubit mappings) system. This is within the capacity of a classical computer to simulate, but such a simulation may perhaps require more resources than is practical for this tutorial. We can reduce the active spin-orbital space by freezing orbitals. While our purpose here is to demonstrate, in a real experiment it may be necessary to freeze orbitals in order to reduce the (exponentially growing) resources to a level that is actually implementable. In inquanto, orbital freezing is performed by passing the `frozen` parameter to the driver:

[ ]:
```
def water_bending_vqe_energy(bond_angle):

    x_h2 = np.sin(bond_angle / 360 * np.pi)
    x_h1 = -x_h2
    y_h1 = np.cos(bond_angle / 360 * np.pi)
    y_h2 = y_h1

    geometry = [['H', [x_h1, y_h1, 0.]], ['O', [0., 0., 0.]], ['H', [x_h2, y_h2, 0.]]]
    basis = 'STO-3G'
    charge = 0
    frozen = [0]
```

(continues on next page)

(continued from previous page)

```
    driver = ChemistryDriverPySCFMolecularRHF(basis=basis, geometry=geometry,␣
↪charge=charge, frozen=frozen)
    fermionic_hamiltonian, fock_space, fock_state = driver.get_system()
    jw = QubitMappingJordanWigner
    qubit_hamiltonian = jw.operator_map(fermionic_hamiltonian)
    ansatz = FermionSpaceAnsatzUCCSD(fock_space, fock_state, jw)
    backend = AerStateBackend()
    minimizer = MinimizerScipy(method="L-BFGS-B")
    vqe = run_vqe(ansatz, qubit_hamiltonian, backend=backend, with_gradient=True,␣
↪minimizer=minimizer)

    ground_state_energy = vqe.generate_report()["final_value"]
    hartree_fock_energy = driver.mf_energy
    return ground_state_energy, hartree_fock_energy

print(water_bending_vqe_energy(104.5))
```

```
# TIMER BLOCK-21 BEGINS AT 2023-03-23 12:20:05.346226
# TIMER BLOCK-21 ENDS - DURATION (s): 32.7244266 [0:00:32.724427]
(-75.01966834467423, -74.96466253913081)
```

This block may take up to a minute to run, as the system is a bit bigger than molecular hydrogen. Here, we have asked the driver to freeze the lowest energy spatial orbital (i.e. the core electrons). Note that frozen orbitals are specified as a list of indices of spatial orbitals, not spin-orbitals - so every orbital frozen in this way will save two qubits. Note that for consistency, we have here specified the geometry in Cartesian co-ordinates by explicitly calculating the position of each atom. It is also possible in inquanto to specify geometries in z-matrix format.

As before, we have successfully calculated the VQE and HF energy at (roughly) the equilibrium geometry. We can again calculate the effect of changing the bond angle and plot the results (this may take a few minutes to run - reducing the amount of data points generated will speed it up if needed):

[ ]:
```
h2o_bond_angles = np.linspace(45.,180.,10)
h2o_bending_results = [water_bending_vqe_energy(x) for x in h2o_bond_angles]
```

```
# TIMER BLOCK-22 BEGINS AT 2023-03-23 12:21:47.110960
# TIMER BLOCK-22 ENDS - DURATION (s): 32.5175582 [0:00:32.517558]
# TIMER BLOCK-23 BEGINS AT 2023-03-23 12:22:21.287194
# TIMER BLOCK-23 ENDS - DURATION (s): 31.7966331 [0:00:31.796633]
# TIMER BLOCK-24 BEGINS AT 2023-03-23 12:22:54.736986
# TIMER BLOCK-24 ENDS - DURATION (s): 31.9879692 [0:00:31.987969]
# TIMER BLOCK-25 BEGINS AT 2023-03-23 12:23:28.369565
# TIMER BLOCK-25 ENDS - DURATION (s): 32.0816250 [0:00:32.081625]
# TIMER BLOCK-26 BEGINS AT 2023-03-23 12:24:02.099793
# TIMER BLOCK-26 ENDS - DURATION (s): 32.1999291 [0:00:32.199929]
# TIMER BLOCK-27 BEGINS AT 2023-03-23 12:24:35.957437
# TIMER BLOCK-27 ENDS - DURATION (s): 35.1730628 [0:00:35.173063]
# TIMER BLOCK-28 BEGINS AT 2023-03-23 12:25:12.790397
# TIMER BLOCK-28 ENDS - DURATION (s): 35.5254725 [0:00:35.525472]
# TIMER BLOCK-29 BEGINS AT 2023-03-23 12:25:49.976774
# TIMER BLOCK-29 ENDS - DURATION (s): 38.1978390 [0:00:38.197839]
# TIMER BLOCK-30 BEGINS AT 2023-03-23 12:26:29.838531
# TIMER BLOCK-30 ENDS - DURATION (s): 41.4493452 [0:00:41.449345]
# TIMER BLOCK-31 BEGINS AT 2023-03-23 12:27:12.571964
# TIMER BLOCK-31 ENDS - DURATION (s): 28.4476903 [0:00:28.447690]
```

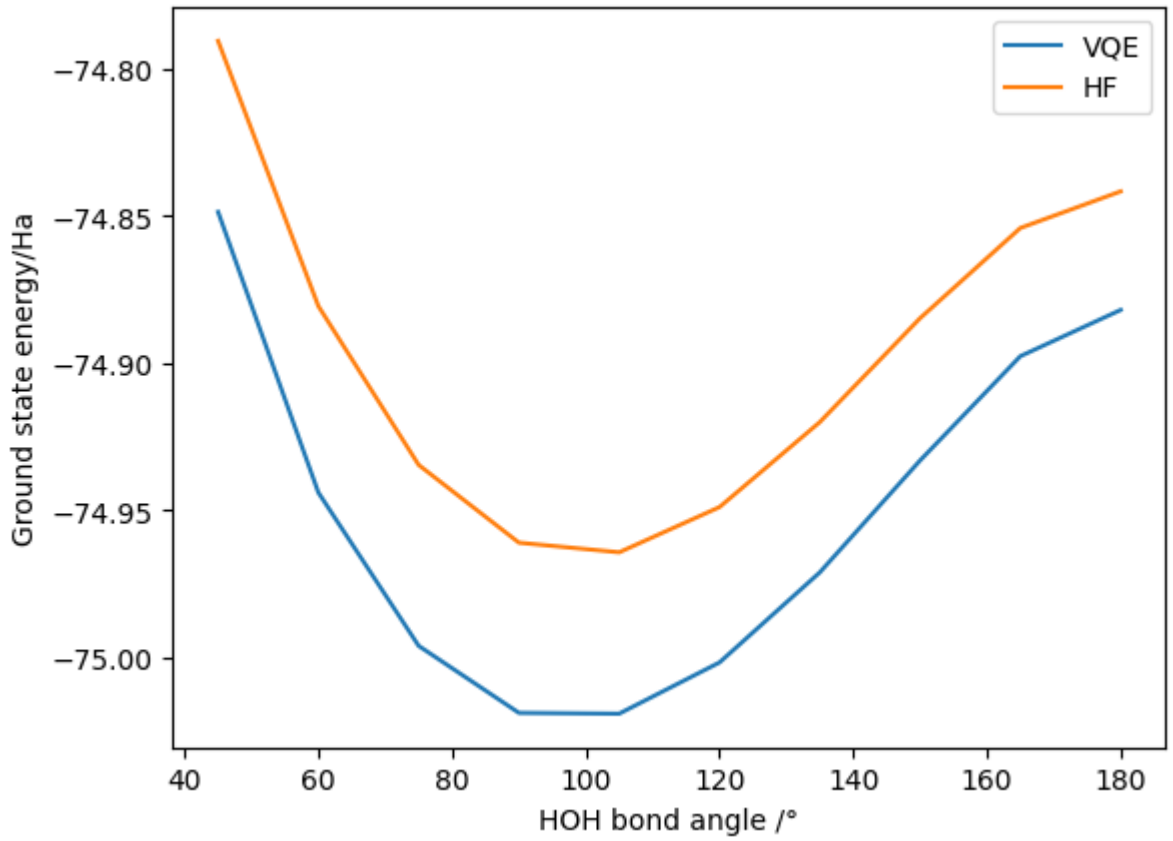
[ ]:
```
h2o_angle_vqe_results,h2o_angle_hf_results = zip(*h2o_bending_results)
plt.plot(h2o_bond_angles,h2o_angle_vqe_results,label='VQE')
```

(continues on next page)

(continued from previous page)

```
plt.plot(h2o_bond_angles,h2o_angle_hf_results,label='HF')
plt.xlabel('HOH bond angle /°')
plt.ylabel('Ground state energy/Ha')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fdf830b3be0>
```



16.2.3 H2O stretching - symmetry-allowed excitations

In larger systems, we may be interested in benchmarking the cost of VQE with various optimization schemes. Choosing optimization schemes is a question of balancing the need for accuracy and resource constraints. Several resource constraints occur when performing quantum algorithms – for instance, the number of qubits and the circuit length. Similar to space and time in classical computing, certain optimization schemes may reduce cost in one metric while having a detrimental effect on others. Such tradeoff in resources applies to VQE itself; VQE as an algorithm is designed to replace the (extremely) long quantum circuits of the phase estimation algorithm with significantly more but much shorter circuits.

Many such optimization schemes are available in inquanto and examples of their use can be found in the examples directory. In this tutorial, we will look at the use of point group symmetry to exclude unphysical symmetry-violating excitations. This is a technique commonly used in quantum chemistry codes on classical computers, and can substantially reduce the number of Ansatz parameters. In turn, the quantum circuit length can be reduced, as is the difficulty of classical optimization (and consequentially the number of individual VQE shots required, and thus the overall runtime.) As this technique is simply removing excitations that are unphysical, it is essentially "free" with regards to other computational resources.

First, we modify our VQE routine to incorporate point group symmetry – this time, in the context of symmetric bond stretching:

[ ]:
```
def water_stretching_vqe_energy(bond_length):

    x_h2 = bond_length * np.sin(104.45 / 360 * np.pi)
    x_h1 = -x_h2
    y_h1 = bond_length * np.cos(104.45 / 360 * np.pi)
    y_h2 = y_h1

    geometry = [['H', [x_h1, y_h1, 0.]], ['O', [0., 0., 0.]], ['H', [x_h2, y_h2, 0.]]]
    basis = 'STO-3G'
    charge = 0
    frozen = [0]

    driver = ChemistryDriverPySCFMolecularRHF(basis=basis, geometry=geometry,␣
↪charge=charge, frozen=frozen,point_group_symmetry=True)
    fermionic_hamiltonian, fock_space, fock_state = driver.get_system()
    jw = QubitMappingJordanWigner
    qubit_hamiltonian = jw.operator_map(fermionic_hamiltonian)
    ansatz = FermionSpaceAnsatzUCCSD(fock_space, fock_state, jw)
    backend = AerStateBackend()
    minimizer = MinimizerScipy(method="L-BFGS-B")
    vqe = run_vqe(ansatz, qubit_hamiltonian, backend=backend, with_gradient=True,␣
↪minimizer=minimizer)

    ground_state_energy = vqe.generate_report()["final_value"]
    hartree_fock_energy = driver.mf_energy
    return ground_state_energy, hartree_fock_energy
print(water_stretching_vqe_energy(1.))
```

```
# TIMER BLOCK-32 BEGINS AT 2023-03-23 12:28:56.469750
# TIMER BLOCK-32 ENDS - DURATION (s): 15.9202285 [0:00:15.920229]
(-75.01969733754392, -74.96468314023907)
```

Here, incorporating point group symmetry is as simple as passing `point_group_symmetry=True` to the driver. Note that the ability to use point group symmetry is reliant on the capacity of the underlying classical quantum chemistry package (in this case, PySCF). We then generate a plot of the change in the ground state energy as the bonds stretch:

[ ]:
```
h2o_bond_lengths = np.linspace(0.6,2.,10)
h2o_stretching_results = [water_stretching_vqe_energy(x) for x in h2o_bond_lengths]
```

```
# TIMER BLOCK-33 BEGINS AT 2023-03-23 12:30:20.479622
# TIMER BLOCK-33 ENDS - DURATION (s): 14.0841113 [0:00:14.084111]
# TIMER BLOCK-34 BEGINS AT 2023-03-23 12:30:36.008481
# TIMER BLOCK-34 ENDS - DURATION (s): 13.6774870 [0:00:13.677487]
# TIMER BLOCK-35 BEGINS AT 2023-03-23 12:30:51.077902
# TIMER BLOCK-35 ENDS - DURATION (s): 15.9926533 [0:00:15.992653]
# TIMER BLOCK-36 BEGINS AT 2023-03-23 12:31:08.614493
# TIMER BLOCK-36 ENDS - DURATION (s): 15.5553551 [0:00:15.555355]
# TIMER BLOCK-37 BEGINS AT 2023-03-23 12:31:25.610316
# TIMER BLOCK-37 ENDS - DURATION (s): 17.1351932 [0:00:17.135193]
# TIMER BLOCK-38 BEGINS AT 2023-03-23 12:31:44.191697
# TIMER BLOCK-38 ENDS - DURATION (s): 20.5675263 [0:00:20.567526]
# TIMER BLOCK-39 BEGINS AT 2023-03-23 12:32:06.109417
# TIMER BLOCK-39 ENDS - DURATION (s): 19.1885227 [0:00:19.188523]
# TIMER BLOCK-40 BEGINS AT 2023-03-23 12:32:26.672505
# TIMER BLOCK-40 ENDS - DURATION (s): 23.8458130 [0:00:23.845813]
# TIMER BLOCK-41 BEGINS AT 2023-03-23 12:32:51.909317
# TIMER BLOCK-41 ENDS - DURATION (s): 30.1823820 [0:00:30.182382]
# TIMER BLOCK-42 BEGINS AT 2023-03-23 12:33:23.596787
```

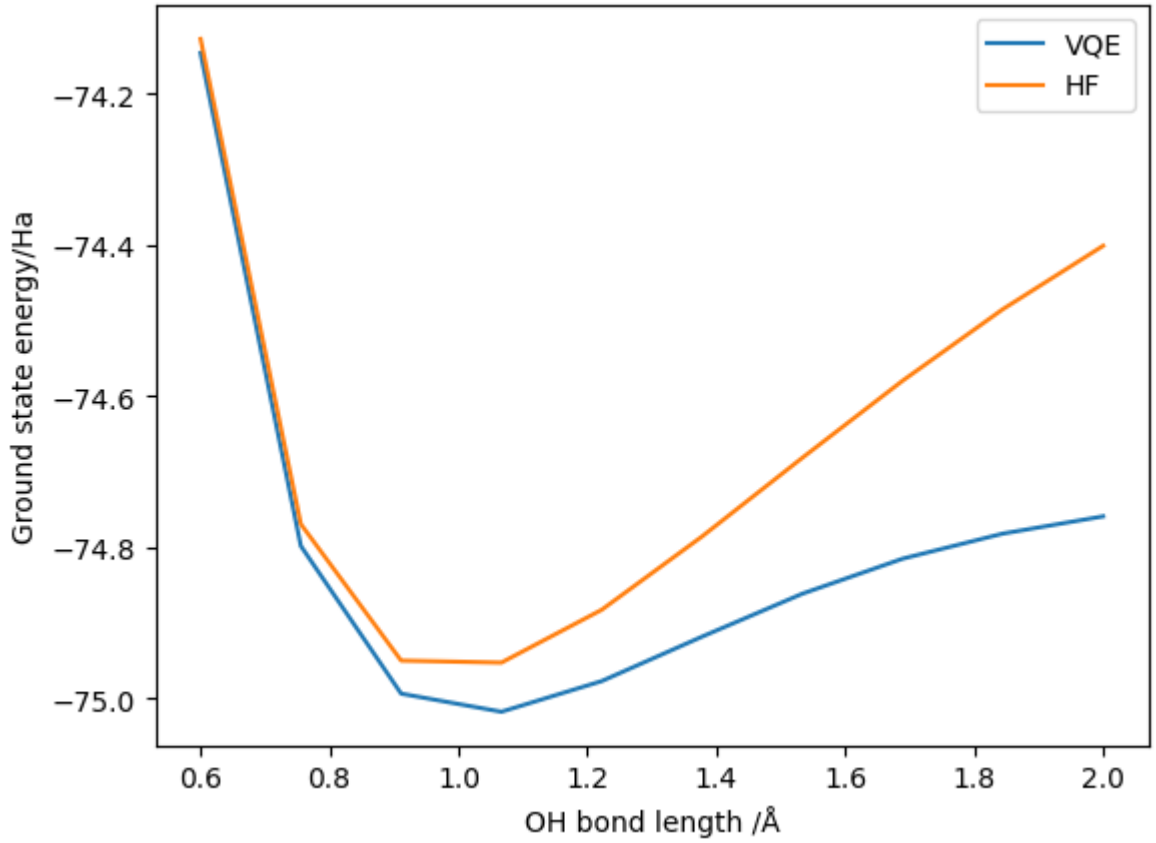
(continues on next page)

(continued from previous page)

```
# TIMER BLOCK-42 ENDS - DURATION (s): 33.3278710 [0:00:33.327871]
```

[ ]:
```
h2o_lengths_vqe_results,h2o_lengths_hf_results = zip(*h2o_stretching_results)
plt.plot(h2o_bond_lengths,h2o_lengths_vqe_results,label='VQE')
plt.plot(h2o_bond_lengths,h2o_lengths_hf_results,label='HF')
plt.xlabel('OH bond length /Å')
plt.ylabel('Ground state energy/Ha')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fdf82f35700>
```



We have successfully demonstrated that the point group symmetry reductions yield the correct ground state energy. However, we have not yet looked at the resource reductions obtained. We can adapt our VQE wrapper functions to return this, but for simplicity here we just look at one configuration:

[ ]:
```
x_h2 = np.sin(104.45 / 360 * np.pi)
x_h1 = -x_h2
y_h1 = np.cos(104.45 / 360 * np.pi)
y_h2 = y_h1

geometry = [['H', [x_h1, y_h1, 0.]], ['O', [0., 0., 0.]], ['H', [x_h2, y_h2, 0.]]]
basis = 'STO-3G'
charge = 0
frozen = [0]

driver_with_symmetry = ChemistryDriverPySCFMolecularRHF(basis=basis,␣
```

(continues on next page)

(continued from previous page)

```
↪geometry=geometry, charge=charge, frozen=frozen,point_group_symmetry=True)
driver_without_symmetry = ChemistryDriverPySCFMolecularRHF(basis=basis,␣
↪geometry=geometry, charge=charge, frozen=frozen,point_group_symmetry=False)
fermionic_hamiltonian_with_symmetry, fock_space_with_symmetry, fock_state_with_
↪symmetry = driver_with_symmetry.get_system()
fermionic_hamiltonian_without_symmetry, fock_space_without_symmetry, fock_state_
↪without_symmetry = driver_without_symmetry.get_system()
jw = QubitMappingJordanWigner()

ansatz_with_symmetry = FermionSpaceAnsatzUCCSD(fock_space_with_symmetry, fock_state_
↪with_symmetry, jw)
ansatz_without_symmetry = FermionSpaceAnsatzUCCSD(fock_space_without_symmetry, fock_
↪state_without_symmetry, jw)
```

For simplicity, we restrict ourselves to looking at the impact on the resources required for generating the Ansatz state. Note that many quantum resources can be estimated without actually running VQE in this manner; this dramatically decreases the resources necessary to perform an experiment.

If we generate a report for each Ansatz:

[ ]:
```
print('### ANSATZ RESOURCES WITH SYMMETRY REDUCTION ###')
print(ansatz_with_symmetry.generate_report())
print('\n### ANSATZ RESOURCES WITHOUT SYMMETRY REDUCTION ###')
print(ansatz_without_symmetry.generate_report())
```

```
### ANSATZ RESOURCES WITH SYMMETRY REDUCTION ###
{'ansatz_circuit_depth': 841, 'ansatz_circuit_gates': 2260, 'n_parameters': 30, 'n_
↪qubits': 12}

### ANSATZ RESOURCES WITHOUT SYMMETRY REDUCTION ###
{'ansatz_circuit_depth': 2798, 'ansatz_circuit_gates': 7032, 'n_parameters': 92, 'n_
↪qubits': 12}
```

16.3 Tutorial 3 - Tackling larger systems with fragmentation

In tutorial 1 and 2, we covered how to run a simple VQE calculation using InQuanto and some optimizations that can be performed. Here, we look at using Density Matrix Embedding Theory (DMET) to examine a larger system by fragmenting it. As an example, we consider HCOOH (formic acid). Without considering symmetry or active space reductions, this system would require 34 qubits to simulate using an STO-3G basis. This requires more resources that are available on current quantum computers, and will be extremely expensive to simulate on a classical device.

DMET is a method of studying large molecules by partitioning the system into fragments containing a smaller number of atoms. Each fragment is treated independently in a bath corresponding to the molecular environment. Crucially, DMET allows for different fragments to be treated using different electronic structure methods. For example, we could imagine using VQE on the quantum computer to treat one particular fragment of interest. We focus here on a simplified implementation of DMET – the so-called one-shot DMET. More examples are in the examples/embeddings folder. For discussion of the theory underpinning DMET, see Knizia & Chan (2012). As DMET relies heavily on performing classical electronic structure calculations in addition to any quantum computations, we need to import a driver and the fragment solvers from the `inquanto-pyscf` extension.

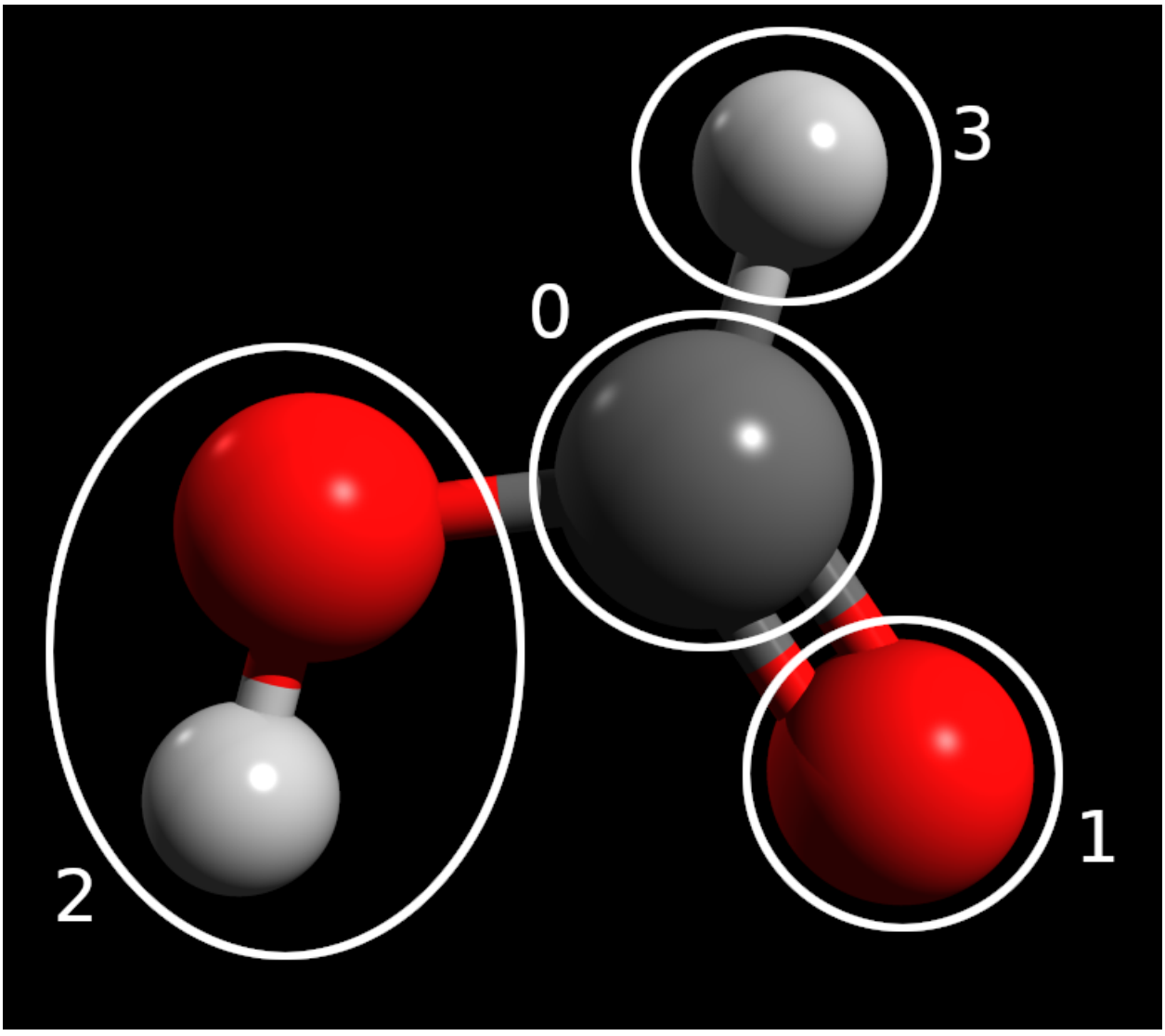
[ ]:
```
from inquanto.geometries import GeometryMolecular
from inquanto.embeddings import DMETRHF
from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF
from inquanto.extensions.pyscf import DMETRHFFragmentPySCFCCSD, get_fragment_orbital_
```

(continues on next page)

(continued from previous page)

```
↪masks, get_fragment_orbitals
from pytket.extensions.qiskit import AerStateBackend
```

In order to use DMET to study our system, we must choose a scheme to split the molecule into fragments. In general, this is a task which requires chemical intuition and an awareness of the resource implications of the size of each fragment. As our goal here is to perform a simulation that runs quickly (and not to obtain highly accurate results), we choose a very fine fragmentation scheme with several small fragments.



The above figure shows the fragmentation scheme graphically. We see that other than the hydroxyl, each atom is in its own fragment. This ensures that the maximum number of qubits required to simulate an individual fragment would be 12 (the hydroxyl fragment), a reasonable number of qubits to simulate on a large classical computer.

[ ]:
```
# ################# #
# MOLECULE & DRIVER #
# ################# #

xyz = [['C', [0.000, 0.442, 0.000]],
       ['O', [-1.046, -0.467, 0.000]],
```

(continues on next page)

(continued from previous page)

```
            ['O', [1.171, 0.120, 0.000]],
            ['H', [-0.389, 1.475, 0.000]],
            ['H', [-0.609, -1.355, 0.000]]]

geometry = GeometryMolecular(xyz)

basis = 'sto-3g'
charge = 0
driver = ChemistryDriverPySCFMolecularRHF(basis='sto-3g',
                                          geometry=geometry,
                                                charge=0,
                                                verbose=0)

hamiltonian_operator, space, rdm1 = driver.get_lowdin_system()

dmet = DMETRHF(hamiltonian_operator, rdm1)
```

As before, we first initialise the driver. The initialisation of the driver is much the same as in standard VQE with regards to the molecule geometry, basis and charge specification. However, because of the spatial fragmentation, DMET requires the localisation of the molecular orbitals. Therefore, when we compute the Hamiltonian operator, instead of `get_system()` we call the `get_lowdin_system()` method. This will perform an RHF simulation of the molecule, and will return the mean-field 1e-RDM as `rdm1`, the `space` and the `hamiltonian_operator` in the Löwdin basis, that is computed by Löwdin symmetric orthogonalisation of the atomic orbitals.

Finally, we initialize the DMET method with the `hamiltonian_operator` and the `rdm1`.

At this stage, we have not specified any particular fragmentation scheme. Fragments in InQuanto are associated with the level of electronic structure theory that will be used to simulate them. As a first test, we try specifying that each fragment should be treated with classical CCSD.

In order to specify a fragment, we need to determine the corresponding Löwdin orbitals. The easiest way to specify the fragments is by atoms. Based on the indices in the `xyz` table, we can make four sets of atom indices, and using the `get_fragment_orbitals()` utility function we can tabulate the orbitals and the four orbital fragment masks that select the Löwdin orbitals corresponding to the fragments:

[ ]:
```
get_fragment_orbitals(driver, [0], [2], [1, 4], [3])
```

		0	1	2	3	4
0	0	C 1s	True	False	False	False
1	0	C 2s	True	False	False	False
2	0	C 2px	True	False	False	False
3	0	C 2py	True	False	False	False
4	0	C 2pz	True	False	False	False
5	1	O 1s	False	False	True	False
6	1	O 2s	False	False	True	False
7	1	O 2px	False	False	True	False
8	1	O 2py	False	False	True	False
9	1	O 2pz	False	False	True	False
10	2	O 1s	False	True	False	False
11	2	O 2s	False	True	False	False
12	2	O 2px	False	True	False	False
13	2	O 2py	False	True	False	False
14	2	O 2pz	False	True	False	False
15	3	H 1s	False	False	False	True
16	4	H 1s	False	False	True	False

With another utility function `get_fragment_orbital_masks(...)` we can obtain the orbital fragment masks as arrays. Once we have the orbital masks we can complete the DMET simulation.

```
[ ]: maskC, maskO, maskOH, maskH = get_fragment_orbital_masks(driver, [0], [2], [1, 4],␣
     ↪[3])

     fragments = [DMETRHFFragmentPySCFCCSD(dmet, maskC, name="C")]
     fragments += [DMETRHFFragmentPySCFCCSD(dmet, maskO, name="O")]
     fragments += [DMETRHFFragmentPySCFCCSD(dmet, maskOH, name="OH")]
     fragments += [DMETRHFFragmentPySCFCCSD(dmet, maskH, name="H")]

     result = dmet.run(fragments)
```

```
# STARTING CHEMICAL POTENTIAL 0.0
# STARTING CORR POTENTIAL PARAMETERS []

# FULL SCF ITERATION 0
# NEWTON ITERATION - CHEMICAL POTENTIAL 0.0
# FRAGMENT 0 - C: <H>=-186.31338432615402  EFR=-58.83801802368333  Q=0.
↪020105631085752584
# FRAGMENT 1 - O: <H>=-186.3133853207737  EFR=-94.07923540835995  Q=-0.
↪08915722472862164
# FRAGMENT 2 - OH: <H>=-186.29269508290218  EFR=-98.21603860323465  Q=-0.
↪07076155306003784
# FRAGMENT 3 - H: <H>=-186.23216575139494  EFR=-3.805406920135896  Q=0.
↪0029867861860907174
# NEWTON ITERATION - CHEMICAL POTENTIAL 0.0001
# FRAGMENT 0 - C: <H>=-186.31396619851364  EFR=-58.838989888576215  Q=0.
↪02031447102446382
# FRAGMENT 1 - O: <H>=-186.31419656704105  EFR=-94.07970951859824  Q=-0.
↪08903855658755866
# FRAGMENT 2 - OH: <H>=-186.29359289137582  EFR=-98.21647309737241  Q=-0.
↪0706448202236345
# FRAGMENT 3 - H: <H>=-186.23226272837837  EFR=-3.805683950277596  Q=0.
↪0030663357332632035
# NEWTON ITERATION - CHEMICAL POTENTIAL 0.026122346642530214
# FRAGMENT 0 - C: <H>=-186.46609970037895  EFR=-59.091294391283746  Q=0.
↪07462903595384773
# FRAGMENT 1 - O: <H>=-186.525712122053  EFR=-94.20327799729976  Q=-0.
↪058040637121511196
# FRAGMENT 2 - OH: <H>=-186.52760819711878  EFR=-98.32803927523938  Q=-0.
↪040673659945895224
# FRAGMENT 3 - H: <H>=-186.25776880475385  EFR=-3.877650190636578 6  Q=0.
↪023767280254408885
# NEWTON ITERATION - CHEMICAL POTENTIAL 0.026183195699923112
# FRAGMENT 0 - C: <H>=-186.4664571127612  EFR=-59.09188298091364  Q=0.
↪07475597327773631
# FRAGMENT 1 - O: <H>=-186.5262076771367  EFR=-94.20356742638009  Q=-0.
↪057967869179652354
# FRAGMENT 2 - OH: <H>=-186.52815629411177  EFR=-98.32829670367684  Q=-0.
↪04060450873466692
# FRAGMENT 3 - H: <H>=-186.2578290778784  EFR=-3.877818167978858  Q=0.
↪023815682210127087
# CHEMICAL POTENTIAL 0.026183334258789784
# FINAL PARAMETERS: []
# FINAL CHEMICAL POTENTIAL: 0.026183334258789784
# FINAL ENERGY: -186.53116138512004
```

In the first block, we have specified our fragments as a list. We use the `inquanto.extensions.pyscf.DMETRHFFragmentPySCFCCSD` class as we want to look at each fragment using classical coupled cluster. Each fragment takes the `dmet` as an argument in addition to an arbitrary string giving the fragment name. It also requires the

masks for the orbitals within the fragment to be specified. These are given as an array of booleans, marking the index of orbitals with `True` if it is in the fragment and `False` if it is outside it.

The `dmet.run()` method is then invoked passing the fragments as a list. During the execution we can observe details about the calculation. The FINAL ENERGY line in the end gives us the final ground state energy of the system calculated by DMET.

[ ]:
```
print("REFERENCE MP2  ENERGY: ", driver.run_mp2())
print("REFERENCE CCSD ENERGY: ", driver.run_ccsd())
```

```
REFERENCE MP2  ENERGY:  -186.37687192609548
REFERENCE CCSD ENERGY:  -186.40300888507954
```

As a reference we computed MP2 and CCSD energies. We can see that in this instance DMET obtains about 0.13 Ha more correlation energy than non-DMET classical CCSD. Although defeating the point of fragmenting the system to reduce resource requirements, benchmarking a DMET calculation with the same level of theory for each fragment against a non-DMET calculation is a good way to estimate the error incurred with the fragmentation scheme. Note that DMET is non-variational and thus can yield energies lower than the exact (i.e. FCI-level) energy.

DMET allows the use of different levels of theory for each fragment. By using the examples, we encourage the reader to modify this notebook such that some of the fragments (for example the lone hydrogen atom) are using the VQE method with a state vector simulator.

16.4 Tutorial 4 - Visualisation with inquanto-nglview

It is often useful to visualise certain molecular data. For this purpose, `inquanto` uses an extension to interface with the nglview package. The inquanto-ngl view extension is focussed on providing some basic visualizing utilities to the user. The return types from the functions are widgets which can be viewed in a jupyter notebook. The functionality includes visualizing molecular structures, fragmentation patterns defined in the `Geometry` object and molecular orbitals.

16.4.1 Visualizing Structures

The argument type to be passed to the constructor for the `VisualizerNGL` object corresponding to the geometry is one of the native inquanto `Geometry` objects. The molecule can then be visualized by calling the `.visualize_molecule()` in the last line of a notebook cell. See below for a short example.

[ ]:
```
# pip install packages/inquanto-nglview ???
# pip install ipywidgets==7.7.3
# pip install pint==0.19.1


from inquanto.geometries import GeometryMolecular
from inquanto.extensions.nglview import VisualizerNGL

xyz = [
    ['C', [ 0.0000000,  1.4113170, 0.0000000]],
    ['C', [ 1.2222370,  0.7056590, 0.0000000]],
    ['C', [ 1.2222370, -0.7056590, 0.0000000]],
    ['C', [ 0.0000000, -1.4113170, 0.0000000]],
    ['C', [-1.2222370, -0.7056590, 0.0000000]],
    ['C', [-1.2222370,  0.7056590, 0.0000000]],
    ['H', [ 0.0000000,  2.5070120, 0.0000000]],
    ['H', [ 2.1711360,  1.2535060, 0.0000000]],
    ['H', [ 2.1711360, -1.2535060, 0.0000000]],
```

(continues on next page)

(continued from previous page)

```
    ['H', [ 0.0000000, -2.5070120, 0.0000000]],
    ['H', [-2.1711360, -1.2535060, 0.0000000]],
    ['H', [-2.1711360,  1.2535060, 0.0000000]]
]
g = GeometryMolecular(xyz)
visualizer = VisualizerNGL(g)
visualizer.visualize_molecule(atom_labels="index")
```

```
NGLWidget()
```

16.4.2 Visualizing Fragments

There are several fragmentation methods available in inquanto, to visualize a fragmentation defined in a `Geometry` object, call the `.visualize_fragmentation()` method as illustrated in the code-snippet below.

[ ]:
```
g.set_groups(
    "fragments",
    {
        "ch1": [0, 6],
        "ch2": [5, 11],
        "ch3": [4, 10],
        "ch4": [3, 9],
        "ch5": [2, 8],
        "ch6": [1, 7]
    }
)
visualizer.visualize_fragmentation("fragments", atom_labels="index")
```

```
NGLWidget()
```

16.4.3 Visualizing Orbitals

To aid in active space selection, it is also possible to view the molecular orbitals of a system if the appropriate `.cube` strings are available.

[ ]:
```
from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF

driver = ChemistryDriverPySCFMolecularRHF(geometry=g.xyz, basis="sto3g")

cube_orbitals=driver.get_cube_orbitals()
ngl_mos = [visualizer.visualize_orbitals(orb, atom_labels="index") for i, orb in␣
↪enumerate(cube_orbitals)]
ngl_mos[16]
```

```
NGLWidget()
```

16.5 Tutorial 5 - A basic VQD simulation with InQuanto

In this file, we will introduce the methodology for finding electronic excited state energies using the Variational Quantum Deflation (VQD) approach in inquanto. For the original publication by Higgot, Wang and Brierley (2019) and more technical details, please go here.

In VQE, one finds a minimum of the energy by classically optimising the function below with respect to the wavefunction parameters, $\{\lambda\}$:

$E(\lambda) = \langle\psi(\lambda)|H|\psi(\lambda)\rangle = \sum_j c_j \langle\psi(\lambda)|P_j|\psi(\lambda)\rangle,$

where P_j are terms in the qubit Hamiltonian, and c_j are classically pre-computed coefficients. In contrast, when executing a VQD simulation, the objective function is modified to include a penalty term, multiplied by a weight β,

$F(\lambda_k) = \langle\psi(\lambda_k)|H|\psi(\lambda_k)\rangle + \sum_{i=0}^{k-1} \beta_i |\langle\psi(\lambda_k)|P_j|\psi(\lambda_i)\rangle|^2.$

This enforces the requirement that each $|\psi(\lambda_k)\rangle$ be orthogonal to the other $|\psi(\lambda_0)\rangle, \ldots |\psi(\lambda_{k-1})\rangle$ found by previous optimizations of the objective function, $F(\lambda_k)$.

In order to run a VQD calculation, one needs several things: 1. A molecular, electronic Hamiltonian operator, H, 2. A mapping object for constructing the qubit Hamiltonian, 3. An ansatz for computing the ground state energy, $|\psi(\lambda)\rangle$, 4. A classical minimizer 5. A complete VQE experiment, 6. A second ansatz for describing the excited states, $\{|\psi(\lambda_k)\rangle\}$, 7. A expression for evaluating the weights, $\{\beta_i\}$, 8. The number of excited states of interest, k.

We are concerned with first finding the ground state energy of the second-quantized molecular electronic Hamiltonian using the Variational Quantum Eigensolver (please see tutorials 1, and 2 for a more in-depth explanation). So let's proceed with points 1-4 in the list above.

First, we need to import a backend, and the appropriate space and state objects. Since we are looking at fermions, these are FermionSpace and FermionState. We will use the AerBackend available through the pytket qiskit extension to simulate the quantum hardware.

[ ]:
```
from pytket.extensions.qiskit import AerBackend
from inquanto.spaces import FermionSpace
from inquanto.states import FermionState
```

We're going to simulate the dihydrogen molecule in the STO-3G basis. There are 4 spin orbitals – two of which are occupied – and the reference state lives in 4-dimensional Fock space. The corresponding objects can be constructed as below:

[ ]:
```
from inquanto.express import load_h5
from inquanto.mappings import QubitMappingJordanWigner
h2 = load_h5("h2_sto3g.h5")
fermion_hamiltonian = h2["hamiltonian_operator"]
qubit_hamiltonian = QubitMappingJordanWigner().operator_map(fermion_hamiltonian)

space = FermionSpace(4)
state = FermionState([1, 1, 0, 0])

print(fermion_hamiltonian.print_table())
```

```
  0.7430177069924179 +
 -1.2702927243904387 F0^ F0  +
-0.45680735030940944 F2^ F2  +
 -1.2702927243904387 F1^ F1  +
-0.45680735030940944 F3^ F3  +
 0.33428863850674484 F2^ F0^ F0  F2  +
-0.08983433978150786 F2^ F0^ F0  F2  +
-0.08983433978150784 F2^ F0^ F0  F2  +
```

(continues on next page)

(continued from previous page)

```
 0.33428863850674473 F2^ F0^ F0  F2  +
 0.33428863850674484 F3^ F1^ F1  F3  +
-0.08983433978150786 F3^ F1^ F1  F3  +
-0.08983433978150784 F3^ F1^ F1  F3  +
 0.33428863850674473 F3^ F1^ F1  F3  +
   0.340030928792064 F1^ F0^ F0  F1  +
 0.33428863850674484 F2^ F1^ F1  F2  +
 0.08983433978150784 F1^ F0^ F2  F3  +
-0.08983433978150786 F2^ F1^ F0  F3  +
-0.08983433978150784 F3^ F0^ F1  F2  +
 0.08983433978150783 F3^ F2^ F0  F1  +
 0.33428863850674473 F3^ F0^ F0  F3  +
  0.3514067666381408 F3^ F2^ F2  F3  +
   0.340030928792064 F1^ F0^ F0  F1  +
 0.33428863850674484 F3^ F0^ F0  F3  +
 0.08983433978150784 F1^ F0^ F2  F3  +
-0.08983433978150786 F3^ F0^ F1  F2  +
-0.08983433978150784 F2^ F1^ F0  F3  +
 0.08983433978150783 F3^ F2^ F0  F1  +
 0.33428863850674473 F2^ F1^ F1  F2  +
  0.3514067666381408 F3^ F2^ F2  F3
None
```

where we have loaded in the molecular Hamiltonian using inquanto's express module, and mapped it to a qubit Hamiltonian using the Jordan-Wigner transformation.

Now, for points 3 and 4 in the list, we need to construct an ansatz for our ground state calculation and choose a classical minimizer. For this example we will use the k-UpCCGSD ansatz and the COBYLA minimizer available through the MinimizerScipy object.

[ ]:
```
from inquanto.ansatzes import FermionSpaceAnsatzkUpCCGSD
from inquanto.minimizers import MinimizerScipy

ansatz = FermionSpaceAnsatzkUpCCGSD(space, state, k_input=2)
minimizer = MinimizerScipy(method="COBYLA")
```

We're now in a position to address item 5 in the list - running a complete VQE calculation. We know from the first equation in this notebook that the objective function is the expectation value of the qubit Hamiltonian.

[ ]:
```
from inquanto.computables import ExpectationValue
from inquanto.algorithms import AlgorithmVQE

expectation_value = ExpectationValue(ansatz, qubit_hamiltonian)
vqe = AlgorithmVQE(
    objective_expression=expectation_value,
    minimizer=minimizer,
    initial_parameters=ansatz.state_symbols.construct_random(seed=0)
)
```

We have passed in some random $\{\lambda\}$ using the state_symbols attribute of the ansatz object as our starting guess.

We now choose our measurement protocol – in this case, a direct measurement by operator averaging, so we choose ProtocolDirect – and the number of shots we wish to simulate in each iteration. Then, we build the algorithm object and execute.

[ ]:
```
from inquanto.protocols import ProtocolDirect
```

(continues on next page)

(continued from previous page)

```
vqe.build(backend=AerBackend(), protocol_expression=ProtocolDirect(), n_shots=10000)

vqe.run(seed=0)

print("VQE Energy:    ", vqe.final_value)
print("VQE Parameters:", vqe.final_parameters.to_array())
```

```
# TIMER BLOCK-0 BEGINS AT 2023-03-23 15:42:35.462133
# TIMER BLOCK-0 ENDS - DURATION (s): 75.6954587 [0:01:15.695459]
VQE Energy:     -1.1350359651397368
VQE Parameters: [ 1.64162074 -1.83379232  0.09597707  1.4768844  -1.31100542 -0.
↪04794482]
```

According to point 6, we now need a second, deflationary ansatz we can use to describe the excited states. To do this, we'll use the same ansatz structure and just make a copy of the first ansatz. We update our symbols from those used in the ground state to some other symbols. Consider this as constructing the symbols in $\{\lambda_k\}$ using those in $\{\lambda\}$ as a reference.

[ ]:
```
ansatz_2 = ansatz.subs("{}_2")
```

It is almost time to construct, build and execute our VQD algorithm. First, we need to write expressions corresponding to the terms in the functional

$F(\lambda_k) = \langle\psi(\lambda_k)|H|\psi(\lambda_k)\rangle + \sum_{i=0}^{k-1} \beta_i |\langle\psi(\lambda_k)|P_j|\psi(\lambda_i)\rangle|^2.$

We will refer to the leading term as expectation_value, the weights as weight_expression, and the overlap term in the penalty as overlap_expression.

We will also select the weights as the expectation value of the deflationary ansatz with respect to the sign-flipped qubit Hamiltonian to ensure it is sufficiently large to act as a constraint rather than a weak penalty.

[ ]:
```
from inquanto.computables import OverlapSquared

expectation_value = ExpectationValue(ansatz_2, qubit_hamiltonian)

weight_expression = ExpectationValue(ansatz_2, -1 * qubit_hamiltonian)

overlap_expression = OverlapSquared(ansatz, ansatz_2)
```

As was the case with the previous VQE experiment, we must choose protocols for measuring the overlaps. For this we choose to use the vacuum test, available through the ProtocolVacuum object.

We must also choose the number of excited states we wish to find. For this calculation, we'll choose to find 3 and pass this into the AlgorithmVQD constructor in the n_vectors argument.

[ ]:
```
from inquanto.algorithms import AlgorithmVQD
from inquanto.protocols import ProtocolVacuum
# instantiate VQD object
vqd = AlgorithmVQD(
    objective_expression=expectation_value,
    overlap_expression=overlap_expression,
    weight_expression=weight_expression,
    minimizer=MinimizerScipy(method="COBYLA"),
    initial_parameters=ansatz_2.state_symbols.construct_random(seed=0),
    vqe_value=vqe.final_value,
    vqe_parameters=vqe.final_parameters,
    n_vectors=3,
)
```

(continues on next page)

(continued from previous page)

```
# build object
vqd.build(
    backend=AerBackend(),
    objective_protocol=ProtocolDirect(),
    weight_protocol=ProtocolDirect(),
    overlap_protocol=ProtocolVacuum(),
    n_shots=1000,
)

# execute
vqd.run(seed=0)

# print results
print("VQD excited state energies:   ", vqd.final_values)
```

```
# TIMER BLOCK-1 BEGINS AT 2023-03-23 16:16:44.261628
# TIMER BLOCK-1 ENDS - DURATION (s): 24.8339346 [0:00:24.833935]
# TIMER BLOCK-2 BEGINS AT 2023-03-23 16:17:09.096227
# TIMER BLOCK-2 ENDS - DURATION (s): 46.7267220 [0:00:46.726722]
# TIMER BLOCK-3 BEGINS AT 2023-03-23 16:17:55.823494
# TIMER BLOCK-3 ENDS - DURATION (s): 86.7953335 [0:01:26.795333]
VQD excited state energies:    [-1.1350359651397368, -0.4967777718439771, -0.
↪13598654953456038, 0.5580064536976894]
```

[ ]:

CHAPTER

SEVENTEEN

HARDWARE TUTORIALS

This documentation also contains tutorials detailing explicitly how simulations may be run on quantum device hardware, as well as demonstrations of noise mitigation methods.

IBMQ - 1	Running experiments on the Aer simulator	`download`
IBMQ setup	How to set up IBM Quantum credentials and backends	`PDF version`
IBMQ - 2	Running experiments on the remote emulator and hardware	`download`

17.1 IBMQ - 1 - Running experiments on the Aer simulator

In this two part tutorial we demonstrate the process of taking a simple quantum chemical computation and performing it on IBM hardware.

As this tutorial focuses on practical quantum computation, we're going to perform a simple calculation: the single point (i.e. not optimised/not variationally solved) total energy evaluation of the H2 molecule in the Unitary Coupled Cluster (UCC) ansatz for a set value of the ansatz variational parameter.

The *second part* of this tutorial will require an IBM Quantum account and the credentials to run on a (at least) 4 qubit machine for a short amount of time. Please see the linked *page* for instructions for details on how to set this up.

The steps below are such:

Notebook 1

- Define the system
- Perform noise-free simulation (AerStateBackend)
- Perform stochastic simulation (AerBackend)
- Perform simulation with simple noisy simulation of the quantum computation (AerBackend + custom noise profile)

Notebook 2

- Redefine the system
- Perform computation with emulated hardware noise (IBMQEmulatorBackend + machine noise profile)
- Demonstrate error mitigation methods on emulated hardware (+ SPAM + PMSV)
- Perform computation on hardware (IBMQBackend)

17.1.1 0. System preparation

To begin, we use the `express` module to load in the converged mean-field (Hartree-Fock) spin-orbitals, potential, and Hamiltonian from a calculation of H2 using the STO-3G basis set.

[ ]:
```
%reset -f
from inquanto.express import load_h5
h2 = load_h5("h2_sto3g.h5", as_tuple=True)
hamiltonian = h2.hamiltonian_operator

import warnings
warnings.filterwarnings('ignore')
```

Now we prepare the Fermionic space and define the Fermionic state.

The space here is defined with 4 spin-orbitals (which matches the full H2 STO-3G space) and we use the D2h point group.

This point group is the most practical high symmetry group to approximate the D(infinity)h group. We also explicitly define the orbital symmetries.

The state is then set by the ground state occupations [1,1,0,0] and the Hamiltonian encoded from the Hartree-fock integrals.

A space of excited states is then created using the UCCSD ansatz, which is then mapped to a quantum circuit using Jordan-Wigner (JW) encoding. InQuanto uses an efficient ansatz circuit compilation approach here, provided by the `FermionSpaceStateExpJWChemicallyAware` class, to reduce the computational resources required.

[ ]:
```
from inquanto.spaces import FermionSpace
from inquanto.states import FermionState
from inquanto.symmetry import PointGroup
from inquanto.ansatzes import FermionSpaceStateExpChemicallyAware

space = FermionSpace(
    4, point_group=PointGroup("D2h"), orb_irreps=["Ag", "Ag", "B1u", "B1u"]
)

state = FermionState([1, 1, 0, 0])
qubit_hamiltonian = hamiltonian.qubit_encode()

exponents = space.construct_single_ucc_operators(state)
## the above adds nothing due to the symmetry of the system
exponents += space.construct_double_ucc_operators(state)
ansatz = FermionSpaceStateExpChemicallyAware(exponents, state)
```

At this point, the circuit structure is well defined, but the guess for the wave function parameters (weights of the exponentiated determinants / gate angles of rotation gates) is not set. We set these parameters with a single value as 'p'.

With the parameters defined, the Computable module is used to specify the quantity of interest: we are asking to evaluate the expectation value of the Hamiltonian for some given parameter value θ, $E =< \Psi(\theta)|\hat{H}|\Psi(\theta) >$.

For demonstration purposes, we fix the random seed on the initialisation of the parameters using `seed=6`, which should set a parameter value of 0.499675. Alternatively, this parameter can be set using `p = ansatz.state_symbols.construct_from_array([0.4996755931358105])`.

[ ]:
```
#p = ansatz.state_symbols.construct_random(seed=6)
p = ansatz.state_symbols.construct_from_array([0.4996755931358105])
print(p.df().iloc[0,2])
```

(continues on next page)

(continued from previous page)

```
from inquanto.computables import ExpectationValue
expectation0 = ExpectationValue(ansatz, hamiltonian.qubit_encode())
```

```
0.4996755931358105
```

With the circuit and parameters defined, the circuit can be displayed and analysed. This circuit has 4 qubits, with 31 gates. The gates which induce the most noise are the multi-qubit gates – in this gate decomposition, the CNOT gates. This circuit has 4 CNOT gates.

[ ]:
```
from pytket.circuit.display import render_circuit_jupyter
render_circuit_jupyter(ansatz.get_circuit(p))

from pytket import Circuit, OpType
print('\nCNOT GATES:  {}'.format(ansatz.state_circuit.n_gates_of_type(OpType.CX)))

print(ansatz.state_circuit)
```

```
<IPython.core.display.HTML object>
```

```
CNOT GATES:  4
<tket::Circuit, qubits=4, gates=31>
```

We are now ready to think about how we run this circuit.

To help understand the effect of noise in computing this circuit, in the next cell we define a set of 'shots' to scan over, as well as a random seed number, and some other parameters. To demonstrate the different stochasticity and quantum noise in backends, we will present convergence plots, showing how the average energy converges towards the expectation value with more sampling.

[ ]:
```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (12,4)

set_shots= [10,20,50,100,200,500,1000,2000,5000,10000,20000,50000]
#N_shots * N_measured_terms is the total number of shots
#n_shots arg is number of samples per commuting set formed from the Hamiltonian terms
set_seed=1
```

17.1.2 1. Noiseless StateVector simulation

The evaluation of a quantum measurement can be carried out directly by state vector methods, which performs the linear algebra of gate operations directly to an explicit 2^N dimensional representation of the quantum state. This method returns the computable of the system directly without considering the stochasticity of the quantum measurements at the end of the circuit. The number of shots here is arbitrary as the computable result is returned directly rather than averaged over sampling the computational basis eigenvalues. For the given system parameter ($\theta = 0.499676$), we expect an energy of -0.587646 Ha.

In general, to define where the computation is performed we set the backend, in this case `backend = AerStateBackend()`.

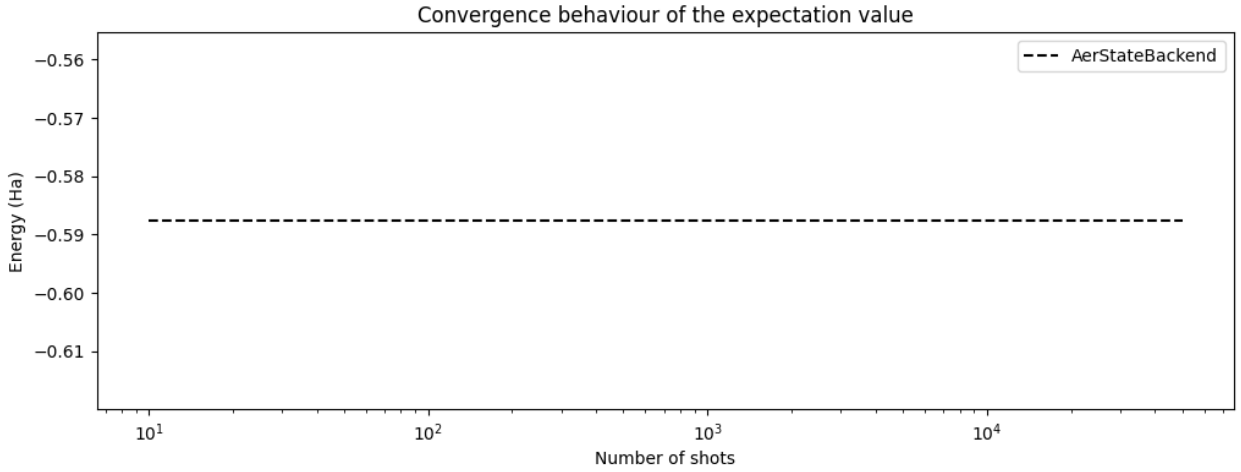
[ ]:
```
from matplotlib import pyplot as plt
import numpy as np

from pytket.extensions.qiskit import AerStateBackend
```

(continues on next page)

(continued from previous page)

```
from inquanto.protocols import ProtocolStateVectorBackendSupport

backend = AerStateBackend()

aer_state_expectation = ExpectationValue(ansatz, hamiltonian.qubit_encode())
aer_state_expectation.build([ProtocolStateVectorBackendSupport()])

shotless_energies=[]
for i in set_shots:
    aer_state_expectation.run(backend, p, n_shots=i, seed=set_seed) ## note the seed here
    shotless_energies.append(aer_state_expectation.evaluate())

plt.plot(set_shots, shotless_energies , label='AerStateBackend', color='black', ls='--')
plt.xscale("log")
plt.xlabel('Number of shots')
plt.ylabel('Energy (Ha)')
plt.title('Convergence behaviour of the expectation value')
plt.legend()
print('Statevector energy is ' +str(np.real(shotless_energies[-1]))+'Ha')
```

```
Statevector energy is -0.5876463677224998Ha
```



17.1.3 2. Noiseless simulation

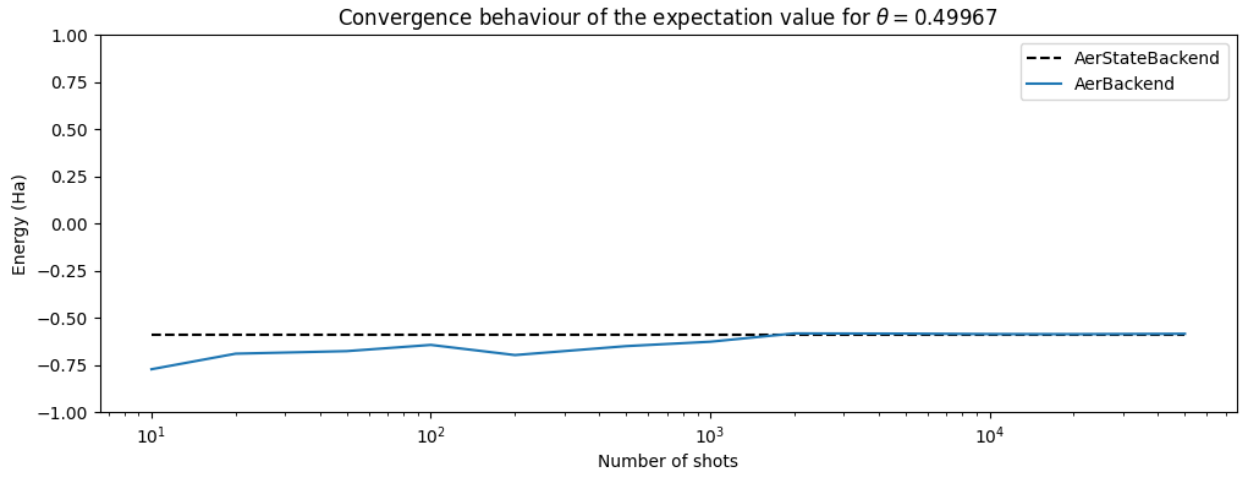
Next, we can simulate 'shots' (runs of the quantum computer) using the AerBackend. This simulation does not have any quantum noise (e.g. due to the decoherence of the quantum state during processing), but does include the stochastic nature of measuring the system, with probabilistic collapse into a computational basis state at the end of the simulation. Each individual shot will return a measurement outcome of +1 or -1 on each qubit. By taking many shots, one can estimate the expectation value of each Pauli string within the Hamiltonian, with increasing shot counts resulting in increased precision on each expectation value. This results in an increased precision of the expectation value of the full Hamiltonian.

We produce a plot to show the convergence of the energy with the number of shots. This shows that at about 1000 shots, the system averaging is good enough that there is a reasonable recreation of the AerStateBackend result (within 0.01 Ha).

We recommend changing the seed value in `aer_expectation.run` to examine different convergence regimes. Consider what number of shots is required for consistent precision.

```
[ ]: from pytket.extensions.qiskit import AerBackend
     from inquanto.protocols import ProtocolDirect

     backend=[]
     backend = AerBackend()

     aer_expectation = ExpectationValue(ansatz, hamiltonian.qubit_encode())
     aer_expectation.build([ProtocolDirect()])

     noiseless_energies= []
     for i in set_shots:
         aer_expectation.run(backend, p, n_shots=i, seed=set_seed) ## note the seed here
         noiseless_energies.append(aer_expectation.evaluate())

     plt.plot(set_shots, shotless_energies , label='AerStateBackend', color='black', ls='--
     ↪')
     plt.plot(set_shots, noiseless_energies, label='AerBackend')
     plt.xscale("log")
     plt.ylim([-1, 1])
     plt.xlabel('Number of shots')
     plt.ylabel('Energy (Ha)')
     plt.title('Convergence behaviour of the expectation value for ' +r'$\theta=$'+str(p.
     ↪df().iloc[0,2])[0:7])
     plt.legend()

     print(noiseless_energies[-1])
```

```
-0.584831900031038
```



17.1.4 3. Simple quantum noise model

Having demonstrated the stochastic nature of quantum measurement, we are ready to consider quantum noise.

To do this, first we will use a simple quantum noise model. The noise in quantum circuits can manifest in many ways. A simple noise model is just to add depolarising error to CNOT gates. This can be a reasonable first approximation as multi-qubit operations generally cause the most quantum noise.

There are many other types of quantum noise that can be added, some of which are detailed in the Qiskit documentation . One other simple example is readout error, which adds a chance to incorrectly measure the qubit state at the end of the circuit.

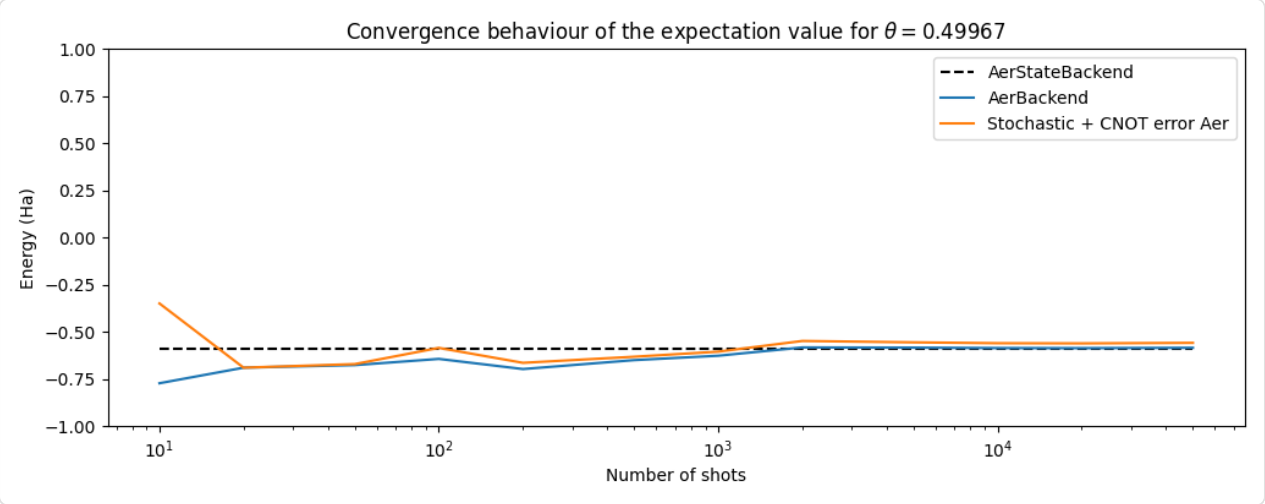
We recommend modifying the `cnot_error_rate` parameter and also the seed number in `noisy_aer_expectation.run()`. In particular, consider whether for larger CNOT error rate (>0.01) the system does or does not practically converge to the AerStateBackend result.

[ ]:
```
## In this cell we define the noise model
from qiskit.providers.aer.noise import NoiseModel #, ReadoutError
import qiskit.providers.aer.noise as noise

cnot_error_rate=0.01
noise_model = NoiseModel()
error_1 = noise.depolarizing_error(cnot_error_rate, 2)

n_qubits = 4
for qubit in range(n_qubits):
    for qubit2 in (x for x in range(n_qubits) if x != qubit) :
        noise_model.add_quantum_error(error_1, ['cx'],[qubit,qubit2])

print(noise_model.is_ideal())  ## this reports false if there is noise in the model
```

```
False
```

[ ]:
```
## This cell runs the cnot noisy simulation

noisy_backend = AerBackend(noise_model=noise_model) # this defaults to no noise,␣
↪which is the same as NoiseModel.is_ideal
noisy_aer_expectation = ExpectationValue(ansatz, hamiltonian.qubit_encode())
noisy_aer_expectation.build([ProtocolDirect()])

noisy_energies= []
for i in set_shots:
    noisy_aer_expectation.run(noisy_backend, p, n_shots=i, seed=set_seed) ## note the␣
↪seed here
    noisy_energies.append(noisy_aer_expectation.evaluate())

plt.plot(set_shots, shotless_energies , label='AerStateBackend', color='black', ls='--
↪')
plt.plot(set_shots, noiseless_energies, label='AerBackend')
plt.plot(set_shots, noisy_energies, label='Stochastic + CNOT error Aer')
plt.xscale("log")
plt.ylim([-1, 1])
plt.xlabel('Number of shots')
plt.ylabel('Energy (Ha)')
plt.title('Convergence behaviour of the expectation value for ' +r'$\theta=$'+str(p.
↪df().iloc[0,2])[0:7])
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7ff1e17b5340>
```



This tutorial has demonstrated the use of freely available qiskit backends to compute a simple quantum system. The final example adds a simple model of quantum noise, and in the *second part* of this tutorial we present the use of a more complex and realistic noise model, the use of noise mitigation, and the performance of the experiment on an IBMQ quantum device.

Part 2 will require an IBM Quantum account and the credentials to run on a (at least) 4 qubit machine for a short amount of time. Please see the linked *page* for instructions for details on how to set this up.

17.2 IBMQ - 2 - Running experiments on the remote emulator and hardware

In this two part tutorial we demonstrate the process of taking a simple quantum chemical computation and performing it on IBM hardware.

The *first part* of this tutorial detailed setting up the simple molecular calculation and performing state vector and simple stochastic simulations.

This part will require an IBM Quantum account and the credentials to run on a at least 4 qubit machine for a short amount of time. Please see the linked *page* for instructions for details on how to set this up.

The steps below are such:

Notebook 2

- Redefine the system
- Perform computation with emulated hardware noise IBMQEmulatorBackend + machine noise profile)
- Demonstrate error mitigation methods on emulated hardware + SPAM + PMSV)
- Perform computation on hardware IBMQBackend)

In *part 1* we demonstrated

- Perform noise-free simulation (AerStateBackend)
- Perform stochastic simulation (AerBackend)
- Perform simulation with simple noisy simulation of the quantum computation (AerBackend + custom noise profile)

17.2.1 0. Redefine system

More details on this are contained in *part 1*.

[23]:
```
%reset -f
from inquanto.express import load_h5
h2 = load_h5("h2_sto3g.h5", as_tuple=True)
hamiltonian = h2.hamiltonian_operator

from inquanto.spaces import FermionSpace
from inquanto.states import FermionState
from inquanto.symmetry import PointGroup
from inquanto.ansatzes import FermionSpaceStateExpChemicallyAware

space = FermionSpace(
    4, point_group=PointGroup("D2h"), orb_irreps=["Ag", "Ag", "B1u", "B1u"]
)

state = FermionState([1, 1, 0, 0])
qubit_hamiltonian = hamiltonian.qubit_encode()

exponents = space.construct_single_ucc_operators(state)
## the above adds nothing due to the symmetry of the system
exponents += space.construct_double_ucc_operators(state)
ansatz = FermionSpaceStateExpChemicallyAware(exponents, state)

p = ansatz.state_symbols.construct_from_array([0.4996755931358105])
#print(p.df().iloc[0,2])

from inquanto.computables import ExpectationValue
expectation0 = ExpectationValue(ansatz, hamiltonian.qubit_encode())

from pytket.circuit.display import render_circuit_jupyter
#render_circuit_jupyter(ansatz.get_circuit(p))

from pytket import Circuit, OpType
#print('\nCNOT GATES:  {}'.format(ansatz.state_circuit.n_gates_of_type(OpType.CX)))

#print(ansatz.state_circuit)

import warnings
warnings.filterwarnings('ignore')
```

[24]:
```
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (12,4)

set_shots= [10,20,50,100,200,500,1000,2000,5000,10000,20000,50000]
#N_shots * N_measured_terms is the total number of shots
#n_shots arg is number of samples per commuting set formed from the Hamiltonian terms
set_seed=1
```

[1]:
```
## load in results from tutorial part 1 for theta = 0.499676

shotless_energies=[-0.5876463677224993, -0.5876463677224993, -0.5876463677224993, -0.
↪5876463677224993, -0.5876463677224993, -0.5876463677224993, -0.5876463677224993, -0.
```

(continues on next page)

(continued from previous page)

```
↪5876463677224993, -0.5876463677224993, -0.5876463677224993, -0.5876463677224993, -0.
↪5876463677224993]
noiseless_energies=[-0.772864962255895, -0.690378841063184, -0.676820904260161, -0.
↪643826455783076, -0.697646323948843, -0.650294528247345, -0.626969894185408, -0.
↪582917543331619, -0.584005407703986, -0.586257848062735, -0.586787891168772, -0.
↪584831900031038]
noisy_energies=[-0.349815706924837, -0.690378841063184, -0.671325314696797, -0.
↪585373354739910, -0.664662878297064, -0.630624301489494, -0.605379049027431, -0.
↪549060997786899, -0.555803530409928, -0.560443191700409, -0.561319370059358, -0.
↪558820274056905]
```

17.2.2 4. Machine emulation for quantum noise

From this point on, running the simulations will require IBM Quantum credentials and an API Token

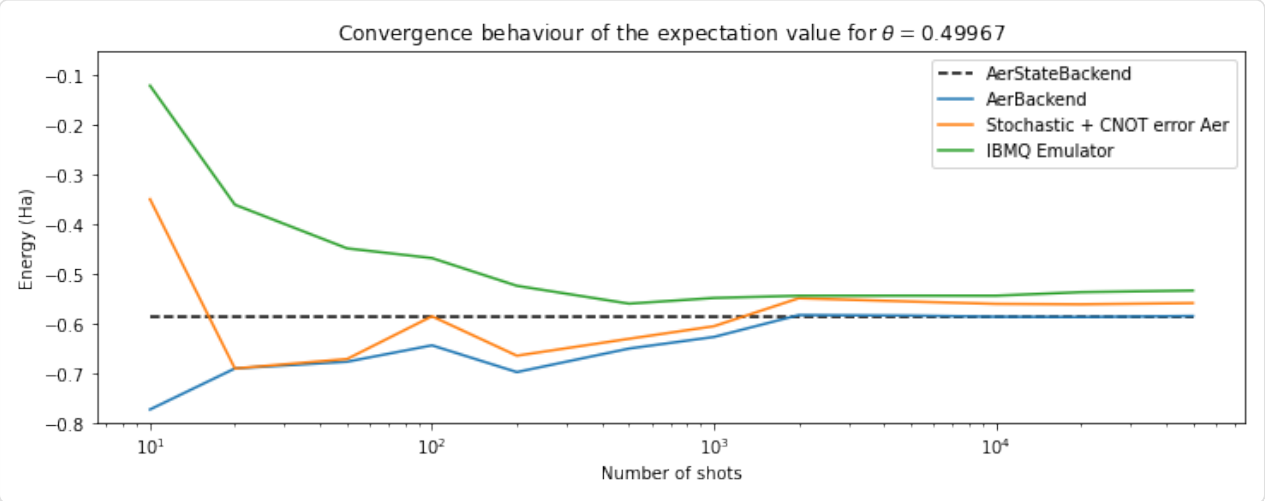
Instructions on how to obtain these credentials are detailed on another *page*.

In the last section we used a single simple noise parameter, but generally quantum computers experience many different types of noise. The amount of each kind of noise varies from device to device, for example IBMQ_Manila may not have the same readout error as IBM_Lagos, and different devices are designed to do some specific operations better than others. The specific noise profiles of machines can be loaded in and applied to simulations using `IBMQEmulatorBackend`, which pulls information from your IBMQ account.

Simulating an actual machine is a key step towards understanding the resources you will need to run on hardware.

[26]:
```
from pytket.extensions.qiskit import set_ibmq_config
set_ibmq_config(ibmq_api_token='')

from pytket.extensions.qiskit import IBMQEmulatorBackend
backend = IBMQEmulatorBackend(backend_name="", hub="", group="", project="")


from inquanto.protocols import ProtocolDirect
noisy_ibmq_expectation=ExpectationValue(ansatz, hamiltonian.qubit_encode())
noisy_ibmq_expectation.build([ProtocolDirect()])

noisy_ibmq_energies=[]
for i in set_shots:
    noisy_ibmq_expectation.run(backend, p, n_shots=i, seed=set_seed)
    noisy_ibmq_energies.append(noisy_ibmq_expectation.evaluate())

plt.plot(set_shots, shotless_energies , label='AerStateBackend', color='black', ls='--
↪')
plt.plot(set_shots, noiseless_energies, label='AerBackend')
plt.plot(set_shots, noisy_energies, label='Stochastic + CNOT error Aer')
plt.plot(set_shots, noisy_ibmq_energies, label='IBMQ Emulator')
plt.xscale("log")
plt.ylim([-0.8, -0.05])
plt.xlabel('Number of shots')
plt.ylabel('Energy (Ha)')
plt.title('Convergence behaviour of the expectation value for ' +r'$\theta=$'+str(p.
↪df().iloc[0,2])[0:7])
plt.legend()
```

[26]:
```
<matplotlib.legend.Legend at 0x7f8c0aa2e940>
```



In this case, the noisier (and more realistic) IBMQ Emulator noise model leads to an increase in energy and poorer accuracy than the simple CNOT error model.

17.2.3 5. Noise mitigation methods in IBMQ emulation

To address the noise in the IBMQ device, we can utilise noise mitigation methods.

In this case we will define the Qubit Operator symmetries in the system so that we may utilise PMSV (Partition Measurement Symmetry Verification). This reduces experimental noise by removing shots wherein a symmetry-breaking error has occurred.

We will also utilise a SPAM (State Preparation And Measurement) correction, which calibrates the calculation for the noise in the system.

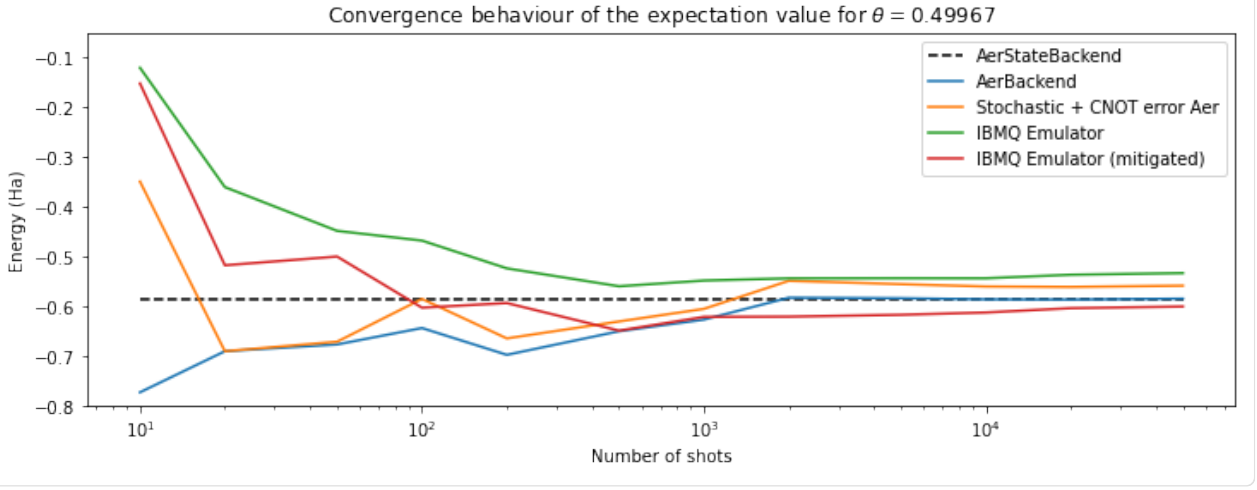
[27]:
```
from pytket.extensions.qiskit import IBMQEmulatorBackend
backend = IBMQEmulatorBackend(backend_name="", hub="", group="", project="")

from inquanto.protocols.supporters import SupporterPMSV, SupporterSPAM
from inquanto.operators import QubitOperator
symmetries = [
    QubitOperator("Z0 Z2", -1),
    QubitOperator("Z1 Z3", -1),
    QubitOperator("Z2 Z3", +1),
]
# these symmetries can also be loaded from express h2_sto3g_symmetry.h5

spam = SupporterSPAM(backend.backend_info.nodes)
spam.calibrate(backend, n_shots=500)
pmsv = SupporterPMSV(symmetries)

from inquanto.protocols import ProtocolDirect
miti_ibmq_expectation=ExpectationValue(ansatz, hamiltonian.qubit_encode())
miti_ibmq_expectation.build([ProtocolDirect().apply(spam).apply(pmsv)]) ## try using␣
↪just one of these

miti_ibmq_energies=[]
for i in set_shots:
    miti_ibmq_expectation.run(backend, p, n_shots=i, seed=set_seed)
```

(continues on next page)

(continued from previous page)

```
    miti_ibmq_energies.append(miti_ibmq_expectation.evaluate())

plt.plot(set_shots, shotless_energies , label='AerStateBackend', color='black', ls='--
↪')
plt.plot(set_shots, noiseless_energies, label='AerBackend')
plt.plot(set_shots, noisy_energies, label='Stochastic + CNOT error Aer')
plt.plot(set_shots, noisy_ibmq_energies, label='IBMQ Emulator')
plt.plot(set_shots, miti_ibmq_energies, label='IBMQ Emulator (mitigated)')
plt.xscale("log")
plt.ylim([-0.8, -0.05])
plt.xlabel('Number of shots')
plt.ylabel('Energy (Ha)')
plt.title('Convergence behaviour of the expectation value for ' +r'$\theta=$'+str(p.
↪df().iloc[0,2])[0:7])
plt.legend()
```

[27]:
```
<matplotlib.legend.Legend at 0x7f8bb88fce50>
```



Again we recommend modifying the random seed of the cell above, and also mixing and matching the use of SPAM and PMSV. The end result should be that, even for this simple system with a shallow circuit, the energy should converge with fewer shots and to greater accuracy using the noise mitigation techniques.

17.2.4 6. Running on hardware (unmitigated)

InQuanto makes replacing realistic machine simulation results with hardware experiments quite easy. This is because the `pytket.extensions.qiskit` requires the same details for emulation of hardware as for sending the experiment to the physical quantum circuit, the only difference is replacing `IBMQEmulatorBackend` with `IBMQBackend`.

The cell below will run on a quantum computing device when given an IBMQ API Token and an accessible 4+ qubit machine (see *instructions*). By default it will run 5 jobs sequentially, with an increasing number of shots so that we may again examine convergence. Each job will queue and run on the device seperately (i.e. send job, queue, run, return results, process, send next job..).

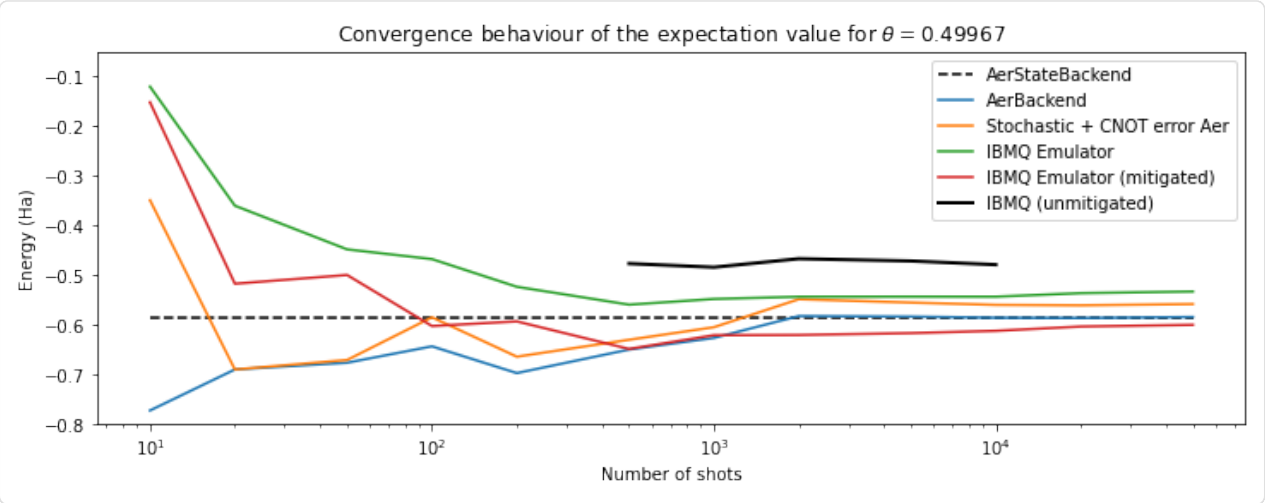
Unlike the previous simulated examples, where the random seed is fixed for increasing number of shots, the experiments will have their own (real) stochasticity and quantum noise. This is why the convergence behaviour will not appear as smooth.

[28]:
```
hw_shots=[500, 1000, 2000, 5000, 10000]
import time

from pytket.extensions.qiskit import IBMQBackend
backend = IBMQBackend(backend_name="", hub="", group="", project="")

spam = SupporterSPAM(backend.backend_info.nodes)
spam.calibrate(backend, n_shots=500)
pmsv = SupporterPMSV(symmetries)

ibmq_hw_expectation=ExpectationValue(ansatz, hamiltonian.qubit_encode())
ibmq_hw_expectation.build([ProtocolDirect()]) #.apply(spam).apply(pmsv)]) # apply␣
↪these if needed

ibmq_hw_energies=[]
for i in hw_shots:
    ibmq_hw_expectation.run(backend, p, n_shots=i) #, seed=set_seed)
    ibmq_hw_energies.append(ibmq_hw_expectation.evaluate())
    print(ibmq_hw_expectation.evaluate())
    time.sleep(5) # small delay to help sync with device
```

```
Job Status: job has successfully run
Job Status: job has successfully run
-0.477315986985964
Job Status: job has successfully run
-0.484732340338749
Job Status: job has successfully run
-0.467600408855598
Job Status: job has successfully run
-0.472133223388001
Job Status: job has successfully run
-0.479563375829108
```

[29]:
```
# these are example solved energies from hardware

stored_hw_energies=[-0.4729481410092342, -0.501901444890315, -0.482607756234069, -0.
↪507372069185149, -0.495002371125095]

plt.plot(set_shots, shotless_energies , label='AerStateBackend', color='black', ls='--
↪')
plt.plot(set_shots, noiseless_energies, label='AerBackend')
plt.plot(set_shots, noisy_energies, label='Stochastic + CNOT error Aer')
plt.plot(set_shots, noisy_ibmq_energies, label='IBMQ Emulator')
plt.plot(set_shots, miti_ibmq_energies, label='IBMQ Emulator (mitigated)')
plt.plot(hw_shots, ibmq_hw_energies, label='IBMQ (unmitigated)', color='black', lw=2)
plt.xscale("log")
plt.ylim([-0.8, -0.05])
plt.xlabel('Number of shots')
plt.ylabel('Energy (Ha)')
plt.title('Convergence behaviour of the expectation value for ' +r'$\theta=$'+str(p.
↪df().iloc[0,2])[0:7])
plt.legend()
```

[29]:
```
<matplotlib.legend.Legend at 0x7f8b340adeb0>
```



In the above figure we compare all results for our fixed parameter, having successfully run on IBMQ hardware (Perth). The hardware may not be quite as good as other methods due to the real noise present, and this can be improved using mitigation techniques. These can be added by re-running the cells above and uncommenting out the spam and pmsv options.

We recommend exploring the convergence and values in these methods by modifying random seeds, including the wave function parameter θ - for example, re-run the notebooks and establish its optimum value.

17.3 IBMQ Setup

To use or emulate the IBM quantum computing devices the user will need to provide *pytket.extensions.qiskit* with an API token from IBM Quantum.

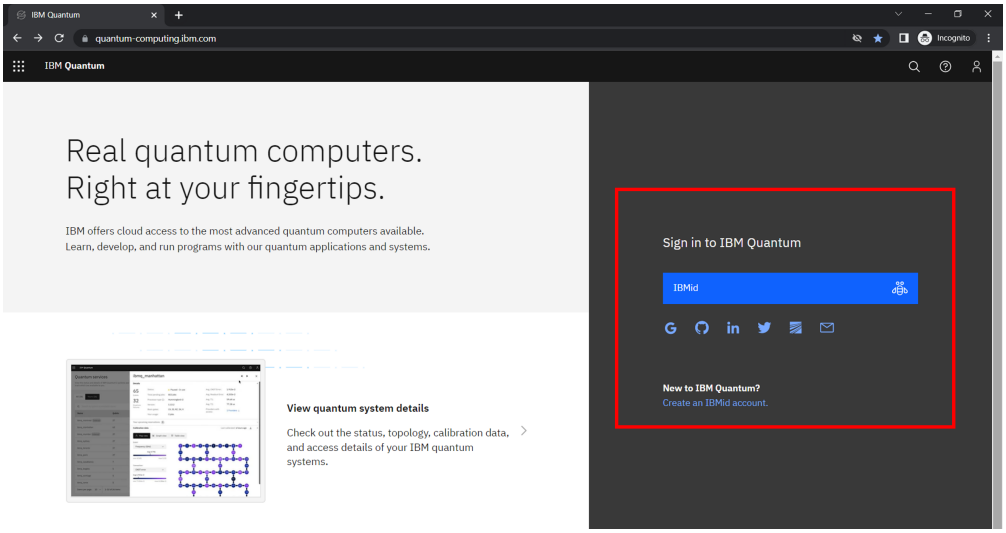
Anyone can make and use IBM resources, but free users will only have access to a few small devices.

To begin, make sure you have both InQuanto and *pytket.extensions.qiskit* in your Python environment (Use `pip install pytket-qiskit` if needed).

17.3.1 Create an IBMQ account

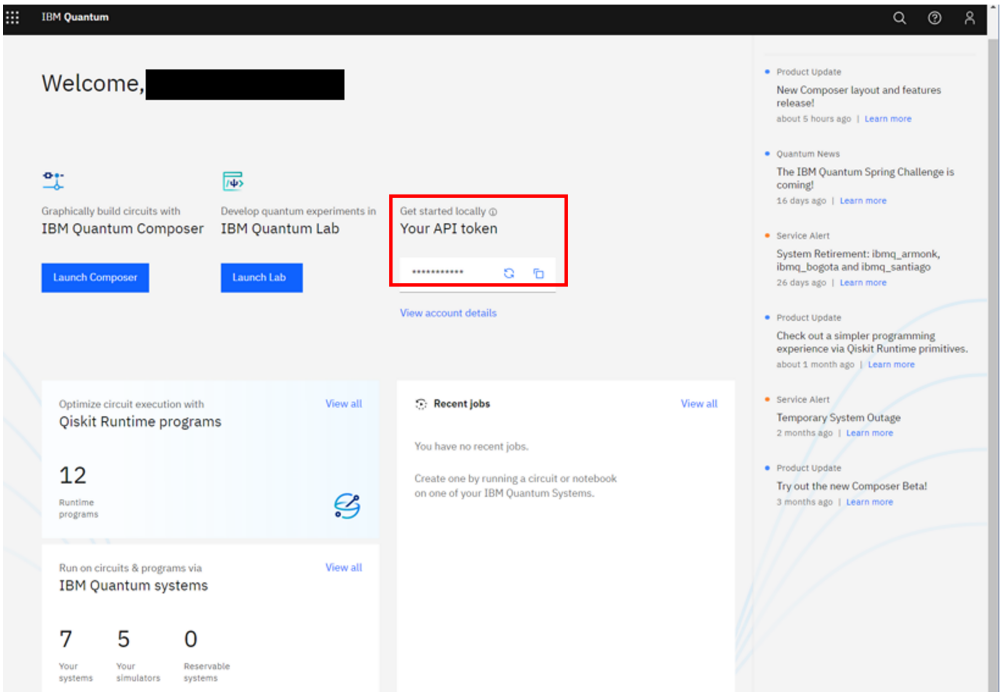
Go to: https://quantum-computing.ibm.com/

Either use an auxiliary login (e.g. Google) or follow instructions for IBMid creation



If using IBMid, fill out the user input details, complete the 2-factor authentication, log-in, and agree to their EULA.

On logging in to https://quantum-computing.ibm.com/ the user should be presented with the following home-screen:



17.3.2 Get IBMQ API token

The API token can be easily copied from the IBMQ home-screen (highlighted in red above).

To store the IBMQ API token for use with InQuanto, open a python shell or notebook and use:

```
from pytket.extensions.qiskit import set_ibmq_config
set_ibmq_config(ibmq_api_token='XXXX')
```

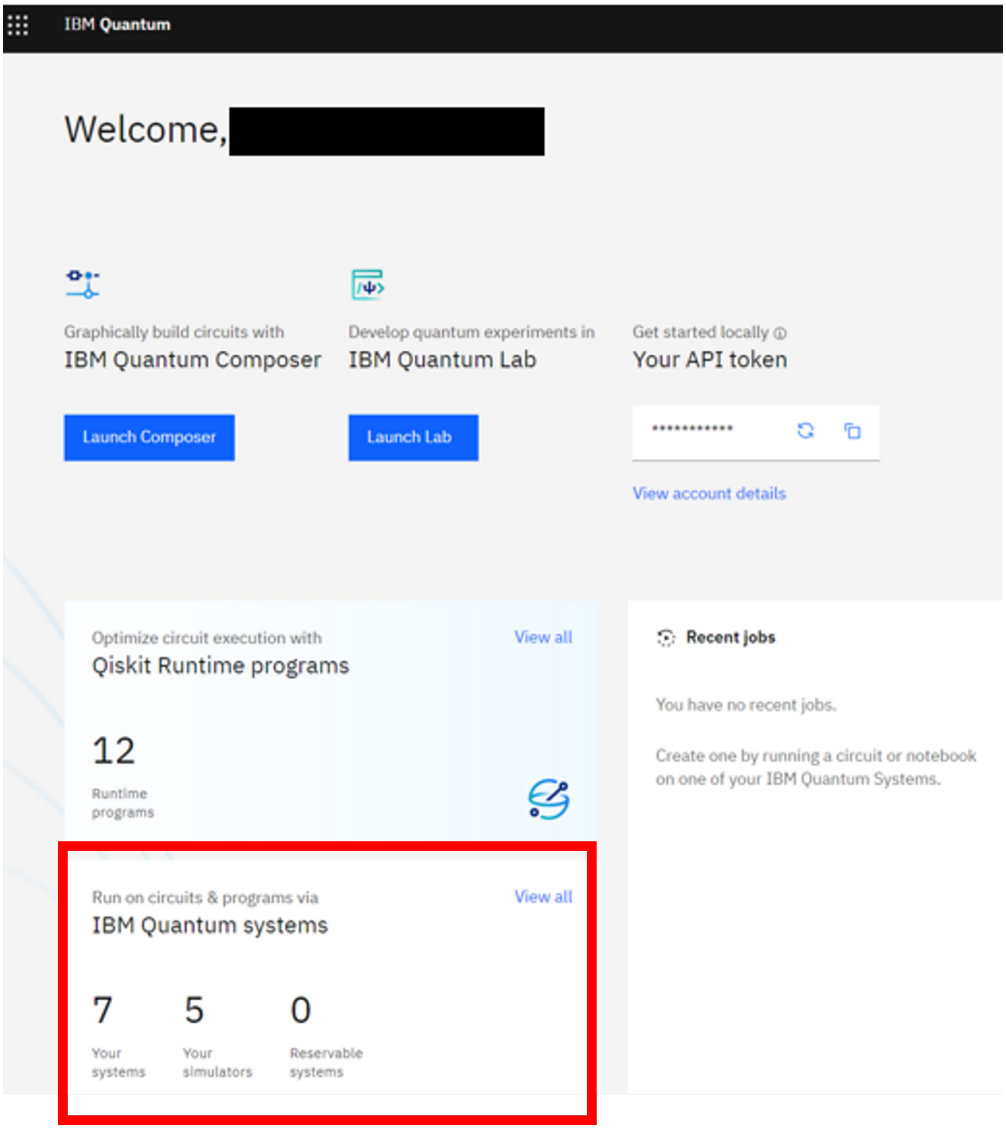
where 'XXXX' is the API Token copied. Note that generating a new API token (♲) will stop the old token working

As well as the API token, the user will need to specify a machine to emulate. This is detailed below.

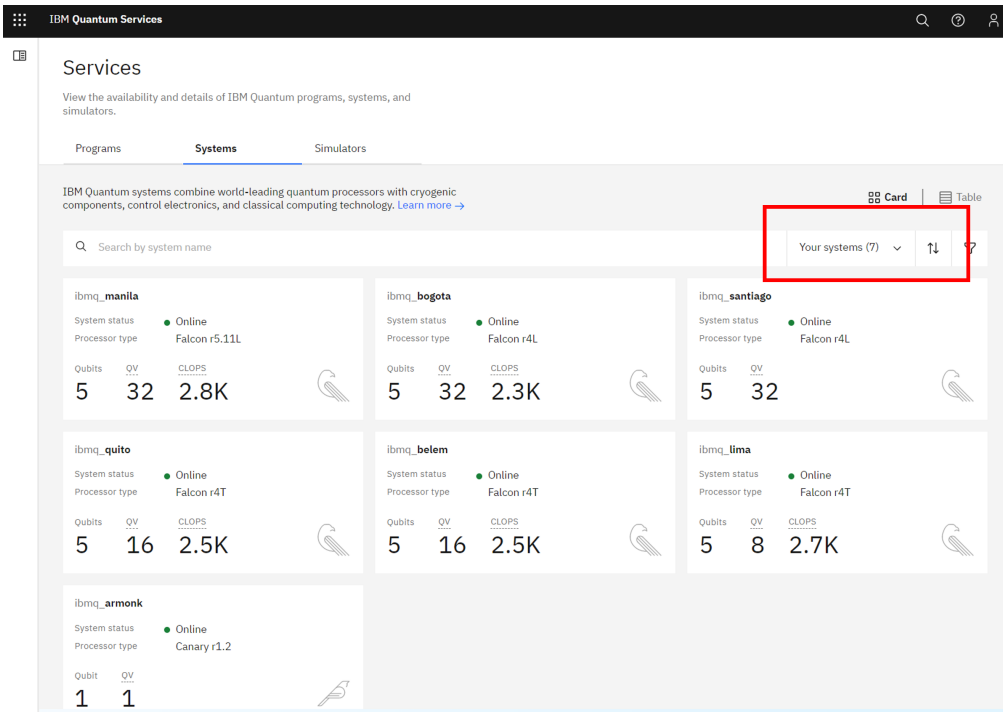
17.3.3 Choosing a quantum device

With the API token set, we can choose a machine to run on/simulate

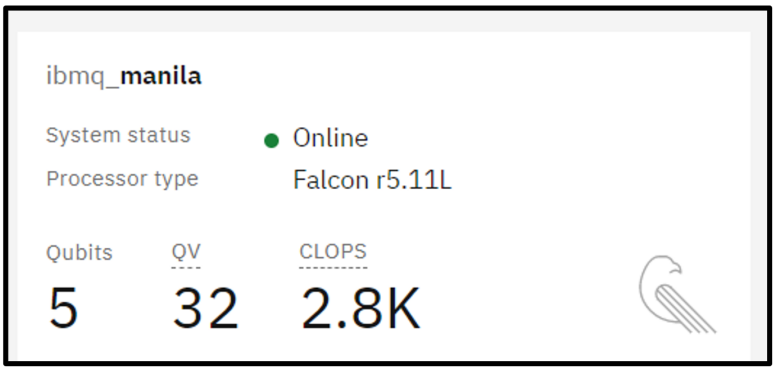
The list of available machines can be found by clicking 'View all' on the IBM Quantum systems section of the IBMQ home page

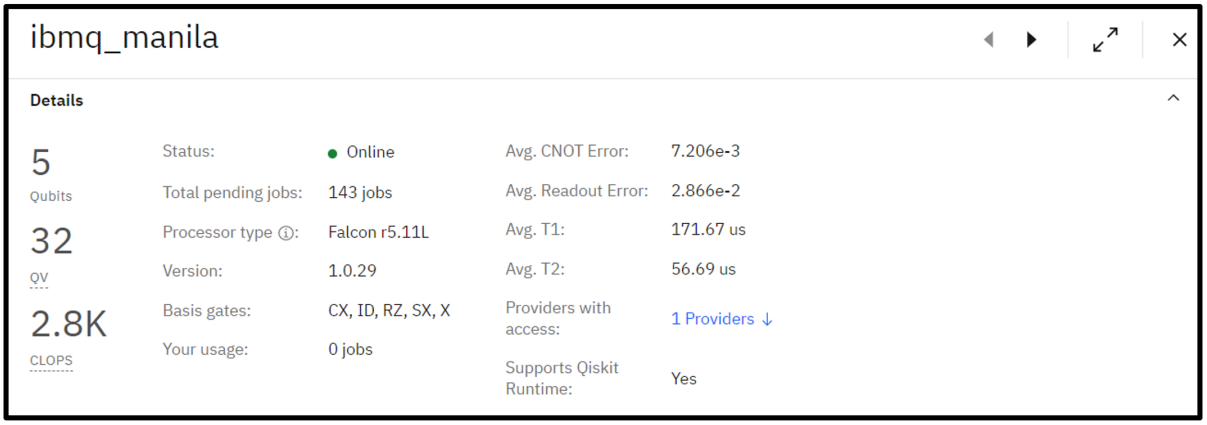


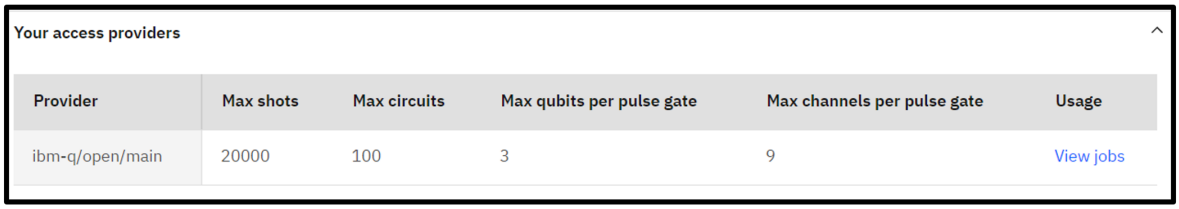
When on the device page, the list can be filtered to show only devices available to the user.



Clicking on a listed machine will show the user many details about that machine. For example; the gate error, or the number of jobs queueing to run on it.

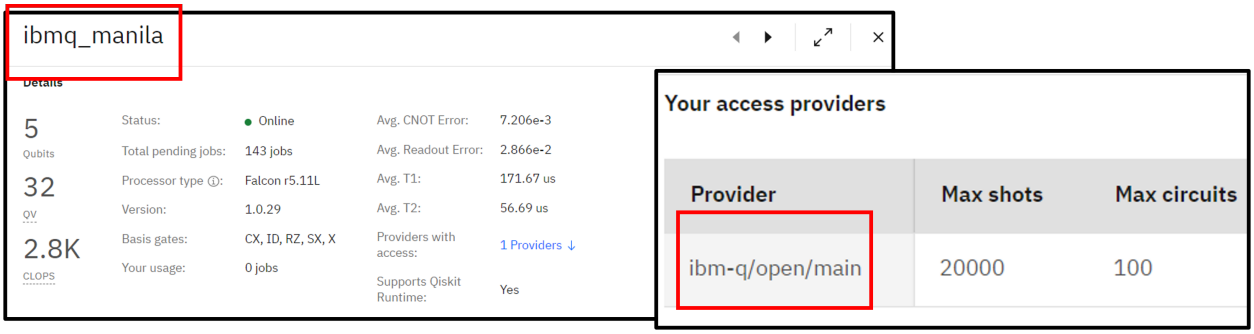






To get the machine details needed for computing, scroll down or click the 'Providers with access' link.

In your python shell or notebook, the machine details can be set, for example using:




```
from pytket.extensions.qiskit import IBMQEmulatorBackend
backend = IBMQEmulatorBackend(backend_name="ibmq_manila", hub="ibm-q", group="open",␣
↪project="main")
```

In the example above we have set up an emulation of shots on the 5 qubit IBMQ Manila machine using the 'free' queue.

To run a calculation on the physical quantum device, simply change *IBMQEmulatorBackend* to *IBMQBackend*. However, when running hardware experiments on a free queue, please be considerate of your usage and other users. Another key point is that when submitting hardware experiments, the user will need to keep the python kernel running until results have been returned and processed.

CHAPTER

EIGHTEEN

OVERVIEW OF EXAMPLES

In addition to the detailed *tutorials*, InQuanto contains several example scripts showing how various functionality is used. In this file we provide a broad overview of the intention of each example, highlighting the functionality that is demonstrated.

18.1 examples/algorithms/adapt

Examples of ADAPT-VQE.

file	description
`algorithm_fermionic_adapt.py`	A simulation of H2 in STO-3G using the ADAPT algorithm with fermionic helper functions.
`algorithm_iqeb.py`	A simulation of H2 in STO-3G using IQEB algorithm.
`algorithm_adapt.py`	A simulation of H2 in STO-3G using the ADAPT algorithm.

18.2 examples/algorithms/qse

Examples of Quantum Subspace Expansion.

file	description
`algorithm_qse.py`	A simulation of H2 in STO-3G using the QSE algorithm

18.3 examples/algorithms/time_evolution

Examples of time evolution algorithms.

file	description
vqs_imag_example.py	A AlgorithmMcLachlanImagTime time evolution simulation
vqs_real_example.py	A AlgorithmMcLachlanRealTime time evolution simulation
time_evolution_example.py	An exact time evolution simulation
vqs_real_example_paper_phased.py	A custom VQS time evolution simulation.
vqs_real_example_paper.py	AlgorithmMcLachlanRealTime time evolution simulation for a small system.
imtime_evolution_example.py	An exact imaginary time evolution simulation

18.4 examples/algorithms/vqd

Examples of Variational Quantum Deflation.

file	description
algorithm_vqd.py	AlgorithmVQD using computables example.

18.5 examples/algorithms/vqe

Examples of canonical Variational Quantum Eigensolver usage.

file	description
algorithm_vqe_uccsd.py	A canonical VQE simulation of H2 in STO-3G using a UCCSD Ansatz.
algorithm_vqe_hea.py	A canonical VQE simulation of H2 in STO-3G using a hardware-efficient Ansatz.

18.6 examples/ansatzes

Examples demonstrating usage of Ansatz classes.

file	description
layered_hea.py	Use of a layered hardware efficient Ansatz.
fermion_space_ansatze.py	Use of a general Fermionic Ansatz.

18.7 examples/computables

Examples demonstrating usage of computables classes.

file	description
qse_computable.py	An example running QSE using computables
computable_hermitian_expect py	Computable expression example (ExpectationValue)
rdm_computable.py	An example for constructing an RDM using computables
computable_h2_symbolic_shor py	Use of computables classes for STO-3G H2 expectation value measurement with symbolic quick evaluation.
computable_h2_symbolic.py	Use of computables classes for STO-3G H2 expectation value measurement with symbolic protocol.
computable_h2_saved_handles py	Use of computables classes for STO-3G H2 expectation value measurement.
examples_derivatives.py	An example showing how to use evaluate gradients
computable_h2_symbolic_deri py	Use of computables classes for STO-3G H2 expectation value and gradients with symbolic protocol.
statevector_accelerated_vqe py	LiH VQE simulation with and without gradients to demonstrate acceleration
computable_h2.py	Use of computables classes for STO-3G H2 expectation value measurement.
examples_finite_difference. py	An example showing how to use evaluate gradients and compare against finite differences.
computables_and_protocols. py	Basic computables and protocols
computable_h2_short.py	Use of computables classes for STO-3G H2 expectation value measurement.

18.8 examples/core

Examples demonstrating usage of logging and debugging functionality.

file	description
context_printing.py	Using context for logging standard outputs

18.9 examples/embeddings

Examples of usage of embedding methods.

file	description
dmet_one_h2x3_express_vqe.py	One-shot DMET calculations on hydrogen rings.
impurity_dmet_h2x3_express.py	An Impurity DMET example for simulating a 3-dihydrogen ring.
fmo_h2x3_integral_operator_V.py	An example FMO simulation of a 3-dihydrogen chain with custom VQE fragment solver.
impurity_dmet_h2x3_express_v.py	An example impurity DMET simulation of a 3-dihydrogen ring using the express module.
impurity_dmet_ch2.py	An example for running impurity DMET on triplet CH2.
dmet_full_h2x3_express_hf.py	Full DMET calculations on hydrogen rings.
dmet_one_h2x3_express_hf.py	One-shot DMET calculations on hydrogen rings.
impurity_dmet_h7.py	An ImpurityDMET example simulating a 7-hydrogen chain.
dmet_one_phenol_vqe.py	An example DMET calculation simulating phenol.

18.10 examples/express

Examples of usage of the built-in example molecular data in InQuanto.

file	description
get_started_express.py	Basic code snippets to use express, operators and states.
h5_operations.py	Built-in example molecular data.

18.11 examples/mappings

Examples of fermion-qubit mapping functionality.

file	description
mapping_bk.py	Use of the Bravyi-Kitaev mapping from fermions to qubits.
mapping_paraparticular.py	Use of the paraparticle mapping from fermions to qubits.
mapping_jw.py	Use of the Jordan-Wigner mapping from fermions to qubits.

18.12 examples/minimizers

Examples of classical minimizers in InQuanto.

file	description
minimizer_scipy_rotosolve.py	Use of Scipy and Rotosolve minimizers.
example_integrators.py	Symbolic evaluation for integrators.

18.13 examples/operators

Examples of functionality of operator and state classes.

file	description
chemistry_integral_operator:py	Creation of ChemistryRestrictedIntegralOperator and conversion to FermionOperator.
orbital_transformer.py	Orbital transformation methods.
orbital_optimizer.py	Orbital optimization using Pipek-Mezey as an example.
fermion_operator.py	Examples of some FermionOperator methods.
qubit_state.py	Construction of QubitState and demonstration of some functionality.
fermion_state.py	Creation of FermionState objects and demonstration of some functionality.
qubit_operator.py	Construction of QubitOperator and demonstration of some functionality.

18.14 examples/spaces

Examples of space classes for generating operators and states.

file	description
fermion_spaces.py	Use of FermionSpace objects to generate fermionic states and operators, including point group symmetry.
parafermion_space.py	Use of ParaFermionSpace objects to generate parafermionic states and operators.

18.15 examples/symmetry

Examples of symmetry classes and functionality.

file	description
qubit_tapering.py	Qubit tapering - operators and Ansatzae.
qubit_symmetry_operators.py	Finding qubit Z2 symmetry operators.
point_group.py	Use of PointGroup class containing point group symmetry information.
fermionic_symmetry_operators.py	Finding fermionic Z2 symmetry operators.

CHAPTER

NINETEEN

INQUANTO-EXTENSIONS

Several interfaces to third-party chemistry programs are available for InQuanto, known as InQuanto extensions.

Currently, these extensions consist of:

- *InQuanto-PySCF*
- *InQuanto-NGLView*

CHAPTER

TWENTY

INQUANTO-PYSCF

The *inquanto-pyscf* extension provides an interface to PySCF, a classical computational chemistry package. This extension allows a user to input chemistry information to be processed by using classical computational methods, and construct the data to be mapped onto quantum computers.

20.1 Basic usage

The main feature of this extension is to construct the fermionic second-quantized Hamiltonian in the molecular orbital basis, along with the fermionic Fock space and state by running a classical computational mean-field (Hartree-Fock) calculation. An *inquanto-pyscf* driver may be used to set up the calculation, and the *get_system()* method generates all of these ingredients, shown below for an H_2 molecule at 0.75 Å separation, with the STO-3G atomic basis set:

```
"""Minimal basis H2."""

from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF

# Initialize the chemistry driver.
driver = ChemistryDriverPySCFMolecularRHF(
    geometry="H 0 0 0; H 0 0 0.75",
    basis="sto3g",
)

# Get the data to be mapped onto quantum computers.
hamiltonian, fock_space, fock_state = driver.get_system()

print("Fock state:")
fock_space.print_state(fock_state)
print("Hamiltonian:")
print(hamiltonian.df())
```

```
Fock state:
 0 0a         :  1
 1 0b         :  1
 2 1a         :  0
 3 1b         :  0
Hamiltonian:
    Coefficients            Terms
0       0.705570
1      -1.247285          F0^ F0
2       0.336424  F1^ F0^ F0  F1
3       0.336424  F1^ F0^ F0  F1
```

(continues on next page)

(continued from previous page)

```
4       0.090886  F1^ F0^ F2  F3
5       0.090886  F1^ F0^ F2  F3
6      -1.247285          F1^ F1
7      -0.090886  F2^ F0^ F0  F2
8      -0.090886  F2^ F0^ F0  F2
9       0.330989  F2^ F0^ F0  F2
10      0.330989  F2^ F0^ F0  F2
11     -0.090886  F2^ F1^ F0  F3
12     -0.090886  F2^ F1^ F0  F3
13      0.330989  F2^ F1^ F1  F2
14      0.330989  F2^ F1^ F1  F2
15     -0.481273          F2^ F2
16      0.330989  F3^ F0^ F0  F3
17      0.330989  F3^ F0^ F0  F3
18     -0.090886  F3^ F0^ F1  F2
19     -0.090886  F3^ F0^ F1  F2
20     -0.090886  F3^ F1^ F1  F3
21     -0.090886  F3^ F1^ F1  F3
22      0.330989  F3^ F1^ F1  F3
23      0.330989  F3^ F1^ F1  F3
24      0.090886  F3^ F2^ F0  F1
25      0.090886  F3^ F2^ F0  F1
26      0.347908  F3^ F2^ F2  F3
27      0.347908  F3^ F2^ F2  F3
28     -0.481273          F3^ F3
```

With the Hamiltonian, Fock space, and Hartree-Fock state, one may construct the quantum problem as usual (see *here* to get started).

Above, the `hamiltonian` is an *integral operator* of type *ChemistryRestrictedIntegralOperator*. For molecular systems, PySCF drivers may produce more specialized integral operators. For example, with the `symmetry` option we generate a *ChemistryRestrictedIntegralOperatorCompact* object, which stores symmetry-compacted representations of the two-body integral tensor for reducing memory requirements:

```
compact_hamiltonian, space, state = driver.get_system(symmetry=8)  # Maximum symmetry␣
↪reduction for restricted spins
```

Or, with the *get_system_ao()* method, we get a *PySCFChemistryRestrictedIntegralOperator*, which wraps a PySCF `SCF` object:

```
pyscf_hamiltonian, space, state = driver.get_system_ao()
```

The latter case is similarly useful for reducing classical memory requirements, and is also an important component in Fragment Molecular Orbital (FMO) calculations (see the *embedding sections* below).

20.2 InQuanto-PySCF driver classes

A variety of drivers are available for different systems and applications, many of which are covered in more detail in the following sections. A full list of all drivers is given below:

For molecular systems:

- *inquanto.extensions.pyscf.ChemistryDriverPySCFMolecularRHF*
- *inquanto.extensions.pyscf.ChemistryDriverPySCFMolecularROHF*

- *inquanto.extensions.pyscf.ChemistryDriverPySCFMolecularUHF*
- *inquanto.extensions.pyscf.ChemistryDriverPySCFMolecularRHFQMMMCOSMO*
- *inquanto.extensions.pyscf.ChemistryDriverPySCFMolecularROHFQMMMCOSMO*
- *inquanto.extensions.pyscf.ChemistryDriverPySCFMolecularUHFQMMMCOSMO*
- *inquanto.extensions.pyscf.ChemistryDriverPySCFEmbeddingRHF*
- *inquanto.extensions.pyscf.ChemistryDriverPySCFEmbeddingROHF*
- *inquanto.extensions.pyscf.ChemistryDriverPySCFEmbeddingROHF_UHF*

For periodic systems:

- *inquanto.extensions.pyscf.ChemistryDriverPySCFGammaRHF*
- *inquanto.extensions.pyscf.ChemistryDriverPySCFGammaROHF*
- *inquanto.extensions.pyscf.ChemistryDriverPySCFMomentumRHF*
- *inquanto.extensions.pyscf.ChemistryDriverPySCFMomentumROHF*

20.3 Active space specification and AVAS

It is standard practice to select an orbital active space to reduce the resource requirements of quantum chemistry calculations. The active space may be specified manually using the `frozen` argument in an *inquanto-pyscf* driver. Below we consider the example of an H_2 O molecule with the 6-31G basis set. The full Fock space is given by:

```
"""H2O with 6-31G basis set."""

from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF

h2o_geom = """
O          0.00000        0.00000        0.11779;
H          0.00000        0.75545       -0.47116;
H          0.00000       -0.75545       -0.47116;"""

# Initialize the chemistry driver.
driver = ChemistryDriverPySCFMolecularRHF(
    geometry=h2o_geom,
    basis="631g",
)
hamiltonian, fock_space, fock_state = driver.get_system()
print("Fock state:")
fock_space.print_state(fock_state)
```

```
Fock state:
 0 0a         :  1
 1 0b         :  1
 2 1a         :  1
 3 1b         :  1
 4 2a         :  1
 5 2b         :  1
 6 3a         :  1
 7 3b         :  1
 8 4a         :  1
 9 4b         :  1
```

(continues on next page)

(continued from previous page)

```
10 5a          :  0
11 5b          :  0
12 6a          :  0
13 6b          :  0
14 7a          :  0
15 7b          :  0
16 8a          :  0
17 8b          :  0
18 9a          :  0
19 9b          :  0
20 10a         :  0
21 10b         :  0
22 11a         :  0
23 11b         :  0
24 12a         :  0
25 12b         :  0
```

We may freeze molecular orbitals by passing a list of frozen orbital indices:

```
"""H2O with 6-31G basis set with 4-orbital 4-electron active space."""
driver = ChemistryDriverPySCFMolecularRHF(
    geometry=h2o_geom,
    basis="631g",
    frozen=[0, 1, 2, 7, 8, 9, 10, 11, 12],
)
hamiltonian, fock_space, fock_state = driver.get_system()

print("Fock state (reduced by setting the active space):")
fock_space.print_state(fock_state)
```

```
Fock state (reduced by setting the active space):
 0 0a          :  1
 1 0b          :  1
 2 1a          :  1
 3 1b          :  1
 4 2a          :  0
 5 2b          :  0
 6 3a          :  0
 7 3b          :  0
```

or, equivalently, by using the *inquanto.extensions.pyscf.FromActiveSpace* callable class, where we specify the number of active spatial orbitals and electrons:

```
from inquanto.extensions.pyscf import FromActiveSpace

driver = ChemistryDriverPySCFMolecularRHF(
    geometry=h2o_geom,
    basis="631g",
    frozen=FromActiveSpace(ncas=4, nelecas=4),
)
hamiltonian, fock_space, fock_state = driver.get_system()

print("Fock state (reduced by setting the active space):")
fock_space.print_state(fock_state)
```

```
Fock state (reduced by setting the active space):
 0 0a          :  1
 1 0b          :  1
 2 1a          :  1
 3 1b          :  1
 4 2a          :  0
 5 2b          :  0
 6 3a          :  0
 7 3b          :  0
```

InQuanto also supports the use of Atomic Valence Active Space (AVAS), a technique for selecting an active space for a localized domain of a molecule. AVAS constructs the active space by projecting the target molecular orbitals onto a set of atomic orbitals.

To use AVAS, instantiate the *inquanto.extensions.pyscf.AVAS* class with the atomic orbital labels, and build the PySCF driver as follows:

```
from inquanto.extensions.pyscf import AVAS
avas= AVAS(aolabels=['Li 2s', 'H 1s'], threshold=0.8, threshold_vir=0.8, verbose=5)

driver = ChemistryDriverPySCFMolecularRHF(
    geometry='Li 0 0 0; H 0 0 1.75',
    basis='631g',
    transf=avas,
    frozen=avas.frozenf
)
hamiltonian, fock_space, fock_state = driver.get_system()
print("Fock state (reduced by setting the active space with AVAS):")
fock_space.print_state(fock_state)
```

```

```

```
******** AVAS flags ********
```

```
aolabels = ['Li 2s', 'H 1s']
```

```
frozen = []
```

```
minao = minao
```

```
threshold = 0.8
```

```
threshold (virtual) = 0.8
```

```
with_iao = False
```

```
force_halves_active = False
```

```
canonicalize = True
```

```

** AVAS **
```

```
Total number of electrons: 4.0
```

```
  Total number of HF MOs  is equal to    11
```

```
  Number of occupied HF MOs is equal to  2
```

```
reference AO indices for minao ['Li 2s', 'H 1s']: [1 2]
```

```
(full basis) reference AO indices for 631g ['Li 2s', 'H 1s']: [1 9]
```

```
Threshold 0.8, threshold_vir 0.8
```

```
Active from occupied = 1 , eig [0.96867343]
```

```
Inactive from occupied = 1
```

```
Active from unoccupied = 1 , eig [0.9854409]
```

```
Inactive from unoccupied = 8
```

```
Number of active orbitals 2
```

```
# of alpha electrons 1
```

```
# of beta electrons 1
```

```
Fock state (reduced by setting the active space with AVAS):
 0 0a         :  1
 1 0b         :  1
 2 1a         :  0
 3 1b         :  0
```

where the `transf` argument specifies the AVAS orbital transformation, and the `AVAS.frozenf` attribute returns the list of frozen orbitals.

20.4 Classical post-HF calculations

Classical post-HF energies are useful to compare against quantum computational results for quick benchmarking. Each chemistry driver has an interface to the post-HF calculators provided by PySCF:

```
driver = ChemistryDriverPySCFMolecularRHF(
    geometry="H 0 0 0; H 0 0 0.75",
    basis="631g",
)
driver.get_system()

# Classical Post-HF energies for benchmarking.
print('HF\t', driver.mf_energy)
print('MP2\t', driver.run_mp2())
print('CCSD\t', driver.run_ccsd())
print('CASCI\t', driver.run_casci())
```

```
HF          -1.1265450345356913
MP2          -1.144034783436534
```

```
CCSD          -1.1516885473648517
CASCI          -1.151688547516609
```

20.5 Generating a driver from a PySCF object

Experienced PySCF users may want to start their calculations with a PySCF mean-field object directly, before preparing the quantum calculation. The *from_mf()* method initializes an *inquanto-pyscf* driver in this way, from which we can prepare the hamiltonian, Fock space, and Fock state as usual:

```
"""Generating a driver from a pyscf mean-field object."""

from pyscf import gto, scf

# PySCF calculation.
mol = gto.Mole(
    atom='H 0 0 0; H 0 0 0.75',
    basis='sto3g',
    verbose=0,
).build()
mf = scf.RHF(mol)
mf.kernel()

# Initialize InQuanto driver
driver = ChemistryDriverPySCFMolecularRHF.from_mf(mf)
hamiltonian, fock_space, fock_state = driver.get_system()
```

Warning: Although this interface is useful in some cases, it is recommended to use the standard driver interface where possible. This advanced feature may cause a runtime error, as it is difficult to guarantee that the wrapper is fully consistent with all PySCF options.

20.6 Periodic systems

Extended solid-state simulations at any level of dimensionality reduction can be performed by taking advantage of PySCF's periodic boundary conditions (PBC) capabilities. The *inquanto-pyscf* extension has two types of *drivers* for periodic calculations: Gamma point drivers, for $k = 0$ calculations, and momentum space drivers, where a k-point grid is specified.

Below, we show a Gamma point example with the *ChemistryDriverPySCFGammaROHF* driver for a Pd (111) surface with a 2x2x1 slab and 20Å of vacuum:

```
from pyscf.pbc import gto as pgto
from inquanto.extensions.pyscf import ChemistryDriverPySCFGammaRHF

cell_pd221=[
    [5.50129075763134, 0.0, 0.0],
    [2.75064537881567, 4.764257549713281, 0.0],
    [0.0, 0.0, 20.0]
]
```

(continues on next page)

(continued from previous page)

```
geometry_pd221=[
    ['Pd', [ 0.,  0., 10.]],
    ['Pd', [ 2.75064538,  0.        , 10.        ]],
    ['Pd', [ 1.37532269,  2.38212877, 10.        ]],
    ['Pd', [ 4.12596807,  2.38212877, 10.        ]]
]

driver_pd221 = ChemistryDriverPySCFGammaRHF(
    basis="lanl2dz",
    ecp="lanl2dz",
    geometry=geometry_pd221,
    charge=0,
    cell=cell_pd221,
    dimension=2, # SLAB 2D PBC system
    df="GDF",
    output=None,
    verbose=1,
    exp_to_discard = 0.1
)
print("Hartree-Fock energy: ", driver_pd221.run_hf())
```

```
Hartree-Fock energy:  -503.28280280926344
```

where we have also used Gaussian density fitting (GDF) to reduce the computational cost of operations with the two-body electronic integrals (see here). From this point, one may use the *get_system()* method as usual to build the fermionic hamiltonian and construct the quantum problem.

Calculations with a non-trivial k-point grid proceed similarly. Below, we consider an H2 chain (1D system) with PBC and a [4,1,1] k-point grid:

```
from inquanto.extensions.pyscf import ChemistryDriverPySCFMomentumRHF

cell_h2 = pgto.Cell()
cell_h2.atom=[['H', [0., 5., 5.]], ['H', [0.75, 5.  , 5.  ]]]
cell_h2.a=[(1.875,0.,0.), (0.,10.0,0.), (0.,0.,10.0)]

driver_h2 = ChemistryDriverPySCFMomentumRHF(
    basis="sto-3g",
    geometry=cell_h2.atom,
    charge=0,
    cell=cell_h2.a,
    nks=[4, 1, 1],
    dimension=1, # 1D linear chain PBC system
    precision=1e-15,
    df="GDF",
    output=None,
    verbose=1,
    exp_to_discard = 0.1
)
print("Hartree-Fock energy: ", driver_h2.run_hf())
```

```
Hartree-Fock energy:  -1.0760400910887071
```

in this case, *get_system()* generates a special Fock space class: *FermionSpaceBrillouin*, which manages the k-point quantum number in addition to orbital and spin indices. Otherwise, construction of the quantum problem proceeds as normal.

20.7 Hamiltonians for Embedding methods

PySCF drivers provide an interface to various orbital localization schemes and a wide range of high-level chemistry methods. These can be utilized in a fragment solver for Density Matrix Embedding Theory (DMET) and Fragment Molecular Orbital (FMO) embedding methods.

Prior to an embedding algorithm one needs to generate a Hamiltonian with a localized and orthonormal basis or atomic orbitals, in which spatial fragments can be specified. To this end, PySCF drivers can generate the Löwdin representation of a system with the *get_lowdin_system* method:

```
driver = ChemistryDriverPySCFMolecularRHF(basis="sto-3g", geometry=g, charge=0)
hamiltonian_operator_lowdin, space, rdm1_hf_lowdin = driver.get_lowdin_system()
```

which performs a full Hartree-Fock calculation, and returns the 1-RDM in the localized basis. This representation is required for DMET calculations.

Additionally, for FMO calculations, one may use the *get_system_ao* method to generate a *PySCFChemistryRestrictedIntegralOperator* which, internally, has access to the atomic orbitals for generating fragment Hamiltonians:

```
driver = ChemistryDriverPySCFMolecularRHF(basis="sto-3g", geometry=g, charge=0)
hamiltonian_operator_pyscf, space, rdm1_hf_pyscf = driver.get_system_ao(run_hf=False)
```

where the `run_hf=False` option prevents a full Hartree-Fock calculation over the entire system.

The general API of both DMET and FMO has two type of classes that are necessary to perform a calculation: one that is responsible for the total system with the embedding method itself, and another that is responsible to compute the high-level solution of a particular fragment. For example, as it is discussed in the *Density Matrix Embedding Theory section*, there is an *ImpurityDMETROHF* class driving the embedding method, and there are fragment solver classes such as *ImpurityDMETROHFFragmentED*. Similarly for FMO we have the *inquanto.extensions.pyscf.fmo.FMO* and *inquanto.extensions.pyscf.fmo.FMOFragmentPySCFCCSD*, for example.

20.8 DMET with PySCF fragment solvers

Here we outline the use of PySCF calculators as fragment solvers for both Impurity DMET and full DMET with InQuanto. The reader is referred to the *main DMET section* for an introduction.

20.8.1 Impurity DMET

InQuanto's core packages provide a simple exact diagonalization solver and a Hartree-Fock fragment solver for impurity DMET. *inquanto-pyscf* provides additional fragment solvers:

- *inquanto.extensions.pyscf.ImpurityDMETROHFFragmentPySCFFCI*
- *inquanto.extensions.pyscf.ImpurityDMETROHFFragmentPySCFMP2*
- *inquanto.extensions.pyscf.ImpurityDMETROHFFragmentPySCFCCSD*
- *inquanto.extensions.pyscf.ImpurityDMETROHFFragmentPySCFROHF*

Provided we have the Hamiltonian and the density matrix in a *localized basis* we can run the single fragment Impurity DMET method. Below we consider a Hamiltonian computed with the Löwdin transformation, with 6 spatial orbitals. The *ImpurityDMETROHF* class is initialized as follows:

```
from inquanto.embeddings import ImpurityDMETROHF

dmet = ImpurityDMETROHF(
    hamiltonian_operator_lowdin, rdm1_hf_lowdin
)
```

The class name of *ImpurityDMETROHF* denotes that this method assumes the Hamiltonian operator and the 1-RDM are spin restricted. After this initialization, a fragment and a corresponding fragment solver are specified in the following way:

```
from numpy import array

from inquanto.extensions.pyscf import (
    ImpurityDMETROHFFragmentPySCFFCI,
    ImpurityDMETROHFFragmentPySCFMP2,
    ImpurityDMETROHFFragmentPySCFCCSD,
    ImpurityDMETROHFFragmentPySCFROHF,
    ImpurityDMETROHFFragmentPySCFActive
 )

mask = array([True, True, False, False, False, False])
fragment = ImpurityDMETROHFFragmentPySCFFCI(dmet, mask)
```

ImpurityDMETROHFFragmentPySCFFCI defines the high-level solver for the fragment, which in this case is an FCI method. The fragment solver is initialized with the `dmet` object and the fragment mask `mask`. The `mask` is a boolean array with the length of the number of localized spatial orbitals. The `True` values in the mask select the indices of the localized spatial orbitals that belongs to the fragment.

Finally the embedding method can be run as usual:

```
result = dmet.run(fragment)
print(result)
```

20.8.2 One-shot DMET and full DMET

In practice, we maybe want to deal with larger molecules, and with multiple fragments. To handle multiple fragments we need to use the one-shot DMET or full DMET methods, both covered by the class *inquanto.embeddings.dmet.DMETRHF*. Similarly to impurity DMET, *inquanto-pyscf* includes a range of PySCF-driven fragment solvers:

- *inquanto.extensions.pyscf.DMETRHFFragmentPySCFFCI*
- *inquanto.extensions.pyscf.DMETRHFFragmentPySCFRHF*
- *inquanto.extensions.pyscf.DMETRHFFragmentPySCFCCSD*
- *inquanto.extensions.pyscf.DMETRHFFragmentPySCFMP2*

Here we consider the example of phenol:

```
from inquanto.geometries import GeometryMolecular
from inquanto.embeddings.dmet import DMETRHF

from inquanto.extensions.pyscf import (
    get_fragment_orbital_masks,
    get_fragment_orbitals,
    ChemistryDriverPySCFMolecularRHF,
    DMETRHFFragmentPySCFCCSD,
```

(continues on next page)

(continued from previous page)

```
    DMETRHFFragmentPySCFMP2,
)

# Phenol (C6H5OH)

# Setup the system - Phenol geometry
geometry = [['C', [-0.921240800,  0.001254500,  0.000000000]],
        ['C', [-0.223482600,  1.216975000,  0.000000000]],
        ['C', [ 1.176941000,  1.209145000,  0.000000000]],
        ['C', [ 1.882124000,  0.000763689,  0.000000000]],
        ['C', [ 1.171469000, -1.208183000,  0.000000000]],
        ['C', [-0.225726600, -1.216305000,  0.000000000]],
        ['O', [-2.284492000, -0.060545780,  0.000000000]],
        ['H', [-0.771286100,  2.161194000,  0.000000000]],
        ['H', [ 1.715459000,  2.156595000,  0.000000000]],
        ['H', [ 2.970767000, -0.000448048,  0.000000000]],
        ['H', [ 1.709985000, -2.155694000,  0.000000000]],
        ['H', [-0.792751600, -2.145930000,  0.000000000]],
        ['H', [-2.630400000,  0.901564000,  0.000000000]]]

g = GeometryMolecular(geometry)

driver = ChemistryDriverPySCFMolecularRHF(basis="sto-3g", geometry=g.xyz, charge=0)
hamiltonian_operator, space, rdm1 = driver.get_lowdin_system()

maskOH, maskCCH, maskCHCH1, maskCHCH2 = get_fragment_orbital_masks(
driver, [6, 12], [0, 1, 7], [2, 8, 3, 9], [4, 10, 5, 11]
)

dmet = DMETRHF(hamiltonian_operator, rdm1)

frOH = DMETRHFFragmentPySCFCCSD(dmet, maskOH)
frCCH = DMETRHFFragmentPySCFCCSD(dmet, maskCCH)
frCHCH1 = DMETRHFFragmentPySCFMP2(dmet, maskCHCH1)
frCHCH2 = DMETRHFFragmentPySCFCCSD(dmet, maskCHCH2)

fragments = [frOH, frCCH, frCHCH1, frCHCH2]

result = dmet.run(fragments)
print(result)
```

The utility function *get_fragment_orbital_masks()* helps to create the orbital masks from atom indices. In this particular examples, the 4 fragments are specified by lists of atom indices defined in the `geometry`. Consequently, *get_fragment_orbital_masks()* returns 4 masks. Note however, that one could in principle define their own masks which selects spatial orbitals within atoms or across multiple intra atomic orbitals. Howeverm for *DMETRHF* it is important the the set of fragments completely cover the entire molecule.

20.9 DMET with a custom solver

Running a VQE calculation on a DMET fragment of a large system requires the definition of a bespoke fragment solver. There are many type of ansatze and strategies to perform VQE; instead of providing a corresponding range of many black-box VQE fragment solvers, *inquanto-pyscf* provides an easy way for a user to define a fragment solver class.

To create a DMET fragment, we subclass the *inquanto.extensions.pyscf.DMETRHFFragmentPySCFActive* class, which requires us to implement the *solve_active()* method. This method assumes that the Hamiltonian, Fragment Energy operator and other arguments are only for the active space specified when the fragment solver is constructed. The mandatory return is the the expectation value of the Hamiltonian and the Fragment Energy operator with the ground state (`energy` and `fragment_energy`) and the 1-RDM of the ground state. Below, we outline the structure of these calculations, but full examples can be found on the examples page.

```
class MyFragment(DMETRHFFragmentPySCFActive):

    def solve_active( self,
        hamiltonian_operator, fragment_energy_operator, fermion_space, fermion_state,
    ):

        # ... your VQE solution ...

        return energy, fragment_energy, vqe_rdm1
```

The active space can be specified with the CAS notation via the *FromActiveSpace* class:

```
fr = MyFragment(dmet, mask, frozen=FromActiveSpace(2, 2))
```

or one can also provide the explicit list of indices of the frozen orbitals of the fragment embedding system. Note that the fragment solver in the case of DMET is solving the embedding system that is the fragment orbitals and the bath orbitals together.

For Impurity DMET the implementation is even simpler, only the energy is required as an output:

```
class MyFragment(ImpurityDMETROHFFragmentPySCFActive):

    def solve_active(
        self,
        hamiltonian_operator: ChemistryRestrictedIntegralOperator,
        fermion_space: FermionSpace,
        fermion_state: FermionState,
    ):

        jw = QubitMappingJordanWigner()
        ansatz = FermionSpaceAnsatzUCCSD(fermion_space, fermion_state, jw)

        h_op = jw.operator_map(hamiltonian_operator)

        objective_expression = ExpectationValue(ansatz, h_op)

        vqe = AlgorithmVQE(
            objective_expression=objective_expression,
            minimizer=MinimizerScipy(method="COBYLA", disp=True),
            initial_parameters=ansatz.state_symbols.construct_random(0, 0.0, 0.01),
        )

        vqe.build(backend=AerStateBackend(), protocol_
↪expression=ProtocolStateVectorSparse())
```

(continues on next page)

(continued from previous page)

```
        vqe.run()

        energy = vqe.final_value

        return energy
```

20.10 Other PySCF Hamiltonians for DMET

The API of DMET based embedding methods does not restrict the origin of the Hamiltonian. As long as it is in a localized orthonormal basis and its type is *ChemistryRestrictedIntegralOperator* the Hamiltonian can be constructed for a periodic system, or it can be a model Hamiltonian from the Hubbard driver, or it can be generated with COSMO.

For example, if one would like to perform an embedding with a COSMO calculation, then the Hamiltonian and the 1-RDM should be computed with the :class`~inquanto.extensions.pyscf.ChemistryDriverPySCFMolecularRHFQMMMCOSMO` driver:

```
from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHFQMMMCOSMO

driver = ChemistryDriverPySCFMolecularRHFQMMMCOSMO(
    basis="sto-3g", geometry=geometry, charge=0, do_cosmo=True
)

hamiltonian, fermion_space, dm = driver.get_lowdin_system()

# ... dmet and fragment definitions ...

energy, chemical_potential, parameters = dmet.run(fragments)

energy_water_frozen = energy + driver.cosmo_correction
```

Note: On the last line above, the COSMO energy is calculated by correcting the ground state energy of the Hamiltonian (which in this case is computed approximately by DMET) with the COSMO correction. This type of calculation assumes non self-consistent COSMO correction.

The COSMO method will be discussed in more detail *below*.

20.11 FMO with a custom solver

In addition to DMET embedding methods, InQuanto supports FMO based embedding calculations with the important difference that the Hamiltonian needs access to the atomic orbital basis:

```
hamiltonian_operator_ao, _, _ = driver.get_system_ao(run_hf=False)
```

To run an FMO simulation, one needs to instantiate an :class`inquanto.extensions.pyscf.FMO` class and, similarly to DMET, define fragment solvers before finally calling the `run()` method. In the example below we create a custom FMO class and perform VQE on the fragment Hamiltonians.

```
from inquanto.extensions.pyscf.fmo import FMOFragmentPySCFActive, FMO
from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF
from pytket.extensions.qiskit import AerStateBackend

class MyFMOFragmentVQE(FMOFragmentPySCFActive):

    def solve_final_active(
        self,
        hamiltonian_operator,
        fermion_space,
        fermion_state,
    ) -> float:

        from inquanto.ansatzes import FermionSpaceAnsatzUCCSD
        qubit_operator = hamiltonian_operator.qubit_encode()

        ansatz = FermionSpaceAnsatzUCCSD(fermion_space, fermion_state)

        from inquanto.express import run_vqe

        vqe = run_vqe(ansatz, hamiltonian_operator, AerStateBackend())

        return vqe.final_value


driver = ChemistryDriverPySCFMolecularRHF(
    geometry=[
        ["H", [0, 0, 0]],
        ["H", [0, 0, 0.8]],
        ["H", [0, 0, 2.0]],
        ["H", [0, 0, 2.8]],
        ["H", [0, 0, 4.0]],
        ["H", [0, 0, 4.8]],
    ],
    basis="sto3g",
    verbose=0,
)
full_integral_operator, _, _ = driver.get_system_ao(run_hf=False)

fmo = FMO(full_integral_operator)

fragments = [
    MyFMOFragmentVQE(
        fmo, [True, True, False, False, False, False], 2, "H2-1"
    ),
    MyFMOFragmentVQE(
        fmo, [False, False, True, True, False, False], 2, "H2-2"
    ),
    MyFMOFragmentVQE(
        fmo, [False, False, False, False, True, True], 2, "H2-3"
    ),
]

fmo.run(fragments)
```

20.12 QM/MM

In QM/MM methods, the energy is partitioned into quantum mechanical (QM) and molecular mechanical (MM) contributions [27] corresponding to different parts of the system. Thus the QM/MM approach combines the accuracy of QM with the speed of MM. Typically, the QM subsystem is a small region where the interesting chemistry takes place, while the MM part corresponds to some larger environment, represented by simple point charges, surrounding the QM region. In general, there are two ways of expressing the total energy in QM/MM, "subtractive" and "additive" [28]. In InQuanto the additive approach is adopted, which means the total energy takes the form

$$E_{\text{tot}} = E_{\text{QM}} + E_{\text{QM/MM}} + E_{\text{MM}} \tag{20.1}$$

where E_{QM} represents the contributions internal to the QM subsystem (calculated using high-level methods which include a two-body interaction or approximation thereof), E_{MM} comes from interactions between particles in the MM environment (a low cost classical term like the Coulomb interaction between point charges), and $E_{\text{QM/MM}}$ represents the interactions between the QM and MM regions (also a classical interaction). The latter involves a modification of the one-body component of the electronic Hamiltonian. The modified one-body part of the Hamiltonian becomes

$$h_{i,j} \to h_{i,j}^{\text{QM/MM}} = \int d\mathbf{x}\phi_i^*(\mathbf{x})\left(\frac{-\nabla^2}{2} - \sum_A \frac{Z_A}{|\mathbf{R}_A - \mathbf{r}|} - \sum_M \frac{q_M}{|\mathbf{R}_M - \mathbf{r}|}\right)\phi_j(\mathbf{x}) \tag{20.2}$$

where $\mathbf{x} = \{\mathbf{r}, s\}$ with $\mathbf{r}$ (s) the electronic position (spin), and the modified classical Coulomb interaction for nuclei becomes

$$V_{\text{nuc}} \to V_{\text{nuc}}^{\text{QM/MM}} = \sum_{A,B} \frac{Z_A Z_B}{|\mathbf{R}_A - \mathbf{R}_B|} + \sum_{A,M} \frac{Z_A q_M}{|\mathbf{R}_A - \mathbf{R}_M|} \tag{20.3}$$

where M labels the MM point charges, with charge q_M and position $\mathbf{R}_M$.

In *inquanto-pyscf* the specialized driver class: *ChemistryDriverPySCFMolecularRHFQMMMCOSMO*, uses the QM/MM scheme of PySCF is used to modify the one-body integrals and nuclear Coulomb interaction. With this driver, QM/MM embedding is performed as follows; we consider an H_2 molecule surrounded by ten random point charges:

```
from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHFQMMMCOSMO
# define the MM region
mm_charges = 0.1  # this will set all MM particles to have charge = +0.1 in units of␣
↪electron charge.
# lists and numpy arrays are also accepted to specify charges individually.
mm_geometry = [  # pregenerated random geometry using nunmpy.random.seed(1)
    [-0.49786797, 1.32194696, -2.99931375],
    [-1.18600456, -2.11946466, -2.44596843],
    [-1.88243873, -0.92663564, -0.61939515],
    [0.2329004, -0.48483291, 1.111317],
    [-1.7732865, 2.26870462, -2.83567444],
    [1.02280506, -0.49617119, 0.35213897],
    [-2.15767837, -1.81139107, 1.80446741],
    [2.80956945, -1.11945493, 1.15393569],
    [2.25833491, 2.36763998, -2.48973473],
    [-2.7656713, -1.98101748, 2.26885502],
]
# e_mm_coulomb should be 0.068638356203663

# Execute the chemistry drivers.
driver_qmmm = ChemistryDriverPySCFMolecularRHFQMMMCOSMO(
    zmatrix="""
        H
```

(continues on next page)

(continued from previous page)

```
        H  1 0.7122
    """,
    basis="sto3g",
    mm_charges=mm_charges,
    mm_geometry=mm_geometry,
    do_mm_coulomb=True,
    do_qmmm=True,
)
```

The Hamiltonian returned by *get_system()* will now contain the modified one-body terms due to the QM/MM interaction. To calculate E_{tot} as above including the Coulomb interaction between MM point charges, the optional boolean keyword `do_mm_coulomb` must be set to `True`, which stores the MM Coulomb interaction energy E_{MM} as a driver property: `e_mm_coulomb`. However, this does not add E_{MM} to the total energy, which must be done manually to recover $E_{\text{QM}} + E_{\text{QM/MM}} + E_{\text{MM}}$.

QM/MM can also be used with ROHF and UHF methods with the corresponding drivers *ChemistryDriverPySCFMolecularROHFQMMMCOSMO* and *ChemistryDriverPySCFMolecularUHFQMMMCOSMO*. In addition, QM/MM can be combined with other embedding schemes.

20.13 COSMO

The COSMO (COnductor-like Screening MOdel) scheme [29] is a method for modelling the interaction of a molecule with a solvent. The solvent is approximated as a continuous medium (implicit solvation), and the interaction between the molecule and solvent corresponds to a modification of the one-body interaction. In InQuanto, the PySCF implementation of ddCOSMO (domain-decomposition COSMO) is utilised. In this scheme, the mean-field SCF problem is solved with a modified Hamiltonian

$$E_{\text{SCF}} = \langle\Psi|\hat{H} + \hat{V}_{\text{ddCOSMO}}|\Psi\rangle \tag{20.4}$$

Where $\hat{V}_{\text{ddCOSMO}}$ is a functional of the electronic density. This allows the orbitals to respond to the implicit solvation during the SCF procedure. However, E_{SCF} would not correspond to the total energy according to the ddCOSMO model. For this, a contribution E_{ddCOSMO} calculated from a classical dielectric problem is needed, in which the molecule is contained in a cavity defined by overlapping spheres centred on its atoms, and outside the cavity there is a continuous medium with dielectric constant fixed to that of the solvent

$$E_{\text{ddCOSMO}} = \frac{1}{2} f(\epsilon_s) \int_{\Omega} \rho(\mathbf{r}) W(\mathbf{r})\, d\mathbf{r} \tag{20.5}$$

where $f(\epsilon_s)$ is an empirical function of the solvent dielectric constant ϵ_s, $\rho(\mathbf{r})$ is the electronic density, $W(\mathbf{r})$ is a surface potential obtained from spherical harmonics, and the integral is over the surface defined by the cavity. For more details see [30].

Since E_{ddCOSMO} is treated as a classical constant term in the expectation value of the energy, the quantity $\langle\Psi|\hat{H}|\Psi\rangle + E_{\text{ddCOSMO}}$ would not contain the response of orbitals to the solvent environment. Hence, the following expression for the total energy is adopted in PySCF and used in the *inquanto-pyscf* extension:

$$E_{\text{tot}} = \langle\Psi|\hat{H} + \hat{V}_{\text{ddCOSMO}}|\Psi\rangle - \text{Tr}(D\hat{V}_{\text{ddCOSMO}}) + E_{\text{ddCOSMO}} \tag{20.6}$$

where D is the one-body reduced density matrix. Hence, the SCF problem is solved in the presence of the solvent potential which affects the orbitals, but this contribution is subtracted out and the classical COSMO energy E_{ddCOSMO} is added back in, so that the energy corresponds to the ddCOSMO model while the orbitals are also consistent with the background solvent potential.

The following example shows the instantiation of the *ChemistryDriverPySCFMolecularRHFQMMMCOSMO* PySCF driver class for COSMO solvation:

```
from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHFQMMMCOSMO

# instantiation of driver with COSMO solvent
driver_cosmo = ChemistryDriverPySCFMolecularRHFQMMMCOSMO(
    zmatrix="""
        H
        H  1 0.7122
    """,
    basis="sto3g",
    do_cosmo = True
)

h, fsp, fst = driver_cosmo.get_system()
mf_solvent_e_tot = driver_cosmo.mf_energy
```

The Hamiltonian `h` returned by *`get_system()`* contains the modified one-body term, and construction of the quantum problem can procede as usual. The energy `mf_solvent_e_tot` corresponds to the mean field energy with the COSMO corrections as specified in (20.6).

It is also possible to combine COSMO with other embedding schemes in InQuanto by setting `do_cosmo=True` in the driver as above.

CHAPTER

TWENTYONE

INQUANTO-NGLVIEW

The *InQuanto-NGLView* extension provides basic utilities for visualizing chemical systems via the NGLView package. These utilities return NGL widgets which can be interactively viewed in a jupyter notebook. The functionality includes visualization of molecular structures, molecular fragmentation schemes, and molecular orbital isosurfaces.

21.1 Visualizing Structures

The *VisualizerNGL* class is the central object of the InQuanto-NGLView extension. *VisualizerNGL* takes an InQuanto *Geometry* object as input, and produces an interactive `NGLWidget` with the *visualize_molecule()* method, visualizable in a jupyter notebook. See below for an example with molecular geometry:

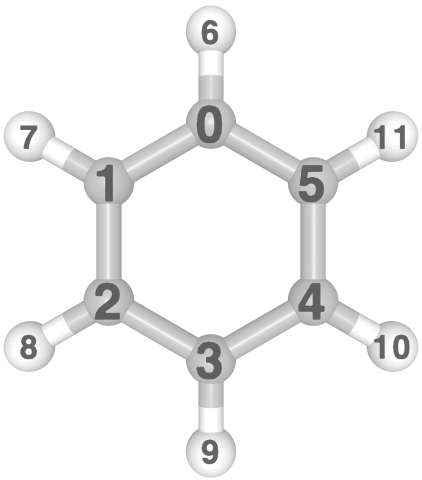
```
from inquanto.geometries import GeometryMolecular
from inquanto.extensions.nglview import VisualizerNGL

xyz = [
    ['C', [ 0.0000000,  1.4113170, 0.0000000]],
    ['C', [ 1.2222370,  0.7056590, 0.0000000]],
    ['C', [ 1.2222370, -0.7056590, 0.0000000]],
    ['C', [ 0.0000000, -1.4113170, 0.0000000]],
    ['C', [-1.2222370, -0.7056590, 0.0000000]],
    ['C', [-1.2222370,  0.7056590, 0.0000000]],
    ['H', [ 0.0000000,  2.5070120, 0.0000000]],
    ['H', [ 2.1711360,  1.2535060, 0.0000000]],
    ['H', [ 2.1711360, -1.2535060, 0.0000000]],
    ['H', [ 0.0000000, -2.5070120, 0.0000000]],
    ['H', [-2.1711360, -1.2535060, 0.0000000]],
    ['H', [-2.1711360,  1.2535060, 0.0000000]]
]
c6h6_geom = GeometryMolecular(xyz)
visualizer = VisualizerNGL(c6h6_geom)
visualizer.visualize_molecule(atom_labels="index")
```

Note: Code snippets on this page generate interactive cells in a jupyter notebook. Static images are shown here for demonstration.

Similarly, periodic systems may be visualized by providing *VisualizerNGL* with a *GeometryPeriodic* object:

```
from inquanto.extensions.nglview import VisualizerNGL
from inquanto.geometries import GeometryPeriodic
import numpy as np
```

(continues on next page)



(continued from previous page)

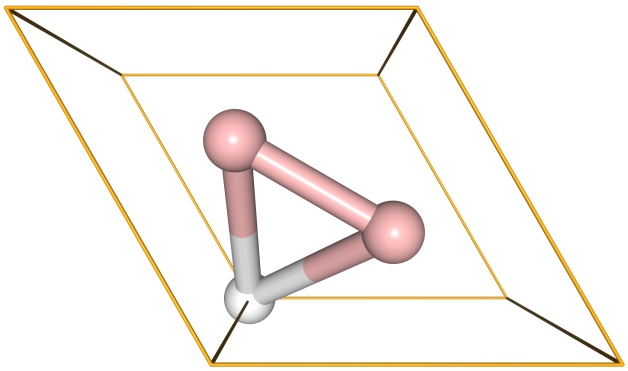
```
 # AlB2 unit cell
 a=3.01 # lattice vectors in Angstroms
 c=3.27
 a, b, c = [
     np.array([a, 0, 0]),
     np.array([-a/2, a*np.sqrt(3)/2, 0]),
     np.array([0, 0, c])
 ]
 atoms = [
     ["Al", [0, 0, 0]],
     ["B", a/3 + b*2/3 + c/2],
     ["B", a*2/3 + b/3 + c/2]
 ]

alb2_geom = GeometryPeriodic(geometry=atoms, unit_cell=[a, b, c])
visualizer = VisualizerNGL(alb2_geom)
visualizer.visualize_unit_cell()
```

21.2 Visualizing Fragments

There are several fragmentation methods available in InQuanto; to visualize a fragmentation scheme defined in a *GeometryMolecular* object, use the *visualize_fragmentation()* method:

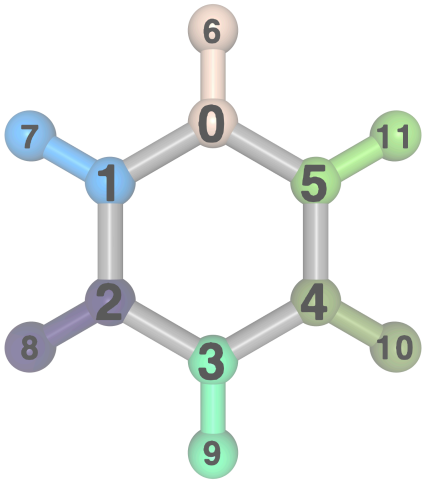
```
c6h6_geom.set_groups(
    "fragments",
    {
        "ch1": [0, 6],
        "ch2": [5, 11],
        "ch3": [4, 10],
        "ch4": [3, 9],
        "ch5": [2, 8],
        "ch6": [1, 7]
    }
```

(continues on next page)



(continued from previous page)

```
)
visualizer.visualize_fragmentation("fragments", atom_labels="index")
```

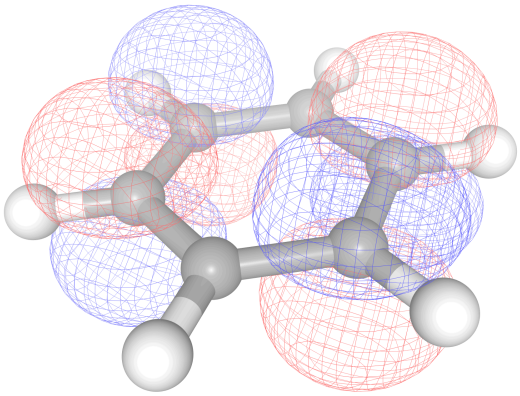


Note: Visualizing fragments is not yet supported for periodic geometries.

21.3 Visualizing Orbitals

To provide a visual aid in active space selection, molecular orbitals may be visualized with the *visualize_orbitals()* method. Orbital information must be provided in `.cube` format, which may be generated for molecular systems with the *InQuanto-PySCF* extension. In the example below, we generate the Hartree-Fock molecular orbitals for benzene in a minimal basis, and select the LUMO for visualisation.

```
from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF

driver = ChemistryDriverPySCFMolecularRHF(geometry=c6h6_geom.xyz, basis="sto3g")

cube_orbitals=driver.get_cube_orbitals()
ngl_mos = [visualizer.visualize_orbitals(orb) for orb in cube_orbitals]
ngl_mos[21]
```



CHAPTER

TWENTYTWO

INQUANTO API REFERENCE

22.1 inquanto.algorithms

class AlgorithmAdaptVQE(*pool*, *state*, *hamiltonian*, *minimizer*, *auxiliary_operator=None*, *n_iterations=100*, *tolerance=1e-3*, *disp=False*)

Bases: `object`

ADAPT - An algorithm for approximating the ground-state energy of a chemical or material system.

The algorithm finds a compact pauli exponential ansatz that is capable of approximating the ground-state energy. The implementation is based on work in arXiv:1812.11173 .

Parameters

- **pool** (*QubitOperatorList*) – Holds the pool of Pauli terms which go the TrotterAnsatz object.
- **state** (*QubitState*) – Initial (para-)fermionic reference state for the chemical system in question (usually the HF determinant).
- **hamiltonian** (*QubitOperator*) – The Hermitian operator to measure for the lowest eigenvalue.
- **minimizer** (`GeneralMinimizer`) – Variational Minimizer to use for the ADAPT experiment.
- **auxiliary_operator** (`Optional`[`List`[*QubitOperator*]], default: `None`) – Additional Hamiltonian operators to evaluate in parallel.
- **n_iterations** (`int`, default: `100`) – Number of iterations before termination.
- **tolerance** (`float`, default: `1e-3`) – Expectation value of commutation between pool and Hamiltonian at which loop is stopped.
- **disp** (`bool`, default: `False`) – If the algorithm should display variational data every iteration.

build(*backend*, *protocol_expectation*, *protocol_pool_metric*, *protocol_gradient=None*, *n_shots=8192*)

Build the algorithm using the provided protocols.

Parameters

- **backend** (`Backend`) – The backend used for circuit simulation.
- **protocol_expectation** (`Protocol`) – The protocol used for expectation value calculation.

- **protocol_pool_metric** (Protocol) – The protocol used to determine the excitation selection metric.
- **protocol_gradient** (Optional[Protocol], default: None) – The protocol used for gradient calculation.
- **n_shots** (int, default: 8192) – The number of shots used per expectation value calculation.

Returns
AlgorithmAdaptVQE – self

generate_report()

Return a dict giving experimental results.

run(*seed=None*, *compiler_passes=None*)

Perform the ADAPT experiment.

After execution, a TrotterAnsatz instance, optimized energy and final parameters will be available.

Parameters

- **seed** (Optional[int], default: None) – Seed used for random number generation in the backend.
- **compiler_passes** (Optional[BasePass], default: None) – Circuit optimisation pass applied at each iteration of the algorithm.

Return type
None

class AlgorithmFermionicAdaptVQE(*pool*, *state*, *hamiltonian*, *minimizer*, *auxiliary_operator=None*, *n_iterations=100*, *tolerance=1e-3*, *disp=False*, *qubit_mapping=QubitMappingJordanWigner()*)

Bases: *AlgorithmAdaptVQE*

Fermionic ADAPT – an algorithm for approximating the ground-state energy of fermionic system.

The algorithm finds a compact pauli exponential ansatz that is capable of approximating the ground-state energy. The implementation is based on work in arXiv:1812.11173 .

Parameters

- **pool** (*FermionOperatorList*) – Holds the pool of Pauli terms which go the TrotterAnsatz object.
- **state** (*FermionState*) – Initial (para-)fermionic reference state for the chemical system in question (usually the HF determinant).
- **hamiltonian** (*FermionOperator*) – The Hermitian operator to measure for the lowest eigenvalue.
- **minimizer** (GeneralMinimizer) – Variational Minimizer to use for the ADAPT experiment.
- **auxiliary_operator** (Optional[List[*FermionOperator*]], default: None) – Additional Hamiltonian operators to evaluate in parallel.
- **n_iterations** (int, default: 100) – Number of iterations before termination.
- **tolerance** (float, default: 1e-3) – Expectation value of commutation between pool and Hamiltonian at which loop is stopped.

- **disp** (bool, default: False) – If the algorithm should display variational data every iteration.
- **qubit_mapping** (*QubitMapping*, default: QubitMappingJordanWigner()) – Fermion-to-qubit mapping scheme to transform fermionic operators and reference state.

build(*backend*, *protocol_expectation*, *protocol_pool_metric*, *protocol_gradient=None*, *n_shots=8192*)

Build the algorithm using the provided protocols.

Parameters

- **backend** (Backend) – The backend used for circuit simulation.
- **protocol_expectation** (Protocol) – The protocol used for expectation value calculation.
- **protocol_pool_metric** (Protocol) – The protocol used to determine the excitation selection metric.
- **protocol_gradient** (Optional[Protocol], default: None) – The protocol used for gradient calculation.
- **n_shots** (int, default: 8192) – The number of shots used per expectation value calculation.

Returns

AlgorithmAdaptVQE – self

generate_report()

Return a dict giving experimental results.

get_ansatz(*state*, *fermion_ansatz_type=FermionSpaceStateExpChemicallyAware*)

Returns an ansatz built from symbol-containing operators of the fermionic pool.

Parameters

- **state** (*FermionState*) – Fermion state object to build an ansatz with.
- **fermion_ansatz_type** (Union[Type[*FermionSpaceStateExp*], Type[*FermionSpaceStateExpChemicallyAware*]], default: FermionSpaceStateExpChemicallyAware) – Type of fermionic ansatz to instansiate.

Returns

Union[*FermionSpaceStateExp*, *FermionSpaceStateExpChemicallyAware*] – A new ansatz object.

get_exponents_with_symbols()

Returns a symbol-containing sublist of the fermionic pool operator list.

run(*seed=None*, *compiler_passes=None*)

Perform the ADAPT experiment.

After execution, a TrotterAnsatz instance, optimized energy and final parameters will be available.

Parameters

- **seed** (Optional[int], default: None) – Seed used for random number generation in the backend.
- **compiler_passes** (Optional[BasePass], default: None) – Circuit optimisation pass applied at each iteration of the algorithm.

Return type

None

class AlgorithmIQEB(*pool*, *state*, *hamiltonian*, *minimizer*, *auxiliary_operator=None*, *n_iterations=100*, *n_grads=10*, *energy_tolerance=1.0e-10*, *disp=False*, *verbose=False*)

Bases: *AlgorithmAdaptVQE*

An ADAPT-like algorithm in which operators correspond to parafermionic qubit excitations.

This algorithm is described in arXiv:2011.10540. The qubit excitations obey different commutation relations and so do not require tensor product over Z rotations which occur in Jordan-Wigner transformed UCC operators. Convergence is reached by checking energy reduction between iterations, while gradients are used to narrow down pool of operators. The ParaFermionSpace generators are used with this class.

Parameters

- **hamiltonian** (*QubitOperator*) – The hermitian operator to measure for the lowest eigenvalue.
- **pool** (*QubitOperatorList*) – Holds the pool of Pauli terms which go the TrotterAnsatz object.
- **state** (*QubitState*) – Initial (para-)fermionic reference state for the chemical system in question (usually the HF determinant).
- **minimizer** (GeneralMinimizer) – Variational minimizer to use for the ADAPT experiment.
- **auxiliary_operator** (Optional[List[*QubitOperator*]], default: None) – Additional Hamiltonian operators to evaluate in parallel.
- **n_iterations** (int, default: 100) – Number of iterations before termination.
- **n_grads** (int, default: 10) – Number of gradients before termination.
- **energy_tolerance** (float, default: 1.0e-10) – Condition of termination of IQEB algorithm. Corresponds to threshold energy difference. between IQEB iterations. Default is 1e-10 (in Hartrees).
- **disp** (bool, default: False) – If the algorithm should display variational data every iteration.
- **verbose** (bool, default: False) – Output information, showing information on IQEB run.

build(*backend*, *protocol_expectation*, *protocol_pool_metric*, *protocol_gradient=None*, *n_shots=8192*)

Build the algorithm using the provided protocols.

Parameters

- **backend** (Backend) – The backend used for circuit simulation.
- **protocol_expectation** (Protocol) – The protocol used for expectation value calculation.
- **protocol_pool_metric** (Protocol) – The protocol used to determine the excitation selection metric.
- **protocol_gradient** (Optional[Protocol], default: None) – The protocol used for gradient calculation.
- **n_shots** (int, default: 8192) – The number of shots used per expectation value calculation.

Returns

AlgorithmAdaptVQE – self

generate_report()

Return a dict giving experimental results.

run(*seed=None*, *compiler_passes=None*)

Run the ADAPT-like IQEB algorithm.

This uses gradients to narrow down the pool of operator exponents. From this smaller pool, a number of VQEs are performed, and the only with the biggest energy reduction is used to choose the operator that gets appended to the final ansatz. The algorithm stops when the energy reduction is smaller than a threshold.

After execution, a TrotterAnsatz instance, optimized energy and final parameters will be available.

Parameters

- **seed** (Optional[int], default: None) – Seed used for random number generation in the backend.
- **compiler_passes** (Optional[BasePass], default: None) – Circuit optimisation pass applied at each iteration of the algorithm.

Return type
None

class AlgorithmQSE(*computable_qse_matrices*, *parameters*)

Bases: object

Quantum Subspace Expansion algorithm.

The implmentation here is based on work in arXiv:1603.05681.

Parameters

- **computable_qse_matrices** (*ComputableQSEMatrices*) – Computable to return H and S matrices.
- **parameters** (*SymbolDict*) – Circuit parameters from which the subspace is generated.

build(*backend*, *objective_protocol*, *n_shots=8192*)

Build the algorithm using the provided protocols.

Parameters

- **backend** (Backend) – The backend used for circuit simulation.
- **objective_protocol** ((*ProtocolDirect*, *ProtocolIndirect*, ProtocolStateVectorSparseLegacy, *ProtocolStateVectorSparse*, *ProtocolStateVectorBackendSupport*)) – The protocol used for objective function evaluation.
- **n_shots** (int, default: 8192) – The number of shots used per expectation value calculation.

Returns
AlgorithmQSE – self

property final_states

Final states in QSE computable.

property final_values

Final expectation values of QSE computable.

generate_report()

Return a dict giving experimental results.

Return type
Dict

run(*seed=0*, *compiler_passes=None*)

Perform the QSE experiment. Results can be queried with .generate_report().

Parameters

- **seed** (Optional[int], default: 0) – Seed used for random number generation in the backend.
- **compiler_passes** (Optional[BasePass], default: None) – Circuit optimisation pass applied at each iteration of the algorithm.

Return type
None

class AlgorithmVQS(*integrator*, *expressions*, *initial_parameters*)

Bases: object

Base class for all the Variational Quantum Simulation Methods.

This implementation is based on work in Quantum 3, 191 (2019).

Parameters

- **integrator** (GeneralIntegrator) – An integrator to solve linear equations.
- **expressions** (Computable) – A computable expression whose evaluate() returns a matrix A and vector b as (A,b) for the linear problem A*x = b.
- **initial_parameters** (*SymbolDict*) – Initial parameters for the time evolution.

build(*backend*, *protocol*)

Executes the build method of a computable expression.

Parameters

- **backend** (Backend) – Backend for the computable protocols.
- **protocol** (Protocol) – Protocol to build computable with.

Returns
AlgorithmVQS – self.

property final_parameters: *SymbolDict*

Parameters used to evaluate final expectation value.

Return type
SymbolDict

final_propagation_evaluation(*built_expression*, ***kwargs*)

Evaluates input computable expression for the last step of the propagation.

Parameters

- **built_expression** (Computable) – Computable expression to be evaluated.
- ****kwargs** – Keyword arguments to be passed to computable run() method.

Returns
ndarray – Evaluated computable expression for the last step of the propagation.

post_propagation_evaluation(*built_expression*, **args*, ***kwargs*)

Evaluates input computable expression along propagation trajectory.

This is done after propagation has been done.

Parameters

- **built_expression** (Computable) – Computable expression to be evaluated along propagation trajectory.
- ***args** – Arguments to be passed to computable run() method.
- ****kwargs** – Keyword arguments to be passed to computable run() method.

Returns

List[ndarray] – A list of evaluated computable expression for each time step of the propagation.

run(***kwargs*)

Performs the ODE experiment.

Parameters

****kwargs** – Passed to the underlying computable run.

Returns

ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Array of parameters, being a solution to ODE experiment.

Raises

RuntimeError – If member variable _backend is not initialised.

class AlgorithmMcLachlanRealTime(*integrator*, *hamiltonian*, *ansatz*, *initial_parameters*)

Bases: *AlgorithmVQS*

Algorithm for real time evolution with McLachlan's variational principle.

Based on work in `Quantum 3, 191 (2019) <https://quantum-journal.org/papers/q-2019-10-07-191/>

Parameters

- **integrator** (GeneralIntegrator) – An integrator to solve linear equations.
- **hamiltonian** (*QubitOperator*) – Hamiltonian under which to time evolve.
- **ansatz** (*GeneralAnsatz*) – Wavefunction ansatz to time evolve.
- **initial_parameters** (*SymbolDict*) – Initial parameters for time evolution.

build(*backend*, *protocol*)

Executes the build method of a computable expression.

Parameters

- **backend** (Backend) – Backend for the computable protocols.
- **protocol** (Protocol) – Protocol to build computable with.

Returns

AlgorithmVQS – self.

property final_parameters: *SymbolDict*

Parameters used to evaluate final expectation value.

Return type

SymbolDict

final_propagation_evaluation (*built_expression*, ***kwargs*)

Evaluates input computable expression for the last step of the propagation.

Parameters

- **built_expression** (Computable) – Computable expression to be evaluated.
- ****kwargs** – Keyword arguments to be passed to computable run() method.

Returns

ndarray – Evaluated computable expression for the last step of the propagation.

post_propagation_evaluation (*built_expression*, **args*, ***kwargs*)

Evaluates input computable expression along propagation trajectory.

This is done after propagation has been done.

Parameters

- **built_expression** (Computable) – Computable expression to be evaluated along propagation trajectory.
- ***args** – Arguments to be passed to computable run() method.
- ****kwargs** – Keyword arguments to be passed to computable run() method.

Returns

List[ndarray] – A list of evaluated computable expression for each time step of the propagation.

run (***kwargs*)

Performs the ODE experiment.

Parameters

****kwargs** – Passed to the underlying computable run.

Returns

ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Array of parameters, being a solution to ODE experiment.

Raises

RuntimeError – If member variable _backend is not initialised.

class AlgorithmMcLachlanImagTime (*integrator*, *hamiltonian*, *ansatz*, *initial_parameters*)

Bases: *AlgorithmVQS*

Algorithm for imaginary time evolution with McLachlan's variational principle.

Based on work in `Quantum 3, 191 (2019) <https://quantum-journal.org/papers/q-2019-10-07-191/>

Parameters

- **integrator** (GeneralIntegrator) – An integrator to solve linear equations.
- **hamiltonian** (*QubitOperator*) – Hamiltonian under which to time evolve.
- **ansatz** (*GeneralAnsatz*) – Wavefunction ansatz to time evolve.
- **initial_parameters** (*SymbolDict*) – Initial parameters for time evolution.

build (*backend*, *protocol*)

Executes the build method of a computable expression.

Parameters

- **backend** (Backend) – Backend for the computable protocols.

- **protocol** (Protocol) – Protocol to build computable with.

Returns
AlgorithmVQS – self.

property final_parameters: *SymbolDict*

Parameters used to evaluate final expectation value.

Return type
SymbolDict

final_propagation_evaluation (*built_expression*, ***kwargs*)

Evaluates input computable expression for the last step of the propagation.

Parameters

- **built_expression** (Computable) – Computable expression to be evaluated.
- ****kwargs** – Keyword arguments to be passed to computable run() method.

Returns
ndarray – Evaluated computable expression for the last step of the propagation.

post_propagation_evaluation (*built_expression*, **args*, ***kwargs*)

Evaluates input computable expression along propagation trajectory.

This is done after propagation has been done.

Parameters

- **built_expression** (Computable) – Computable expression to be evaluated along propagation trajectory.
- ***args** – Arguments to be passed to computable run() method.
- ****kwargs** – Keyword arguments to be passed to computable run() method.

Returns
List[ndarray] – A list of evaluated computable expression for each time step of the propagation.

run (***kwargs*)

Performs the ODE experiment.

Parameters
****kwargs** – Passed to the underlying computable run.

Returns
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Array of parameters, being a solution to ODE experiment.

Raises
RuntimeError – If member variable _backend is not initialised.

class AlgorithmVQD (*objective_expression*, *overlap_expression*, *weight_expression*, *minimizer*, *initial_parameters*, *vqe_value*, *vqe_parameters*, *n_vectors*)

Bases: object

Algorithm to sequentially obtain the excited states of a Hamiltonian.

The algorithm orthogonally constrains the previously found eigenstates (see Quantum 3, 156 (2019)), where each eigenstate is found with a separate VQE experiment. A number-conserving ansatz should be used for VQD. Sometimes, spin-crossing excitations are required.

Parameters

- **objective_expression** (*ExpectationValue*) – A preconfigured ExpectationValue expression to evaluate the excited state energy of some trial wave function and Hamiltonian.
- **overlap_expression** (*OverlapSquared*) – Computable expression to calculate the square of the overlap between two trial states.
- **weight_expression** (*ExpectationValue*) – Computable expression used for the deflation scheme, the penalty applied due to the difference between the initial expectation value of this expression and the n-1 excited state energy.
- **minimizer** (GeneralMinimizer) – Variational classical minimizer to perform the parameter search.
- **initial_parameters** (*SymbolDict*) – A set of initial Ansatz parameters for the objective and weight expressions.
- **vqe_value** (float) – Energy of the reference state.
- **vqe_parameters** (*SymbolDict*) – Parameters of the reference (ground) state.
- **n_vectors** (int) – Number of subsequent excited states to generate.

build(*backend*, *objective_protocol*, *weight_protocol*, *overlap_protocol*, *n_shots=8192*)

Build the VQD experiment.

Provide objects required to estimate the expectation value of the ansatz, weighting of Hamiltonian eigenstates, and calculation of the overlap between states for the VQE experiment.

Parameters

- **backend** (Backend) – The backend used for circuit simulation.
- **objective_protocol** (Protocol) – The protocol used for energy expectation value estimation.
- **weight_protocol** (Protocol) – The protocol used to estimate the weighting of eigenstates of the Hamiltonian.
- **overlap_protocol** (Protocol) – The protocol used for overlap estimation between successive states.
- **n_shots** (int, default: 8192) – The number of shots used for expectation value estimation.

Returns

AlgorithmVQD – self

property final_parameters

Parameters used to evaluate final expectation value.

property final_values

Expectation value result of VQE experiment.

generate_report()

Return a dict giving experimental results.

Return type

Dict

run(*seed=0*, *compiler_passes=None*)

Perform the VQD experiment.

Results can be queried with .generate_report()

Parameters

- **seed** (Optional[int], default: 0) – Seed used for random number generation in the backend
- **compiler_passes** (Optional[BasePass], default: None) – Circuit optimisation pass applied at each iteration of the algorithm

class AlgorithmVQE(*objective_expression*, *minimizer*, *initial_parameters*, *gradient_expression=GradientType.NONE*, *auxiliary_expression=None*)

Bases: object

Variational quantum eigensolver algorithm.

An algorithm for finding ground state energies of molecular Hamiltonians.

Parameters

- **objective_expression** (*ExpectationValue*) – A preconfigured ExpectationValue expression to evaluate the ground-state energy of some trial wave function and hamiltonian.
- **minimizer** (GeneralMinimizer) – Variational classical minimizer to perform the parameter search.
- **initial_parameters** (*SymbolDict*) – A set of initial Ansatz parameters.
- **gradient_expression** (Union[*ExpectationValueDerivativeReal*, GradientType], default: GradientType.NONE) – An expression to evaluate gradient of objective function.
- **auxiliary_expression** (Optional[Computable], default: None) – Additional expressions to evaluate alongside the energy.

build(*backend*, *protocol_expression*, *protocol_gradient=None*, *n_shots=8192*)

Build the VQE experiment.

Provide objects required to estimate the expectation value of the ansatz, and gradient estimation used for minimization.

Parameters

- **backend** (Backend) – The backend used for circuit simulation.
- **protocol_expression** (Union[*ProtocolDirect*, *ProtocolIndirect*]) – The protocol used for energy expectation value estimation.
- **protocol_gradient** (Union[*ProtocolDirect*, *ProtocolIndirect*, None], default: None) – The protocol used for energy gradient estimation.
- **n_shots** (int, default: 8192) – The number of shots used for expectation value estimation.

Returns

AlgorithmVQE – self

property final_evaluated_auxiliary_expression

Expectation value result of VQE experiment on auxiliary expression.

property final_evaluated_objective_expression

Expectation value of objective function using the final set of parameters.

property final_parameters

Parameters used to evaluate final expectation value.

property final_value

Expectation value result of VQE experiment.

generate_report()

Return a dict giving experimental results.

Return type
Dict

run(*seed=None*, *compiler_passes=None*)

Perform the VQE experiment.

Results may be queried with `.generate_report()`

Parameters

- **seed** (Optional[int], default: None) – Seed used for random number generation in the backend.
- **compiler_passes** (Optional[BasePass], default: None) – Circuit optimisation pass applied at each iteration of the algorithm.

Return type
AlgorithmVQE

22.2 inquanto.ansatzes

class GeneralAnsatz(*reference*, *symbols*, *compiler_passes*, **args*, ***kwargs*)

Bases: Symbolic, Representable

Base class for a quantum state that can be represented with a single circuit.

Parameters

- **reference** (Union[int, List[Qubit], List[int], *QubitSpace*, *QubitState*, Circuit]) – An reference state circuit or any valid initializer for reference_circuit_builder(…).
- **symbols** (*SymbolSet*) – An ordered set of symbols that are in the circuit.
- **compiler_passes** (BasePass) – An option compiler passes to circuit compilation.
- **args** (Any) –
- **kwargs** (Any) –

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

abstract free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

abstract free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

abstract get_circuit(*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns
Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.
- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns
Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, ***, *space=None*)

Constructs a single symbolic matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns
Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns
str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type
int

property n_symbols: int

Returns the number of free symbols in the object.

Return type
int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns
QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type
Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs(*symbol_map*)

Returns a new objects with symbols substituted.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type

TypeVar(SYMBOLICTYPE, bound= Symbolic)

abstract symbol_substitution(*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns

TypeVar(SYMBOLICTYPE, bound= Symbolic) – self, with symbols substituted.

to_QubitState()

Create a symbolic QubitState representation of the ansatz.

Returns

QubitState – Ansatz as a QubitState.

class CircuitAnsatz(*circuit*, *reference=None*, *symbols=None*, *compiler_passes=None*)

Bases: *GeneralAnsatz*

An ansatz that stores a single symbolic circuit.

Parameters

- **circuit** (Circuit) – A symbolic circuit.
- **reference** (Union[int, List[Qubit], List[int], *QubitSpace*, *QubitState*, Circuit, None], default: None) – An optional reference state circuit or any valid initializer for reference_circuit_builder().
- **symbols** (Optional[*SymbolSet*], default: None) – An optional set of symbols that are in the circuit.
- **compiler_passes** (Optional[BasePass], default: None) – An option compiler passes to circuit compilation.

clone()

Performs shallow copy of the object.

Return type

TypeVar(SYMBOLICTYPE, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

get_circuit(*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns
Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.
- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns
Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, ***, *space=None*)

Constructs a single symbolic matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns
Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns
str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type
int

property n_symbols: int

Returns the number of free symbols in the object.

Return type
int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns
QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type
Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs(*symbol_map*)

Returns a new objects with symbols substituted.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type

TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution(*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns

CircuitAnsatz – self, with symbols substituted.

to_QubitState()

Create a symbolic QubitState representation of the ansatz.

Returns

QubitState – Ansatz as a QubitState.

class ComposedAnsatz(**ansatzes*, *reference=None*, *symbols=None*)

Bases: *CircuitAnsatz*

Composes circuit ansatzes into one circuit ansatz.

Notes

The composed circuit is the reversely joined circuits of the ansatzes list. That is the composed ansatz of A and B represents the state AB|0>.

Examples

```
>>> a = CircuitAnsatz(Circuit(1).Y(0))
>>> b = CircuitAnsatz(Circuit(1).Z(0), [1])
>>> ComposedAnsatz(a, b).get_circuit()
[X q[0]; Z q[0]; Y q[0]; ]
>>> c = CircuitAnsatz(Circuit(1).Y(0))
>>> d = CircuitAnsatz(Circuit(1).Z(0))
>>> ComposedAnsatz(c, d, reference = [1]).get_circuit()
[X q[0]; Z q[0]; Y q[0]; ]
```

Parameters

- ***ansatzes** (*GeneralAnsatz*) – sequence of circuit ansatzes that will be appended together.
- **reference** (Union[int, List[Qubit], List[int], *QubitSpace*, *QubitState*, Circuit, None], default: None) – An optional reference state circuit or any valid initializer for reference_circuit_builder().
- **symbols** (Optional[*SymbolSet*], default: None) – An optional ordered set of symbols that are in the circuit.

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

get_circuit(*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns
Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, ***, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.
- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns

Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, *, *space=None*)

Constructs a single symbolic matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns

Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns

str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type

int

property n_symbols: int

Returns the number of free symbols in the object.

Return type

int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns
QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type
Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs (*symbol_map*)

Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution (*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns
CircuitAnsatz – self, with symbols substituted.

to_QubitState ()

Create a symbolic QubitState representation of the ansatz.

Returns
QubitState – Ansatz as a QubitState.

class TrotterAnsatz (*exponents*, *reference=None*, *symbols=None*, *compiler_passes=DEFAULT_PASS*)

Bases: *GeneralAnsatz*

Ansatz representing a state build up from a product of Pauli-exponentials.

This is at the core of the UCC family of ansatze in inquanto.

Parameters
- **exponents** (*QubitOperatorList*) – Contains exponent and symbol data.

- **reference** (Union[int, List[Qubit], List[int], *QubitSpace*, *QubitState*, Circuit, None], default: None) – An optional reference state circuit or any valid initializer for reference_circuit_builder(…).
- **symbols** (Optional[*SymbolSet*], default: None) – An optional ordered set of symbols that are in the circuit.
- **compiler_passes** (Optional[BasePass], default: DEFAULT_PASS) – An option compiler passes to circuit compilation.

Examples

```
>>> from inquanto.states import QubitState
>>> exponents = QubitOperatorList.from_string("a [(1j, Y0 X2)], b [(1j, Y1 X3)]")
>>> ref = QubitState([1, 1, 0, 0])
>>> ansatz = TrotterAnsatz(exponents, reference=ref)
>>> ansatz.free_symbols_ordered() # returns and ordered set of symbols
SymbolSet([a, b])
>>> ansatz.free_symbols() == {Symbol("a"), Symbol("b")} # returns a set of symbols
True
>>> ansatz.subs("new_{}").free_symbols() == {Symbol("new_a"), Symbol("new_b")} #␣
↪new instance
True
>>> ansatz2 = ansatz.symbol_substitution("new_{}")  # in-place substitution
>>> ansatz2 is ansatz
True
>>> ansatz.free_symbols() == {Symbol("new_a"), Symbol("new_b")}
True
```

DEFAULT_PASS = ***PassType: StandardPass* Preconditions: Specific Postconditions: Generic Postconditions: Default Postcondition: Clear**

The default compilation pass that is applied to the ansatz circuit.

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

property exponents: *QubitOperatorList*

Returns the qubit operator exponents.

Return type
QubitOperatorList

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns

SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type

dict

get_circuit(*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns

Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.
- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns

Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, *, *space=None*)

Constructs a single symbolic matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns
Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()
Returns a hashable string representation of the ansatz object.

Returns
str – Hashable string representation of ansatz.

property n_qubits: int
Returns the number of qubits.

Return type
int

property n_symbols: int
Returns the number of free symbols in the object.

Return type
int

reference_qubit_state()
Create a symbolic QubitState representation of the reference state.

Returns
QubitState – Reference state as a QubitState.

property state_circuit: Circuit
Returns the symbolic state circuit with a default compilation.

Return type
Circuit

property state_symbols: *SymbolSet*
Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs(*symbol_map*)
Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution(*symbol_map=None*)
Performs an in-place symbol substation in the object.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns

TrotterAnsatz – self, with symbols substituted.

to_QubitState()

Create a symbolic QubitState representation of the ansatz.

Returns

QubitState – Ansatz as a QubitState.

to_QubitState_direct(*reverse=False*)

Returns a QubitState object corresponding to the generated state.

This proceeds through direct exponentiation of individual terms and application to the reference state. The ansatz must have been constructed with a QubitState reference.

Danger: In general, this will blow up exponentially.

Parameters

reverse (bool, default: False) – set to True to reverse the order of term application

Returns

QubitState – The Ansatz state.

class FermionSpaceStateExp(*fermion_operator_exponents*, *fock_state*, *qubit_mapping=QubitMappingJordanWigner()*, *qubits=None*, *taperer=None*, *tapering_exponent_check_behaviour='except'*, **args*, ***kwargs*)

Bases: *TrotterAnsatz*

Fermion operator exponentiation (e.g. for UCC). Also initialises state trotterisation.

Qubit tapering can optionally be performed. In order to enable qubit tapering, pass a TapererZ2 object to taperer. Tapering behaviour can be modified by passing tapering_exponent_check_behaviour as specified below.

Parameters

- **fermion_operator_exponents** (*FermionOperatorList*) – Excitation operators (anti-hermitian for UCC).
- **fock_state** (*FermionState*) – Spin orbital occupations.
- **qubit_mapping** (*QubitMapping*, default: QubitMappingJordanWigner()) – How to map fock state operators and states to qubit operators and circuits.
- **qubits** (Optional[List[Qubit]], default: None) – The qubit register used to represent the ansatz state. If no register is provided, a minimal register consisting of qubits indexed from 0 to N, where N is the number of spin-orbitals in the reference state provided. Note that this may include qubits corresponding to spin-orbitals which the excitations do not act on.

- **taperer** (Optional[*TapererZ2*], default: None) – The taperer object used to control how ansatz is tapered. Set to None (default) to skip.
- **tapering_exponent_check_behaviour** (str, default: "except") – controls treatment of exponents which don't commute with the Z2 symmetry operators. Options are:
  - 'except': will test each exponent and throw an exception if any does not commute with the symmetry operators.
  - 'skip': skips exponent testing entirely. This may be dangerous (and is untested) but will be faster when
    exponents are known to be safe.
  - 'discard': will test each exponent and discard any that don't commute with the Z2 symmetry operators. This *should* only
    discard excitations which don't contribute to the ground state, but may be unsafe.
- ***args** – Additional arguments offered by parent object.
- ****kwargs** – Additional keyword arguments offered by parent object.

DEFAULT_PASS = ***PassType: StandardPass* Preconditions: Specific Postconditions: Generic Postconditions: Default Postcondition: Clear**

The default compilation pass that is applied to the ansatz circuit.

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

property exponents: *QubitOperatorList*

Returns the qubit operator exponents.

Return type
QubitOperatorList

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

get_circuit (*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns

Circuit – A circuit that represent the state.

get_numeric_representation (*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.
- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns

Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation (*symbol_map=None*, *, *space=None*)

Constructs a single symbolic matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns

Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns
str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type
int

property n_symbols: int

Returns the number of free symbols in the object.

Return type
int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns
QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type
Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs(*symbol_map*)

Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution(*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns
TrotterAnsatz – `self`, with symbols substituted.

to_QubitState()

Create a symbolic `QubitState` representation of the ansatz.

Returns
QubitState – Ansatz as a `QubitState`.

to_QubitState_direct(*reverse=False*)

Returns a `QubitState` object corresponding to the generated state.

This proceeds through direct exponentiation of individual terms and application to the reference state. The ansatz must have been constructed with a QubitState reference.

> **Danger:** In general, this will blow up exponentially.

Parameters
reverse (`bool`, default: `False`) – set to True to reverse the order of term application

Returns
QubitState – The Ansatz state.

class FermionSpaceAnsatzUCCSD(*fermion_space*, *fermion_state*, *qubit_mapping=QubitMappingJordanWigner()*, **args*, ***kwargs*)

Bases: *FermionSpaceStateExp*

Unitary coupled cluster with singles and doubles excitations (UCCSD), for a given fock space and fock state.

Parameters

- **fermion_space** (`Union`[*FermionSpace*, `int`]) – Spin orbital indices, occupations, and spatial orbitals indices.
- **fermion_state** (*FermionState*) – Spin orbital occupations.
- **qubit_mapping** (*QubitMapping*, default: `QubitMappingJordanWigner()`) – How to map fock state operators and states to qubit operators and circuits.
- ***args** – Additional arguments offered by parent object.
- ****kwargs** – Additional keyword arguments offered by parent object.

DEFAULT_PASS = ***PassType: StandardPass* Preconditions: Specific Postconditions: Generic Postconditions: Default Postcondition: Clear**

The default compilation pass that is applied to the ansatz circuit.

clone()

Performs shallow copy of the object.

Return type
`TypeVar`(`SYMBOLICTYPE`, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
`TypeVar`(`SYMBOLICTYPE`, bound= Symbolic)

property exponents: *QubitOperatorList*

Returns the qubit operator exponents.

Return type
QubitOperatorList

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

get_circuit(*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns
Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.
- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns
Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, ***, *space=None*)

Constructs a single symbolic matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns
Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns
str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type
int

property n_symbols: int

Returns the number of free symbols in the object.

Return type
int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns
QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type
Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs (*symbol_map*)

Returns a new objects with symbols substituted.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type

TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution (*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns

TrotterAnsatz – self, with symbols substituted.

to_QubitState ()

Create a symbolic QubitState representation of the ansatz.

Returns

QubitState – Ansatz as a QubitState.

to_QubitState_direct (*reverse=False*)

Returns a QubitState object corresponding to the generated state.

This proceeds through direct exponentiation of individual terms and application to the reference state. The ansatz must have been constructed with a QubitState reference.

Danger: In general, this will blow up exponentially.

Parameters

reverse (bool, default: False) – set to True to reverse the order of term application

Returns

QubitState – The Ansatz state.

class FermionSpaceAnsatzUCCD (*fermion_space*, *fermion_state*, *qubit_mapping=QubitMappingJordanWigner()*, **args*, ***kwargs*)

Bases: *FermionSpaceStateExp*

Unitary coupled cluster with doubles excitations, no singles (UCCD), for a given fock space and fock state.

Parameters

- **fermion_space** (Union[*FermionSpace*, int]) – Spin orbital indices, occupations, and spatial orbitals indices.

- **fermion_state** (*FermionState*) – Spin orbital occupations.
- **qubit_mapping** (*QubitMapping*, default: QubitMappingJordanWigner()) – How to map fock state operators and states to qubit operators and circuits.
- ***args** – Additional arguments offered by parent object.
- ****kwargs** – Additional keyword arguments offered by parent object.

DEFAULT_PASS = ***PassType: StandardPass* Preconditions: Specific Postconditions: Generic Postconditions: Default Postcondition: Clear**

The default compilation pass that is applied to the ansatz circuit.

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

property exponents: *QubitOperatorList*

Returns the qubit operator exponents.

Return type
QubitOperatorList

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

get_circuit(*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns
Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, ***, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.
- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns

Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, ***, *space=None*)

Constructs a single symbolic matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns

Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns

str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type

int

property n_symbols: int

Returns the number of free symbols in the object.

Return type
int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns
QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type
Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs(*symbol_map*)

Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution(*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns
TrotterAnsatz – self, with symbols substituted.

to_QubitState()

Create a symbolic QubitState representation of the ansatz.

Returns
QubitState – Ansatz as a QubitState.

to_QubitState_direct(*reverse=False*)

Returns a `QubitState` object corresponding to the generated state.

This proceeds through direct exponentiation of individual terms and application to the reference state. The ansatz must have been constructed with a QubitState reference.

> **Danger:** In general, this will blow up exponentially.

Parameters

reverse (`bool`, default: `False`) – set to True to reverse the order of term application

Returns

QubitState – The Ansatz state.

class FermionSpaceStateExpChemicallyAware(*fermion_operator_exponents*, *fermion_state*, *symbols=None*, *compiler_passes=DEFAULT_PASS*)

Bases: *GeneralAnsatz*

Efficient Fermion operator exponentiation (e.g. for UCC). Also initialises state trotterization.

Synthesizes molecular orbital to molecular orbital double excitations uniquely. Changes trotter order of excitations to synthesize circuit with fewer two qubit gates compared to *FermionSpaceStateExp*. Circuit is synthesized in Jordan-Wigner encoding.

Parameters

- **fermion_operator_exponents** (*FermionOperatorList*) – Contains exponents and symbols. Assuming input exponents are ordered as single exponents first, followed by double exponents.
- **fermion_state** (*FermionState*) – Initial fermionic reference state.
- ***args** – Additional arguments offered by parent object.
- ****kwargs** – Additional keyword arguments offered by parent object.
- **symbols** (`Optional`[*SymbolSet*], default: `None`) –
- **compiler_passes** (`Optional`[`BasePass`], default: `DEFAULT_PASS`) –

DEFAULT_PASS = ***PassType: StandardPass* Preconditions: Specific Postconditions: Generic Postconditions: Default Postcondition: Clear**

The default compilation pass that is applied to the ansatz circuit.

clone()

Performs shallow copy of the object.

Return type

`TypeVar`(`SYMBOLICTYPE`, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type

`TypeVar`(`SYMBOLICTYPE`, bound= Symbolic)

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()
Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()
Returns a dict with data describing state object.

Return type
dict

get_circuit(*symbol_map=None*)
Constructs a single symbolic state circuit.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns
Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*)
Constructs a single numeric matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters
- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.
- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns
Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, *, *space=None*)
Constructs a single symbolic matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns

Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns

str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type

int

property n_symbols: int

Returns the number of free symbols in the object.

Return type

int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns

QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type

Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type

SymbolSet

subs(*symbol_map*)

Returns a new objects with symbols substituted.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution(*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns
FermionSpaceStateExpChemicallyAware – self, with symbols substituted.

to_QubitState()

Create a symbolic QubitState representation of the ansatz.

Returns
QubitState – Ansatz as a QubitState.

class FermionSpaceAnsatzChemicallyAwareUCCSD(*fermion_space*, *fermion_state*, **args*, ***kwargs*)

Bases: *FermionSpaceStateExpChemicallyAware*

Chemically Aware Unitary coupled cluster with singles and doubles excitations (UCCSD).

Builds ansatz for a given fermion space and fermion state. Circuit is synthesized in Jordan-Wigner encoding.

Parameters
- **fermion_space** (Union[*FermionSpace*, int]) – Spin orbital indices, occupations, and spatial orbitals indices.
- **fermion_state** (*FermionState*) – Spin orbital occupations.
- ***args** – Additional arguments offered by parent object.
- ****kwargs** – Additional keyword arguments offered by parent object.

DEFAULT_PASS = ***PassType: StandardPass* Preconditions: Specific Postconditions: Generic Postconditions: Default Postcondition: Clear**

The default compilation pass that is applied to the ansatz circuit.

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

get_circuit(*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns
Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.
- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns
Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, *, *space=None*)

Constructs a single symbolic matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns

Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns

str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type

int

property n_symbols: int

Returns the number of free symbols in the object.

Return type

int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns

QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type

Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type

SymbolSet

subs(*symbol_map*)

Returns a new objects with symbols substituted.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float],

Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution(*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns
FermionSpaceStateExpChemicallyAware – self, with symbols substituted.

to_QubitState()

Create a symbolic QubitState representation of the ansatz.

Returns
QubitState – Ansatz as a QubitState.

class FermionSpaceAnsatzkUpCCGD(*fermion_space*, *fermion_state*, *k_input*, *qubit_mapping=QubitMappingJordanWigner()*, **args*, ***kwargs*)

Bases: *FermionSpaceStateExp*

k-UpCCGD Ansatz.

Ansatz consisting of k factors of variationally independent unitary coupled cluster operators with generalised spin-paired doubles excitations, no singles (k-UpCCGD).

Parameters
- **fermion_space** (Union[*FermionSpace*, int]) – Spin orbital indices, occupations, and spatial orbitals indices.
- **fermion_state** (*FermionState*) – Spin orbital occupations.
- **k_input** (int) – Value of k in k-UpCC; results in k cluster operators.
- **qubit_mapping** (*QubitMapping*, default: QubitMappingJordanWigner()) – How to map fock state operators and states to qubit operators and circuits.
- ***args** – Additional arguments offered by parent object.
- ****kwargs** – Additional keyword arguments offered by parent object.

DEFAULT_PASS = ***PassType: StandardPass* Preconditions: Specific Postconditions: Generic Postconditions: Default Postcondition: Clear**

The default compilation pass that is applied to the ansatz circuit.

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

property exponents: *QubitOperatorList*

Returns the qubit operator exponents.

Return type
QubitOperatorList

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

get_circuit(*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns
Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.

- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns

Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, ***, *space=None*)

Constructs a single symbolic matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns

Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns

str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type

int

property n_symbols: int

Returns the number of free symbols in the object.

Return type

int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns

QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type

Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs (*symbol_map*)

Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution (*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns
TrotterAnsatz – self, with symbols substituted.

to_QubitState ()

Create a symbolic QubitState representation of the ansatz.

Returns
QubitState – Ansatz as a QubitState.

to_QubitState_direct (*reverse=False*)

Returns a QubitState object corresponding to the generated state.

This proceeds through direct exponentiation of individual terms and application to the reference state. The ansatz must have been constructed with a QubitState reference.

Danger: In general, this will blow up exponentially.

Parameters
reverse (bool, default: False) – set to True to reverse the order of term application

Returns
QubitState – The Ansatz state.

class FermionSpaceAnsatzkUpCCGSD (*fermion_space*, *fermion_state*, *k_input*, *qubit_mapping=QubitMappingJordanWigner()*, **args*, ***kwargs*)

Bases: *FermionSpaceStateExp*

k-UpCCGSD ansatz.

UCC Ansatz consisting of k factors of variationally independent unitary coupled cluster operators with fully generalised singles and spin-paired doubles excitations (k-UpCCGSD). See https://arxiv.org/abs/1810.02327 for more details.

Parameters

- **fermion_space** (Union[*FermionSpace*, int]) – Spin orbital indices, occupations, and spatial orbitals indices.
- **fermion_state** (*FermionState*) – Spin orbital occupations.
- **k_input** (int) – Value of k in k-UpCC; results in k cluster operators.
- **qubit_mapping** (*QubitMapping*, default: QubitMappingJordanWigner()) – How to map fock state operators and states to qubit operators and circuits.
- ***args** – Additional arguments offered by parent object.
- ****kwargs** – Additional keyword arguments offered by parent object.

DEFAULT_PASS = ***PassType: StandardPass* Preconditions: Specific Postconditions: Generic Postconditions: Default Postcondition: Clear**

The default compilation pass that is applied to the ansatz circuit.

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

property exponents: *QubitOperatorList*

Returns the qubit operator exponents.

Return type
QubitOperatorList

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

get_circuit(*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns
Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.
- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns
Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, *, *space=None*)

Constructs a single symbolic matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns
Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()
Returns a hashable string representation of the ansatz object.

Returns
str – Hashable string representation of ansatz.

property n_qubits: int
Returns the number of qubits.

Return type
int

property n_symbols: int
Returns the number of free symbols in the object.

Return type
int

reference_qubit_state()
Create a symbolic QubitState representation of the reference state.

Returns
QubitState – Reference state as a QubitState.

property state_circuit: Circuit
Returns the symbolic state circuit with a default compilation.

Return type
Circuit

property state_symbols: *SymbolSet*
Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs(*symbol_map*)
Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution(*symbol_map=None*)
Performs an in-place symbol substation in the object.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns
TrotterAnsatz – `self`, with symbols substituted.

to_QubitState()

Create a symbolic `QubitState` representation of the ansatz.

Returns
QubitState – Ansatz as a `QubitState`.

to_QubitState_direct(*reverse=False*)

Returns a `QubitState` object corresponding to the generated state.

This proceeds through direct exponentiation of individual terms and application to the reference state. The ansatz must have been constructed with a QubitState reference.

Danger: In general, this will blow up exponentially.

Parameters
reverse (`bool`, default: `False`) – set to True to reverse the order of term application

Returns
QubitState – The Ansatz state.

class FermionSpaceAnsatzkUpCCGSDSinglet(*fermion_space*, *fermion_state*, *k_input*, *qubit_mapping=QubitMappingJordanWigner()*, **args*, ***kwargs*)

Bases: *FermionSpaceStateExp*

k-UpCCGSD ansatz.

Ansatz consisting of k factors of variationally independent unitary coupled cluster operators with fully generalised singles and spin paired doubles excitations (k-UpCCGSD). Single excitations are adapted to singlets, i.e. alpha-alpha and beta-beta excitations between a given pair of spatial orbitals have the same parameter.

Parameters

- **fermion_space** (`Union`[*FermionSpace*, `int`]) – Spin orbital indices, occupations, and spatial orbitals indices.
- **fermion_state** (*FermionState*) – Spin orbital occupations.
- **k_input** (`int`) – Value of k in k-UpCC; results in k cluster operators.
- **qubit_mapping** (*QubitMapping*, default: `QubitMappingJordanWigner()`) – How to map fock state operators and states to qubit operators and circuits.
- ***args** – Additional arguments offered by parent object.
- ****kwargs** – Additional keyword arguments offered by parent object.

DEFAULT_PASS = ***PassType: StandardPass* Preconditions: Specific Postconditions: Generic Postconditions: Default Postcondition: Clear**

The default compilation pass that is applied to the ansatz circuit.

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

property exponents: *QubitOperatorList*

Returns the qubit operator exponents.

Return type
QubitOperatorList

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

get_circuit(*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns
Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.

- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.
- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns
Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, *, *space=None*)

Constructs a single symbolic matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns
Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns
str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type
int

property n_symbols: int

Returns the number of free symbols in the object.

Return type
int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns
QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type
Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs(*symbol_map*)

Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution(*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns
TrotterAnsatz – self, with symbols substituted.

to_QubitState()

Create a symbolic QubitState representation of the ansatz.

Returns
QubitState – Ansatz as a QubitState.

to_QubitState_direct(*reverse=False*)

Returns a QubitState object corresponding to the generated state.

This proceeds through direct exponentiation of individual terms and application to the reference state. The ansatz must have been constructed with a QubitState reference.

Danger: In general, this will blow up exponentially.

Parameters
reverse (bool, default: False) – set to True to reverse the order of term application

Returns
QubitState – The Ansatz state.

class FermionSpaceAnsatzUCCGD(*fermion_space*, *fermion_state*, *qubit_mapping=QubitMappingJordanWigner()*, **args*, ***kwargs*)

Bases: *FermionSpaceStateExp*

UCC Ansatz with fully generalised doubles excitations, no singles (UCCGD).

Here, generalised means occupied and virtual orbital subspaces are undistinguished, hence occupied-occupied and virtual-virtual excitations are allowed. See https://arxiv.org/abs/1810.02327 for more details.

Parameters

- **fermion_space** (Union[*FermionSpace*, int]) – Spin orbital indices, occupations, and spatial orbitals indices.
- **fermion_state** (*FermionState*) – Spin orbital occupations.
- **qubit_mapping** (*QubitMapping*, default: QubitMappingJordanWigner()) – How to map fock state operators and states to qubit operators and circuits.
- ***args** – Additional arguments offered by parent object.
- ****kwargs** – Additional keyword arguments offered by parent object.

DEFAULT_PASS = ***PassType: StandardPass* Preconditions: Specific Postconditions: Generic Postconditions: Default Postcondition: Clear**

The default compilation pass that is applied to the ansatz circuit.

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

property exponents: *QubitOperatorList*

Returns the qubit operator exponents.

Return type
QubitOperatorList

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

get_circuit (*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns

Circuit – A circuit that represent the state.

get_numeric_representation (*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.
- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns

Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation (*symbol_map=None*, *, *space=None*)

Constructs a single symbolic matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns

Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns
str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type
int

property n_symbols: int

Returns the number of free symbols in the object.

Return type
int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns
QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type
Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs(*symbol_map*)

Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution(*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns
TrotterAnsatz – `self`, with symbols substituted.

to_QubitState()

Create a symbolic `QubitState` representation of the ansatz.

Returns
QubitState – Ansatz as a `QubitState`.

to_QubitState_direct(*reverse=False*)

Returns a `QubitState` object corresponding to the generated state.

This proceeds through direct exponentiation of individual terms and application to the reference state. The ansatz must have been constructed with a QubitState reference.

> **Danger:** In general, this will blow up exponentially.

Parameters
reverse (`bool`, default: `False`) – set to True to reverse the order of term application

Returns
QubitState – The Ansatz state.

class FermionSpaceAnsatzUCCGSD(*fermion_space*, *fermion_state*, *qubit_mapping=QubitMappingJordanWigner()*, **args*, ***kwargs*)

Bases: *FermionSpaceStateExp*

UCC Ansatz with fully generalised singles and doubles excitations (UCCGSD).

Here, generalised means occupied and virtual orbital subspaces are undistinguished, hence occupied-occupied and virtual-virtual excitations are allowed. See https://arxiv.org/abs/1810.02327 for more details.

Parameters

- **fermion_space** (`Union`[*FermionSpace*, `int`]) – Spin orbital indices, occupations, and spatial orbitals indices.
- **fermion_state** (*FermionState*) – Spin orbital occupations.
- **qubit_mapping** (*QubitMapping*, default: `QubitMappingJordanWigner()`) – How to map fock state operators and states to qubit operators and circuits.
- ***args** – Additional arguments offered by parent object.
- ****kwargs** – Additional keyword arguments offered by parent object.

DEFAULT_PASS = ***PassType: StandardPass* Preconditions: Specific Postconditions: Generic Postconditions: Default Postcondition: Clear**

The default compilation pass that is applied to the ansatz circuit.

clone()

Performs shallow copy of the object.

Return type
`TypeVar`(`SYMBOLICTYPE`, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

property exponents: *QubitOperatorList*

Returns the qubit operator exponents.

Return type
QubitOperatorList

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

get_circuit(*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns
Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.

- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns

Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, ***, *space=None*)

Constructs a single symbolic matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns

Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns

str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type

int

property n_symbols: int

Returns the number of free symbols in the object.

Return type

int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns

QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type

Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs (*symbol_map*)

Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution (*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns
TrotterAnsatz – self, with symbols substituted.

to_QubitState ()

Create a symbolic QubitState representation of the ansatz.

Returns
QubitState – Ansatz as a QubitState.

to_QubitState_direct (*reverse=False*)

Returns a QubitState object corresponding to the generated state.

This proceeds through direct exponentiation of individual terms and application to the reference state. The ansatz must have been constructed with a QubitState reference.

Danger: In general, this will blow up exponentially.

Parameters
reverse (bool, default: False) – set to True to reverse the order of term application

Returns
QubitState – The Ansatz state.

class FermionSpaceAnsatzUCCSDSinglet (*fermion_space*, *fermion_state*, *qubit_mapping=QubitMappingJordanWigner()*, **args*, ***kwargs*)

Bases: *FermionSpaceStateExp*

Unitary coupled cluster with singles and doubles excitations (UCCSD), for a given fock space and fock state.

Adapted to singlets: alpha-alpha and beta-beta single excitations between a given pair of spatial orbitals have the same parameter.

Parameters

- **fermion_space** (Union[*FermionSpace*, int]) – Spin orbital indices, occupations, and spatial orbitals indices.
- **fermion_state** (*FermionState*) – Spin orbital occupations.
- **qubit_mapping** (*QubitMapping*, default: QubitMappingJordanWigner()) – How to map fock state operators and states to qubit operators and circuits.
- ***args** – Additional arguments offered by parent object.
- ****kwargs** – Additional keyword arguments offered by parent object.

DEFAULT_PASS = ***PassType: StandardPass* Preconditions: Specific Postconditions: Generic Postconditions: Default Postcondition: Clear**

The default compilation pass that is applied to the ansatz circuit.

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

property exponents: *QubitOperatorList*

Returns the qubit operator exponents.

Return type
QubitOperatorList

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

get_circuit (*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns

Circuit – A circuit that represent the state.

get_numeric_representation (*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.
- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns

Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation (*symbol_map=None*, *, *space=None*)

Constructs a single symbolic matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns

Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns
str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type
int

property n_symbols: int

Returns the number of free symbols in the object.

Return type
int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns
QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type
Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs(*symbol_map*)

Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution(*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns
TrotterAnsatz – `self`, with symbols substituted.

to_QubitState()

Create a symbolic `QubitState` representation of the ansatz.

Returns
QubitState – Ansatz as a `QubitState`.

to_QubitState_direct(*reverse=False*)

Returns a `QubitState` object corresponding to the generated state.

This proceeds through direct exponentiation of individual terms and application to the reference state. The ansatz must have been constructed with a QubitState reference.

> **Danger:** In general, this will blow up exponentially.

Parameters
reverse (`bool`, default: `False`) – set to True to reverse the order of term application

Returns
QubitState – The Ansatz state.

class HamiltonianVariationalAnsatz(*fermion_state*, *hamiltonian_operator=FermionOperator()*, *qubit_mapping=QubitMappingJordanWigner()*, *s=1*, **args*, ***kwargs*)

Bases: *TrotterAnsatz*

Variational Hamiltonian Ansatz introduced in https://arxiv.org/abs/1507.08969.

Parameters

- **fermion_state** (*FermionState*) – Spin orbital occupations. If applied to molecules, this should be represent a single configuration mean field state.
- **qubit_mapping** (*QubitMapping*, default: `QubitMappingJordanWigner()`) – How to map fock state operators and states to qubit operators and circuits.
- **hamiltonian_operator** (*FermionOperator*, default: `FermionOperator()`) – The hamiltonian operator produced by driver.
- **s** (`int`, default: `1`) – The number of terms in the multiplication; the total number of parameters will be 5*s.
- ***args** – Additional arguments offered by parent object.
- ****kwargs** – Additional keyword arguments offered by parent object.

DEFAULT_PASS = ***PassType: StandardPass* Preconditions: Specific Postconditions: Generic Postconditions: Default Postcondition: Clear**

The default compilation pass that is applied to the ansatz circuit.

clone()

Performs shallow copy of the object.

Return type
`TypeVar`(`SYMBOLICTYPE`, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

property exponents: *QubitOperatorList*

Returns the qubit operator exponents.

Return type
QubitOperatorList

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

get_circuit(*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns
Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.

- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns

Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, *, *space=None*)

Constructs a single symbolic matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns

Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns

str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type

int

property n_symbols: int

Returns the number of free symbols in the object.

Return type

int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns

QubitState – Reference state as a QubitState.

property split_hamiltonian: Tuple[*FermionOperator*]

Split hamiltonian into diagonal and non-diagonal elements.

Hamiltonian Variational Ansatz based on separating parts of the Hamiltonian to Hdiag, Hhop, Hex *FermionOperator.terms* format: {(#orb,up/down=1/0) : coef}

Diagonal terms:
$H_{diag} = \sum_p (e_p a_p^\dagger a_p) + \sum_{p,q} (h_{pqqp} a_p^\dagger a_p a_q^\dagger a_q)$
in terms of *FermionOperator*:
$((p,1),(p,0)) : e_p$ and $((p,1),(q,1),(p,0),(q,0)) : h_{pqqp}$

Hopping terms:
$H_{hop} = \sum_{p,q} (e_{p,q} a_p^\dagger a_q) + \sum_{p,r,q} (h_{prrp} a_p^\dagger a_q a_r^\dagger a_r)$
in terms of *FermionOperator*:
$((p,1),(q,0)) : e_p, q$ and $((p,1),(q,1),(q,0),(r,0)) : h_{pqqp}$

Exchange terms:
$H_{ex} = \sum_{p,q,r,s} (h_{pqrs} a_p^\dagger a_r^\dagger a_r a_s)$
in terms of *FermionOperator*:
$(((p,1),(q,1),(r,0),(s,0)) : h_{pqrs}$

Returns
Tuple[*FermionOperator*] – Diagonal, hopping, and exchange operators extracted from the Hamiltonian.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type
Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs (*symbol_map*)

Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution (*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns
TrotterAnsatz – `self`, with symbols substituted.

to_QubitState()

Create a symbolic `QubitState` representation of the ansatz.

Returns
QubitState – Ansatz as a `QubitState`.

to_QubitState_direct(*reverse=False*)

Returns a `QubitState` object corresponding to the generated state.

This proceeds through direct exponentiation of individual terms and application to the reference state. The ansatz must have been constructed with a QubitState reference.

> **Danger:** In general, this will blow up exponentially.

Parameters
reverse (`bool`, default: `False`) – set to True to reverse the order of term application

Returns
QubitState – The Ansatz state.

class LayeredAnsatz(*rotations_def*, *reference*, *n_layers=1*, *inter_block_entangler=True*, *entangler_def=None*, *cap_layers=True*, *cap_def=None*, *symbols=None*, *compiler_passes=DecomposeBoxes()*)

Bases: *GeneralAnsatz*

Parent class for anszate with k-replicas of a group of circuit primitives.

Hierarchy:
Block (CustomGateDef): Consists of a collection of parametric pytket.circuit.OpType operations added to a circuit. Layer (CircBox): Consists of a collection of blocks. If inter_block_entangler=True, blocks are

seperated by a group of two_qubit operations.

Parameters

- **rotations_def** (`List`[`CustomGateDef`]) – A list of circuit_primitives as CustomGateDefs to be placed inside a layer.
- **reference** (`Union`[`int`, `List`[`Qubit`], `List`[`int`], *QubitSpace*, *QubitState*, `Circuit`]) – An reference state circuit or any valid initializer for reference_circuit_builder(…).
- **n_layers** (`int`, default: `1`) – Number of layers of blocks to be added to circuit.
- **inter_block_entangler** (`bool`, default: `True`) – If True add a primitive consisting of two-qubit gates inbetween blocks within a layer.
- **block_entanglers** – Definition of the entangler primitive
- **cap_layers** (`bool`, default: `True`) – If True end circuit by adding one extra set of blocks.
- **cap_def** (`Optional`[`List`[`CustomGateDef`]], default: `None`) – A list of circuit_primitives as CustomGateDefs for the cap.
- **symbols** (`Optional`[*SymbolSet*], default: `None`) – An ordered set of symbols that are in the circuit.

- **compiler_passes** (Optional[BasePass], default: DecomposeBoxes()) – An option compiler passes to circuit compilation.
- **entangler_def** (Optional[CustomGateDef], default: None) –

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

get_circuit(*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns
Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, ***, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.
- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns
Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation (*symbol_map=None*, ***, *space=None*)

Constructs a single symbolic matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns
Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns
str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type
int

property n_symbols: int

Returns the number of free symbols in the object.

Return type
int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns
QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type
Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs (*symbol_map*)

Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution (*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns
LayeredAnsatz – self, with symbols substituted.

to_QubitState()

Create a symbolic QubitState representation of the ansatz.

Returns
QubitState – Ansatz as a QubitState.

class HardwareEfficientAnsatz (*rotation_operations*, *reference*, *n_layers*, *entangler_operation=OpType.CX*, **args*, ***kwargs*)

Bases: *LayeredAnsatz*

Hardware Efficient Ansatz as defined in https://arxiv.org/abs/1704.05018.

Parameters
- **rotation_operations** (List[OpType]) – Use blocks for Ry, RxRy and RxRyRz.
- **qubit_state** – Initial state of qubit register.
- **n_layers** (int) – Number of layers.
- **entangler_operation** (OpType, default: OpType.CX) – The operation to build the entangler primitive. Default value is a CX gate
- ***args** – Additional arguments offered by parent object.
- ****kwargs** – Additional keyword arguments offered by parent object.

- **reference** (Union[int, List[Qubit], List[int], *QubitSpace*, *QubitState*, Circuit]) –

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

get_circuit(*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns
Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.

- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns

Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, ***, *space=None*)

Constructs a single symbolic matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns

Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns

str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type

int

property n_symbols: int

Returns the number of free symbols in the object.

Return type

int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns

QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type

Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs (*symbol_map*)

Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution (*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns
LayeredAnsatz – self, with symbols substituted.

to_QubitState ()

Create a symbolic QubitState representation of the ansatz.

Returns
QubitState – Ansatz as a QubitState.

class MultiReferenceState (*input_states*)

Bases: *GeneralAnsatz*

Arbitrary state preparation using Givens rotations.

https://arxiv.org/pdf/2106.13839.pdf

Note:

Currently supports only basis states which have the same number of 1s in the bit string. For JW encoding this
corresponds to states with the same particle number.

Parameters
input_states (*QubitState*) – Multi-reference qubit state input.

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

get_circuit(*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns
Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.

- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns

Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, *, *space=None*)

Constructs a single symbolic matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns

Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns

str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type

int

property n_symbols: int

Returns the number of free symbols in the object.

Return type

int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns

QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type

Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs (*symbol_map*)

Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution (*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns
TypeVar(SYMBOLICTYPE, bound= Symbolic) – self, with symbols substituted.

to_QubitState ()

Create a symbolic QubitState representation of the ansatz.

Returns
QubitState – Ansatz as a QubitState.

class GivensAnsatz (*configurations*, *reference=None*, *symbols=None*, *compiler_passes=DecomposeBoxes()*)

Bases: *GeneralAnsatz*

Arbitrary state preparation using Givens rotations.

https://arxiv.org/pdf/2106.13839.pdf

Note:

Currently supports only basis states which have the same number of 1s in the bit string. For JW encoding this
corresponds to states with the same particle number.

Parameters

- **configurations** (Union[List[*QubitStateString*], List[List[int]]]) – List of bit strings or QubitStateString objects providing the basis for the ansatz linear combination.

- **reference** (Union[int, List[Qubit], List[int], *QubitSpace*, *QubitState*, Circuit, None], default: None) – An optional reference state circuit or any valid initializer for reference_circuit_builder(…).
- **symbols** (Optional[*SymbolSet*], default: None) – An optional ordered set of symbols that are in the circuit.
- **compiler_passes** (Optional[BasePass], default: DecomposeBoxes()) – An option compiler passes to circuit compilation.

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

free_symbols()

Returns the free symbols in the object.

Returns
Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns
SymbolSet – Ordered free symbols in object.

generate_report()

Returns a dict with data describing state object.

Return type
dict

get_circuit(*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns
Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, ***, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.
- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns

Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, *, *space=None*)

Constructs a single symbolic matrix/vector representation.

> **Danger:** This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns

Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns

str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type

int

property n_symbols: int

Returns the number of free symbols in the object.

Return type

int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns
QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type
Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs (*symbol_map*)

Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution (*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns
GivensAnsatz – self, with symbols substituted.

to_QubitState ()

Create a symbolic QubitState representation of the ansatz.

Returns
QubitState – Ansatz as a QubitState.

class RealBasisRotationAnsatz (*reference*, *symbols=None*, *compiler_passes=DecomposeBoxes()*)

Bases: *GeneralAnsatz*

Basis rotation ansatz, for which the rotation matrix is a real unitary.

Constructs the ansatz $\hat{R}|\Psi\rangle$ where $\hat{R}$ is given by the Thouless theorem: $\hat{R} = \exp \sum_{ij} [\ln U]_{ij} a_i^\dagger a_j$ and U is a single-particle basis rotation matrix (real, unitary). See https://arxiv.org/abs/1711.04789.

Parameters

- **reference** (Union[int, List[Qubit], List[int], *QubitSpace*, *QubitState*, Circuit]) – Initial reference state, $|\Psi\rangle$ above.

- **symbols** (Optional[*SymbolSet*], default: None) – An optional ordered set of symbols that are in the circuit.
- **compiler_passes** (Optional[BasePass], default: DecomposeBoxes()) – An option compiler passes to circuit compilation.

ansatz_parameters_from_unitary(*rotation_unitary*)

Converts a real unitary matrix to ansatz parameters.

Performs a QR decomposition of the rotation matrix to find ansatz parameters.

Parameters

rotation_unitary (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – A real unitary matrix with the dimensions of number of qubits.

Returns

SymbolDict – A SymbolDict of the ansatz parameters.

clone()

Performs shallow copy of the object.

Return type

TypeVar(SYMBOLICTYPE, bound= Symbolic)

copy()

Performs deep copy of the object.

Return type

TypeVar(SYMBOLICTYPE, bound= Symbolic)

free_symbols()

Returns the free symbols in the object.

Returns

Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols as SymbolSet.

Returns

SymbolSet – Ordered free symbols in object.

classmethod from_basis_rotation(*basis_rotation_unitary*, ***kwargs*)

Builds a parameterless circuit ansatz from basis_rotation_unitary matrix.

Parameters

- **basis_rotation_unitary** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – A basis rotation matrix
- ****kwargs** – Additional parameters for an ansatz construction.

Returns

CircuitAnsatz – A circuit ansatz representing the basis rotation matrix.

generate_report()

Returns a dict with data describing state object.

Return type

dict

get_circuit(*symbol_map=None*)

Constructs a single symbolic state circuit.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – An optional symbol mapping for substitution.

Returns

Circuit – A circuit that represent the state.

get_numeric_representation(*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.
- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns

Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, *, *space=None*)

Constructs a single symbolic matrix/vector representation.

Danger: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object.

Returns

Expr – A symbolic expression as a representation, which is a symbolic NDArray.

make_hashable()

Returns a hashable string representation of the ansatz object.

Returns
str – Hashable string representation of ansatz.

property n_qubits: int

Returns the number of qubits.

Return type
int

property n_symbols: int

Returns the number of free symbols in the object.

Return type
int

reference_qubit_state()

Create a symbolic QubitState representation of the reference state.

Returns
QubitState – Reference state as a QubitState.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

Return type
Circuit

property state_symbols: *SymbolSet*

Returns the ordered parameter symbols this state uses.

Return type
SymbolSet

subs(*symbol_map*)

Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution(*symbol_map=None*)

Performs an in-place symbol substation in the object.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – Dictionary or Callable mapping existing symbols to new symbols or values.

Note: While this is an in-place operation, consistency in free_symbols_ordered() is not guaranteed.

Returns
RealBasisRotationAnsatz – self, with symbols substituted.

to_QubitState()

Create a symbolic QubitState representation of the ansatz.

Returns

QubitState – Ansatz as a QubitState.

basis_rotation_to_circuit(*rotation_unitary*, *orbital=False*)

Converts a real unitary basis rotation matrix to a circuit.

Note: This assumes the use of a Jordan-Wigner transformation.

Parameters

- **rotation_unitary** (ndarray[Any, dtype[TypeVar(ScalarType, bound=generic, covariant=True)]]) – A real unitary matrix with the dimensions of number of orbitals or spin-orbitals.
- **orbital** (bool, default: False) – If true the rotation matrix is only for spatial orbitals, and it will be extended to spin-orbitals internally, therefore the number of qubits will be twice as the dimension.

Returns

Circuit – A circuit representing the basis rotation matrix, the number of qubits equals to the number of spin-orbitals.

22.3 inquanto.computables

Module for quantum computable expressions.

class ComputableArray(*array*)

Bases: Computable

Expression for an ndarray of other expressions.

Accepts ndarrays of any dimension.

Parameters

array (ndarray) – Input ndarray of computable expressions.

ALLOWED_PROTOCOLS = ()

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) –

Return type

ComputableArray

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns
True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns
Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns
List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns
float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

cost_estimate_elementwise(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the array on a device/simulator.

The costs returned are the costs per element if they are individual experiments without redundancies removed. Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuits to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns

ndarray[Any, dtype[float64]] – The cost in quantinuum credits for evaluating each element of the array, in the shape of the array.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns

Value of the measurement.

evaluate(*parameters=None*)

Evaluate the result of a calculation.

Once the experiment has been run, this function evaluates the computable expression and returns its numerical values.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional symbols to value mapping, if the expression has symbolic part

Returns

Value of the computables

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.

- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves(*measuarable_type=None*, *condition=None*)

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

print_tree()

Print the computable dependence tree.

Return type
None

rebuild (*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

retrieve_distributions (*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns

List[List[ResultHandle]] – Results distributions.

run (*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns

Computable – self.

symbolic_evaluate (*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker (*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

class ComputableCommutator (*state*, *operator_left*, *operator_right*)

Bases: ComputableExpression

Measure the commutator between two QubitOperators.

Parameters

- **state** (*GeneralAnsatz*) – Trial state to use.
- **operator_left** (*QubitOperator*) – Operator on the left hand-side.
- **operator_right** (*QubitOperator*) – Operator on the right hand-side.

ALLOWED_PROTOCOLS = (<class 'inquanto.protocols._protocol_standard.ProtocolDirect'>, <class 'inquanto.protocols._protocol_standard.ProtocolIndirect'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparse'>, <class 'inquanto.protocols._protocol.ProtocolStateVectorBackendSupport'>)

Protocols this computable are compatible with.

approximate_error (*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) –

Return type
Any

build (*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns

True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns

Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns

List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns

float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns

Value of the measurement.

evaluate(*parameters=None*)

Evaluate the result of a calculation.

Once the experiment has been run, this function evaluates the computable expression and returns its numerical values.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional symbols to value mapping, if the expression has symbolic part

Returns

complex – Value of the computables

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns

Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type

bool

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves (*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

print_tree ()

Print the computable dependence tree.

Return type
None

rebuild (*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

retrieve_distributions (*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns
List[List[ResultHandle]] – Results distributions.

run (*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.

- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Computable – self.

symbolic_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker(*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

class ComputableList(*computables*)

Bases: Computable

Combines multiple computables into one as a list, for one unified run and evaluation.

Parameters
computables (List[Computable]) – A list of other computables to be combined.

ALLOWED_PROTOCOLS = ()

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Estimates the standard error of each elements after the measurements have been run.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional parameters to substitute post-evaluation.

Note: Not all protocols and computables can compute valid standard errors.

Returns
List – A list of standard errors.

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns

True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns

Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns

List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.

- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns

float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

cost_estimate_elementwise(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable list on a device/simulator.

The costs returned are the costs per element if they are individual experiments without redundancies removed. Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns

List[float] – The cost in Quantinuum credits for evaluating the computables as a list of costs.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns

Value of the measurement.

evaluate(*parameters=None*)

Evaluates the computable and returns its actual value after the measurements have been run.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional parameters to substitute post-evaluation.

Returns

List – A list of the values of the computables it contains.

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.

- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves(*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

print_tree()

Print the computable dependence tree.

Return type
None

rebuild (*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

retrieve_distributions (*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns

List[List[ResultHandle]] – Results distributions.

run (*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns

Computable – self.

symbolic_evaluate (*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker (*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

class ComputableMetricTensorReal (*state*, *symbols*)

Bases: ComputableSuper

Computes the real part of the metric tensor.

Calculates: $\frac{1}{4}\Re < \partial_{\theta_i}\Psi(\theta)|\partial_{\theta_j}\Psi > (\theta)$ for all i, j.

The computable is supported by the following protocols:

- ProtocolHadamardDerivativeOverlap
- ProtocolStateVectorSparseLegacy
- ProtocolSymbolic

Parameters

- **state** (*GeneralAnsatz*) – State Psi for the expectation value.
- **symbols** (Iterable[Symbol]) – List of symbols with respect to the derivatives are computed.

ALLOWED_PROTOCOLS = [<class 'inquanto.protocols._protocol_derivative.ProtocolHadamardDerivativeOverlap'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparse'>, <class 'inquanto.protocols._protocol_symbolic.ProtocolSymbolic'>]

Protocols this computable are compatible with.

approximate_error (*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) –

build (*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
`Computable` – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (`Union[str, List]`) – Returned data of the launch or the filename of json file.

Returns
True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns
`Computable` – self.

collect_protocols()

Collect all the protocols to be measured.

Returns
`List[Protocol]` – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (`Backend`) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (`Optional[SymbolDict]`) – The parameters of the circuit to be evaluated.
- **n_shots** (`int`, default: `8192`) – The number of shots.
- **compiler_passes** (`Optional[BasePass]`, default: `None`) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (`Optional[str]`, default: `None`) – Which syntax checker to use. The default is None.
- **use_websocket** (`Optional[bool]`, default: `None`) – Whether to use a web connection.

Returns
`float` – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters
parameters (`Union[SymbolDict, Dict, None]`, default: `None`) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

evaluate (*parameters=None*)

Evaluate the result of a calculation.

Once the experiment has been run, this function evaluates the computable expression and returns its numerical values.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional symbols to value mapping, if the expression has symbolic part

Returns
ndarray – Value of the computables

generate_circuits (*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf ()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch (*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.

- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves (*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

print_tree ()

Print the computable dependence tree.

Return type
None

rebuild (*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

retrieve_distributions (*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns
List[List[ResultHandle]] – Results distributions.

run (*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.

- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Computable – self.

symbolic_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker(*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

class ComputablePDM1234Real(*space*, *ansatz*, *encoding*, *symmetry_operators*, *cas_elec*, *cas_orbs*, *cu4=True*, *taperer=None*)

Bases: ComputableSuper

Measures 1,2,3-PDMs for a given state, defined by ansatz and parameters.

Estimates 4-RDM via the cumulant expansion or measures it.

Parameters

- **space** (*FermionSpace*) – Fermion space where the operators are defined.
- **ansatz** (*GeneralAnsatz*) – Ansatz with respect to which the expectation values are computed.
- **encoding** (*QubitMapping*) – Qubit encoding from fermion space to qubit space.
- **symmetry_operators** (List[*SymmetryOperatorPauli*]) – Z2 symmetries of the Hamiltonian.
- **cas_elec** (int) – Number of active electrons.
- **cas_orbs** (int) – Number of active orbitals.
- **cu4** (bool, default: True) – If True (default), 4-RDM is estimated via the cumulant expansion approximation, otherwise it is measured.

- **taperer** (Optional[*TapererZ2*], default: None) – TapererZ2 initialized with the Hamiltonian, or None if tapering is not used.

Returns

1,2,3,4-PDMs (or RDMs) as a list of arrays, in PySCF's ordering (<p^r^sq>)

ALLOWED_PROTOCOLS = (<class 'inquanto.protocols._protocol_standard.ProtocolDirect'>, <class 'inquanto.protocols._protocol_standard.ProtocolIndirect'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>, <class 'inquanto.protocols._protocol.ProtocolStateVectorBackendSupport'>)

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) –

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns

True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns

Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns

List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns

float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns

Value of the measurement.

evaluate(*parameters=None*)

Evaluate the result of a calculation.

Once the experiment has been run, this function evaluates the computable expression and returns its numerical values.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional symbols to value mapping, if the expression has symbolic part

Returns

Value of the computables

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.

- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves(*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

print_tree()

Print the computable dependence tree.

Return type
None

rebuild(*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

retrieve_distributions (*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns
List[List[ResultHandle]] – Results distributions.

run (*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Computable – self.

symbolic_evaluate (*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker(*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

class ComputableQSEMatrices(*state*, *hermitian_operator*, *expansion_operators*)

Bases: ComputableSuper

Computable QSE (quantum subspace expansion) matrices.

Measure matrix elements of a matrix H and overlap matrix S in the generalised eigenvalue equation HC = eSC. The H matrix is the matrix representation of a hermitian operator, typically the Hamiltonian, in the subspace spanned by the excitation operators, and S is the overlap matrix of the subspace generating states. The method aims to obtain a description of low-lying excited states described as an expansion of excitation operators acting on the effective ground-state obtained from a variational calculation.

Based on *arXiv:1603.05681 <https://arxiv.org/abs/1603.05681>*.

Parameters

- **state** (*GeneralAnsatz*) – Ansatz used to represent the ground state, it is used as a reference state for the excitations to generate the subspace
- **hermitian_operator** (*QubitOperator*) – A hermitian operator, typically a hamiltonian, to be expanded in the subspace.
- **expansion_operators** (List[*QubitOperator*]) – A List of excitation operators spanning the subspace.

ALLOWED_PROTOCOLS = (<class 'inquanto.protocols._protocol_standard.ProtocolDirect'>, <class 'inquanto.protocols._protocol_standard.ProtocolIndirect'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparse'>, <class 'inquanto.protocols._protocol.ProtocolStateVectorBackendSupport'>)

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) –

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns
True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns
Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns
List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns
float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

evaluate (*parameters=None*)

Evaluate the result of a calculation.

Once the experiment has been run, this function evaluates the computable expression and returns its numerical values.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional symbols to value mapping, if the expression has symbolic part

Returns
Value of the computables

generate_circuits (*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf ()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch (*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.

- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves (*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

print_tree ()

Print the computable dependence tree.

Return type
None

rebuild (*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

retrieve_distributions (*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns
List[List[ResultHandle]] – Results distributions.

run (*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.

- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Computable – self.

symbolic_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker(*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

class ComputableRDM1234Real(*space*, *ansatz*, *encoding*, *symmetry_operators*, *cas_elec*, *cas_orbs*, *cu4=True*, *taperer=None*)

Bases: ComputableSuper

Measures 1,2,3-RDMs for a given state, defined by ansatz and parameters.

Estimates 4-RDM via the cumulant expansion or measures it.

Parameters

- **space** (*FermionSpace*) – Fermion space where the operators are defined.
- **ansatz** (*GeneralAnsatz*) – Ansatz with respect to which the expectation values are computed.
- **encoding** (*QubitMapping*) – Qubit encoding from fermion space to qubit space.
- **symmetry_operators** (List[*SymmetryOperatorPauli*]) – Z2 symmetries of the Hamiltonian.
- **cas_elec** (int) – Number of active electrons.
- **cas_orbs** (int) – Number of active orbitals.
- **cu4** (bool, default: True) – If True (default), the 4-RDM is estimated via the cumulant expansion approximation, otherwise it is measured.

- **taperer** (Optional[*TapererZ2*], default: None) – TapererZ2 initialized with the Hamiltonian, or None if tapering is not used.

Returns

1,2,3,4-PDMs (or RDMs) as a list of arrays, in PySCF's ordering (<p^r^sq>)

ALLOWED_PROTOCOLS = (<class 'inquanto.protocols._protocol_standard.ProtocolDirect'>, <class 'inquanto.protocols._protocol_standard.ProtocolIndirect'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>, <class 'inquanto.protocols._protocol.ProtocolStateVectorBackendSupport'>)

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) –

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns

True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns

Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns

List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns

float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns

Value of the measurement.

evaluate(*parameters=None*)

Evaluate the result of a calculation.

Once the experiment has been run, this function evaluates the computable expression and returns its numerical values.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional symbols to value mapping, if the expression has symbolic part

Returns

Value of the computables

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.

- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves(*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

print_tree()

Print the computable dependence tree.

Return type
None

rebuild(*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

retrieve_distributions (*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns
List[List[ResultHandle]] – Results distributions.

run (*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Computable – self.

symbolic_evaluate (*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker(*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

class ComputableRestrictedOneBodyRDM(*fermion_space*, *ansatz*, *encoding*)

Bases: ComputableCollinearOneBodyRDM

Computable for RestrictedOneBodyRDM matrices.

Parameters

- **fermion_space** (*FermionSpace*) –
- **ansatz** (*GeneralAnsatz*) –
- **encoding** (*QubitMapping*) –

ALLOWED_PROTOCOLS = (<class 'inquanto.protocols._protocol_standard.ProtocolDirect'>, <class 'inquanto.protocols._protocol_standard.ProtocolIndirect'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparse'>, <class 'inquanto.protocols._protocol.ProtocolStateVectorBackendSupport'>)

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) –

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns
True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns
Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns
List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns
float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

evaluate(*parameters=None*)

Evaluate the result of a calculation.

Once the experiment has been run, this function evaluates the computable expression and returns its numerical values.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional symbols to value mapping, if the expression has symbolic part

Returns
RestrictedOneBodyRDM – Value of the computables

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves(*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

print_tree()

Print the computable dependence tree.

Return type
None

rebuild(*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

retrieve_distributions(*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns
List[List[ResultHandle]] – Results distributions.

run(*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Computable – self.

symbolic_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker(*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

class ComputableRestrictedOneBodyRDMReal(*fermion_space*, *ansatz*, *encoding*)

Bases: ComputableCollinearOneBodyRDM

Computable for real RestrictedOneBodyRDM matrices.

Parameters
- **fermion_space** (*FermionSpace*) –
- **ansatz** (*GeneralAnsatz*) –
- **encoding** (*QubitMapping*) –

ALLOWED_PROTOCOLS = (<class 'inquanto.protocols._protocol_standard.ProtocolDirect'>, <class 'inquanto.protocols._protocol_standard.ProtocolIndirect'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparse'>, <class 'inquanto.protocols._protocol.ProtocolStateVectorBackendSupport'>)

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) –

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters
- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.

- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns
True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns
Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns
List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns
float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

evaluate(*parameters=None*)

Evaluate the result of a calculation.

Once the experiment has been run, this function evaluates the computable expression and returns its numerical values.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional symbols to value mapping, if the expression has symbolic part

Returns
RestrictedOneBodyRDM – Value of the computables

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles

- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves (*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

print_tree ()

Print the computable dependence tree.

Return type
None

rebuild (*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

retrieve_distributions (*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns
List[List[ResultHandle]] – Results distributions.

run (*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns

Computable – self.

symbolic_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns

Value of the measurement.

walker(*depth=0*)

Walks over all computables including self.

Parameters

depth – Depth counter.

Yields

Depth, Computable.

Return type

Iterator[Tuple[int, Computable]]

class ComputableSpinlessNBodyPDMTensorReal(*n*, *fermion_space*, *ansatz*, *encoding*, *symmetry_operators*, *taperer=None*)

Bases: ComputableSuper

Computable for spinless, n-body PDM matrices.

Parameters

- **n** (int) – n-body PDM.
- **fermion_space** (*FermionSpace*) – Fermion space where the operators are defined.
- **ansatz** (*GeneralAnsatz*) – Ansatz with respect to which the expectation values are computed.
- **encoding** (*QubitMapping*) – Qubit encoding from fermion space to qubit space.
- **symmetry_operators** (List[*SymmetryOperatorPauli*]) – List of Z2 symmetries of the Hamiltonian.
- **taperer** (Optional[*TapererZ2*], default: None) – Optional taperer object.

ALLOWED_PROTOCOLS = (<class 'inquanto.protocols._protocol_standard.ProtocolDirect'>, <class 'inquanto.protocols._protocol_standard.ProtocolIndirect'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>, <class 'inquanto.protocols._protocol.ProtocolStateVectorBackendSupport'>)

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) –

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns

True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns

Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns

List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns
float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate (*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

evaluate (*parameters=None*)

Evaluate the result of a calculation.

Once the experiment has been run, this function evaluates the computable expression and returns its numerical values.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional symbols to value mapping, if the expression has symbolic part

Returns
Value of the computables

generate_circuits (*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves(*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

print_tree()

Print the computable dependence tree.

Return type
None

rebuild(*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

retrieve_distributions(*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns
List[List[ResultHandle]] – Results distributions.

run(*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Computable – self.

symbolic_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker(*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

class ComputableSpinlessNBodyRDMTensor(*n*, *fermion_space*, *ansatz*, *encoding*)

Bases: ComputableSuper

Computable for spinless n-body RDM matrices.

Parameters

- **n** (int) – n-body RDM.
- **fermion_space** (*FermionSpace*) – Fermion space where the operators are defined.
- **ansatz** (*GeneralAnsatz*) – The ansatz with respect to the expectation values which are computed.
- **encoding** (*QubitMapping*) – Qubit encoding from fermion space to qubit space.

ALLOWED_PROTOCOLS = (<class 'inquanto.protocols._protocol_standard.ProtocolDirect'>, <class 'inquanto.protocols._protocol_standard.ProtocolIndirect'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparse'>, <class 'inquanto.protocols._protocol.ProtocolStateVectorBackendSupport'>)

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) –

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns
True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns
Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns
List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns
float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

evaluate(*parameters=None*)

Evaluate the result of a calculation.

Once the experiment has been run, this function evaluates the computable expression and returns its numerical values.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional symbols to value mapping, if the expression has symbolic part

Returns
Value of the computables

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns

Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type

bool

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns

Handles of measurement results.

leaves(*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields

All contained computables of requested type, satisfying condition.

print_tree()

Print the computable dependence tree.

Return type
None

rebuild(*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

retrieve_distributions(*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns
List[List[ResultHandle]] – Results distributions.

run(*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Computable – self.

symbolic_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns

Value of the measurement.

walker(*depth=0*)

Walks over all computables including self.

Parameters

depth – Depth counter.

Yields

Depth, Computable.

Return type

Iterator[Tuple[int, Computable]]

class ComputableSpinlessNBodyRDMTensorReal(*n*, *fermion_space*, *ansatz*, *encoding*, *symmetry_operators*, *taperer=None*, *ordering='p^r^sq'*)

Bases: ComputableSuper

Computable for spinless, n-body RDM matrices.

Calculates a general n-body RDM $\Gamma^{i\ldots j}_{n\ldots m} = \langle\Psi_0|\, \hat{E}^{i\ldots j}_{n\ldots m}\, |\Psi_0\rangle$ where $\hat{E}^{i\ldots j}_{n\ldots m}$ is a spin-traced excitation operator. For example, in the one-body case this is:

$\hat{E}^p_q = \hat{a}^\dagger_{p\uparrow}\hat{a}_{q\uparrow} + \hat{a}^\dagger_{p\downarrow}\hat{a}_{q\downarrow}$.

And in the two-body case:

$\hat{E}^{pr}_{qs} = \hat{a}^\dagger_{p\uparrow}\hat{a}^\dagger_{r\uparrow}\hat{a}_{s\uparrow}\hat{a}_{q\uparrow} + \hat{a}^\dagger_{p\uparrow}\hat{a}^\dagger_{r\downarrow}\hat{a}_{s\downarrow}\hat{a}_{q\uparrow} + \hat{a}^\dagger_{p\downarrow}\hat{a}^\dagger_{r\uparrow}\hat{a}_{s\uparrow}\hat{a}_{q\downarrow} + \hat{a}^\dagger_{p\downarrow}\hat{a}^\dagger_{r\downarrow}\hat{a}_{s\downarrow}\hat{a}_{q\downarrow}$.

Parameters

- **n** (int) – n-body RDM.
- **fermion_space** (*FermionSpace*) – Fermion space where the operators are defined.
- **ansatz** (*GeneralAnsatz*) – Ansatz with respect to which the expectation values are computed.
- **encoding** (*QubitMapping*) – Qubit encoding from fermion space to qubit space.
- **symmetry_operators** (List[*SymmetryOperatorPauli*]) – List of Z2 symmetries of the Hamiltonian.
- **taperer** (Optional[*TapererZ2*], default: None) – Optional taperer object.
- **ordering** (str, default: "p^r^sq") – RDM index convention. The default corresponds to PySCF.

ALLOWED_PROTOCOLS = (<class 'inquanto.protocols._protocol_standard.ProtocolDirect'>, <class 'inquanto.protocols._protocol_standard.ProtocolIndirect'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>, <class 'inquanto.protocols._protocol.ProtocolStateVectorBackendSupport'>)

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) –

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns

True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns

Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns

List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.

- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns
float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

evaluate(*parameters=None*)

Evaluate the result of a calculation.

Once the experiment has been run, this function evaluates the computable expression and returns its numerical values.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional symbols to value mapping, if the expression has symbolic part

Returns
Value of the computables

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters
- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch (*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns

Handles of measurement results.

leaves (*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields

All contained computables of requested type, satisfying condition.

print_tree ()

Print the computable dependence tree.

Return type

None

rebuild (*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

retrieve_distributions(*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns

List[List[ResultHandle]] – Results distributions.

run(*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns

Computable – self.

symbolic_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns

Value of the measurement.

walker(*depth=0*)

Walks over all computables including self.

Parameters

depth – Depth counter.

Yields

Depth, Computable.

Return type

Iterator[Tuple[int, Computable]]

class ComputableUnrestrictedOneBodyRDM(*fermion_space*, *ansatz*, *encoding*)

Bases: ComputableCollinearOneBodyRDM

Computable for UnrestrictedOneBodyRDM matrices.

Parameters

- **fermion_space** (*FermionSpace*) –
- **ansatz** (*GeneralAnsatz*) –
- **encoding** (*QubitMapping*) –

ALLOWED_PROTOCOLS = (<class 'inquanto.protocols._protocol_standard.ProtocolDirect'>, <class 'inquanto.protocols._protocol_standard.ProtocolIndirect'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparse'>, <class 'inquanto.protocols._protocol.ProtocolStateVectorBackendSupport'>)

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) –

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns

True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns

Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns

List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns

float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns

Value of the measurement.

evaluate(*parameters=None*)

Evaluate the result of a calculation.

Once the experiment has been run, this function evaluates the computable expression and returns its numerical values.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional symbols to value mapping, if the expression has symbolic part

Returns

UnrestrictedOneBodyRDM – Value of the computables

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves(*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

print_tree()

Print the computable dependence tree.

Return type
None

rebuild(*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

retrieve_distributions(*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns
List[List[ResultHandle]] – Results distributions.

run(*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Computable – self.

symbolic_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker (*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

class ComputableUnrestrictedOneBodyRDMReal (*fermion_space*, *ansatz*, *encoding*)

Bases: ComputableCollinearOneBodyRDM

Computable for real UnrestrictedOneBodyRDM matrices.

Parameters
- **fermion_space** (*FermionSpace*) –
- **ansatz** (*GeneralAnsatz*) –
- **encoding** (*QubitMapping*) –

ALLOWED_PROTOCOLS = (<class 'inquanto.protocols._protocol_standard.ProtocolDirect'>, <class 'inquanto.protocols._protocol_standard.ProtocolIndirect'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparse'>, <class 'inquanto.protocols._protocol.ProtocolStateVectorBackendSupport'>)

Protocols this computable are compatible with.

approximate_error (*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) –

build (*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters
- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns

True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns

Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns

List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns

float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns

Value of the measurement.

evaluate(*parameters=None*)

Evaluate the result of a calculation.

Once the experiment has been run, this function evaluates the computable expression and returns its numerical values.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional symbols to value mapping, if the expression has symbolic part

Returns

UnrestrictedOneBodyRDM – Value of the computables

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns

Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type

bool

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves (*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

print_tree ()

Print the computable dependence tree.

Return type
None

rebuild (*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

retrieve_distributions (*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns
List[List[ResultHandle]] – Results distributions.

run (*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.

- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Computable – self.

symbolic_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker(*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

class Computables(**computables*)

Bases: Computable

Combines multiple computables into one as a tuple, for one unified run and evaluation.

Parameters
***computables** (Computable) – Other computables to be combined.

ALLOWED_PROTOCOLS = ()

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Estimates the standard error of each element after the measurements have been run.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional parameters to substitute post-evaluation.

Note: Not all protocols and computables can compute valid standard errors.

Returns
Tuple – A tuple of standard errors.

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns

True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns

Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns

List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.

- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns

float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

cost_estimate_elementwise(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the Computables on a device/simulator.

The costs returned are the costs per element if they are individual experiments without redundancies removed. Syntax checker will usually be automatically selected but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instatiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns

float – The cost in Quantinuum credits for evaluating the computables given the input arguments in the shape of the Computables object.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns

Value of the measurement.

evaluate(*parameters=None*)

Evaluates the computable and returns its actual value after the measurements have been run.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional parameters to substitute post-evaluation.

Returns

Tuple – A tuple of the values of the computables it contains.

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.

- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves(*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

print_tree()

Print the computable dependence tree.

Return type
None

rebuild(*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

retrieve_distributions(*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns

List[List[ResultHandle]] – Results distributions.

run(*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns

Computable – self.

symbolic_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker (*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

class ExpectationValue (*state*, *kernel*)

Bases: ComputableExpression

Computes the expectation value of a kernel operator with a state: $< \Psi|O|\Psi >$.

The computable is supported by the following protocols:

- ProtocolDirect
- ProtocolIndirect
- ProtocolStateVectorSparseLegacy
- ProtocolStateVectorSparse
- ProtocolStateVectorBackendSupport
- ProtocolSymbolic

Parameters

- **state** (*GeneralAnsatz*) – State Psi for the expectation value.
- **kernel** (*QubitOperator*) – Operator O for the expectation value.

ALLOWED_PROTOCOLS = (<class 'inquanto.protocols._protocol_standard.ProtocolDirect'>, <class 'inquanto.protocols._protocol_standard.ProtocolIndirect'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparse'>, <class 'inquanto.protocols._protocol.ProtocolStateVectorBackendSupport'>, <class 'inquanto.protocols._protocol_symbolic.ProtocolSymbolic'>)

Protocols this computable are compatible with.

approximate_error (*parameters=None*)

Estimates the standard error of the evaluated value after the measurements have been run.

It calculates the approximate error on the expectation value evaluation according to the formulae given in the appendix of this paper https://arxiv.org/abs/1704.05018.

Note: It only supports the error of the real part.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional parameters to substitute post-evaluation.

Returns

Union[complex, Expr] –:

A standard error calculated using the covariance of the hamiltonian,
as a complex floating-point value or symbolic expression.

bra_derivative_imag (*factor=1.0*)

Returns a computable for the imaginary part of partial bra derivatives of this expectation value.

$f\Im < \partial_{\theta_i}\Psi(\theta)|O|\Psi > (\theta)$ for all i

Note: Currently supports Hermitian operators only.

Parameters

factor (float, default: 1.0) – A factor f for the result, the default value is 1.

Returns

ExpectationValueBraDerivativeImag – Computable for the imaginary part of partial bra derivative.

bra_derivative_real (*factor=1.0*)

Returns a computable for the real part of partial bra derivatives of this expectation value.

$f\Re < \partial_{\theta_i}\Psi(\theta)|O|\Psi > (\theta)$ for all i

Note: Currently supports Hermitian operators only.

Parameters

factor (float, default: 1.0) – A factor f for the result, the default value is 1.

Returns

ExpectationValueBraDerivativeReal – Computable for the real part of partial bra derivative.

build (*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

static check_status_ready (*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.

- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns
True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns
Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns
List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns
float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

evaluate(*parameters=None*)

Evaluate the result of a calculation.

Once the experiment has been run, this function evaluates the computable expression and returns its numerical values.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional symbols to value mapping, if the expression has symbolic part

Returns
Union[complex, Expr] – Value of the computables

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

gradient_real()

Returns a computable for the real part of the derivatives of this expectation value.

$\Re \frac{d<\Psi(\theta)|O|\Psi(\theta)>}{d\theta}$ for all θ

Note: Currently supports Hermitian operators only.

Returns
ExpectationValueDerivativeReal – Computable for the real part of derivates of the expectation value.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles

- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves (*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

metric_tensor_expression_real ()

Returns a computable metric tensor corresponding to this expectation value and its parameters.

$\frac{1}{4}\Re < \partial_{\theta_i}\Psi(\theta)|\partial_{\theta_j}\Psi > (\theta)$ for all i, j

Returns
ComputableMetricTensorReal – Computable metric tensor.

classmethod multiple_broadcast (*state*, **kernels*)

Broadcast numpy arrays to expectation value computables.

Parameters

- **state** (*GeneralAnsatz*) – State in expectation value.
- **kernels*** – Arrays of floats or ints representing the operator kernels.

Returns

Computables –:

Collection of ComputableArrays, each storing computable ExpectationValue objects corresponding to the
input arrays.

classmethod ndarray_broadcast (*state*, *kernel_array*)

Broadcast a numpy array to an expectation value computable array.

Parameters

- **state** (*GeneralAnsatz*) – State in expectation value.
- **kernel_array** (ndarray) – Array of floats or ints representing the operator kernel.

Returns
ComputableArray – ComputableArray of computable ExpectationValue objects corresponding to input array.

print_tree ()

Print the computable dependence tree.

Return type
None

rebuild(*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

retrieve_distributions(*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns

List[List[ResultHandle]] – Results distributions.

run(*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns

Computable – self.

symbolic_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker (*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

class ExpectationValueBraDerivativeImag (*state*, *kernel*, *symbols*, *factor=1.0*)

Bases: ComputableSuper

Computes the imaginary part of the partial derivatives of an expectation value of an operator.

:math:` f Im <partial_{theta} Psi(theta) | O | Psi>(theta)`

The computable is supported by the following protocols:

- ProtocolHadamardDirectPauliZ
- ProtocolHadamardIndirectPauliZ
- ProtocolMidMeasurementGradient
- ProtocolStateVectorSparseLegacy
- ProtocolSymbolic

Note: Currently supports Hermitian operators only.

Parameters

- **state** (*GeneralAnsatz*) – State Psi for the expectation value.
- **kernel** (*QubitOperator*) – Operator O for the expectation value.
- **symbols** (Iterable[Symbol]) – List of symbols with respect to the derivatives are computed.
- **factor** (float, default: 1.0) – A simple factor f, by default it is 1.

ALLOWED_PROTOCOLS = [<class
'inquanto.protocols._protocol_derivative.ProtocolHadamardDirectPauliZ'>,
<class
'inquanto.protocols._protocol_derivative.ProtocolHadamardIndirectPauliZ'>,
<class
'inquanto.protocols._protocol_derivative.ProtocolMidMeasurementGradient'>,
<class
'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>,
<class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparse'>,
<class 'inquanto.protocols._protocol_symbolic.ProtocolSymbolic'>]

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) –

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns

True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns

Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns

List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.

- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns
float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

evaluate(*parameters=None*)

Evaluates the computable and returns its actual value after the measurements have been run.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional parameters to substitute post-evaluation.

Returns
ndarray – An array of floating-point values or symbolic expressions.

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves(*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

print_tree()

Print the computable dependence tree.

Return type
None

rebuild(*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

retrieve_distributions(*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns
List[List[ResultHandle]] – Results distributions.

run (*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Computable – self.

symbolic_evaluate (*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker (*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

class ExpectationValueBraDerivativeReal (*state*, *kernel*, *symbols*, *factor=1.0*)

Bases: ComputableSuper

Computes the real part of the partial derivatives of an expectation value of an operator.

:math:` f Re <partial_{theta} Psi(theta) | O | Psi>(theta)`

The computable is supported by the following protocols:

- ProtocolHadamardIndirectPauliY
- ProtocolHadamardDirectPauliY
- ProtocolPhaseShift
- ProtocolStateVectorSparseLegacy
- ProtocolSymbolic

Note: Currently only supports Hermitian operators.

Parameters

- **state** (*GeneralAnsatz*) – State Psi for the expectation value.
- **kernel** (*QubitOperator*) – Operator O for the expectation value.
- **symbols** (Iterable[Symbol]) – List of symbols with respect to the derivatives are computed.
- **factor** (float, default: 1.0) – A simple factor f, by default it is 1.

ALLOWED_PROTOCOLS = [<class 'inquanto.protocols._protocol_derivative.ProtocolHadamardIndirectPauliY'>, <class 'inquanto.protocols._protocol_derivative.ProtocolHadamardDirectPauliY'>, <class 'inquanto.protocols._protocol_derivative.ProtocolPhaseShift'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparse'>, <class 'inquanto.protocols._protocol_symbolic.ProtocolSymbolic'>]

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) –

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns
True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns
Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns
List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns
float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

evaluate(*parameters=None*)

Evaluates the computable and returns its actual value after the measurements have been run.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional parameters to substitute post-evaluation.

Returns
ndarray – An array of floating-point values or symbolic expressions.

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves(*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

print_tree()

Print the computable dependence tree.

Return type
None

rebuild(*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

retrieve_distributions(*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns
List[List[ResultHandle]] – Results distributions.

run(*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.

- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Computable – self.

symbolic_evaluate (*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker (*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

class ExpectationValueDerivativeReal (*state*, *kernel*, *symbols*)

Bases: ComputableSuper

Computes the real part of the total derivatives (gradient) of an expectation value of an operator.

:math:` Re frac{d<Psi(theta) | O | Psi(theta)>}{dtheta}`

The computable is supported by the following protocols:

- ProtocolHadamardIndirectPauliY
- ProtocolHadamardDirectPauliY
- ProtocolPhaseShift
- ProtocolStateVectorSparseLegacy
- ProtocolSymbolic

Note: Currently only supports Hermitian operators.

Parameters

- **state** (*GeneralAnsatz*) – State Psi for the expectation value.
- **kernel** (*QubitOperator*) – Operator O for the expectation value.
- **symbols** (Iterable[Symbol]) – List of symbols with respect to the derivatives are computed.

ALLOWED_PROTOCOLS = ()

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) –

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns

True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns

Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns

List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.

- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns
float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

evaluate(*parameters=None*)

Evaluate the result of a calculation.

Once the experiment has been run, this function evaluates the computable expression and returns its numerical values.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional symbols to value mapping, if the expression has symbolic part

Returns
Value of the computables

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters
- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch (*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns

Handles of measurement results.

leaves (*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields

All contained computables of requested type, satisfying condition.

print_tree ()

Print the computable dependence tree.

Return type

None

rebuild (*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

retrieve_distributions(*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns

List[List[ResultHandle]] – Results distributions.

run(*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns

Computable – self.

symbolic_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns

Value of the measurement.

walker(*depth=0*)

Walks over all computables including self.

Parameters

depth – Depth counter.

Yields

Depth, Computable.

Return type

Iterator[Tuple[int, Computable]]

class Overlap(*bra_state*, *ket_state*, *kernel=None*)

Bases: ComputableExpression

Computes the overlap of states with a kernel operator.

$< \Psi_1 | O | \Psi_2 >$

The computable is supported by the following protocols:

- ProtocolStateVectorSparseLegacy
- ProtocolSymbolic

Note: Currently only supports Hermitian operators.

Parameters

- **bra_state** (*GeneralAnsatz*) – State Psi_1 for the overlap.
- **ket_state** (*GeneralAnsatz*) – State Psi_2 for the overlap.
- **kernel** (Optional[*QubitOperator*], default: None) – Operator O for the overlap.

ALLOWED_PROTOCOLS = (<class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparse'>, <class 'inquanto.protocols._protocol_symbolic.ProtocolSymbolic'>)

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) –

Return type

Any

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.

- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns
True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns
Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns
List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns
float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

evaluate(*parameters=None*)

Evaluates the computable and returns its actual value after the measurements have been run.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional parameters to substitute post-evaluation.

Returns
Union[complex, Expr] – A complex floating-point value or symbolic expressions.

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves(*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

classmethod multiple_broadcast(*bra_state*, *ket_state*, **kernels*)

Broadcast numpy arrays to overlap computables.

Parameters

- **bra_state** (*GeneralAnsatz*) – Bra in overlap.
- **ket_state** (*GeneralAnsatz*) – Ket in overlap.
- **kernels*** – Arrays of floats or ints representing the overlap kernels.

Returns
Computables – Collection of ComputableArrays, each storing computable Overlap objects corresponding to the input arrays.

classmethod ndarray_broadcast(*bra_state*, *ket_state*, *kernel_array*)

Broadcast a numpy array to an overlap computable array.

Parameters

- **bra_state** (*GeneralAnsatz*) – Bra in overlap.
- **ket_state** (*GeneralAnsatz*) – Ket in overlap.
- **kernel_array** (ndarray) – Array of floats or ints representing the overlap kernel.

Returns
ComputableArray of computable Overlap objects corresponding to input array.

print_tree()

Print the computable dependence tree.

Return type
None

rebuild(*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

retrieve_distributions(*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.

- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns
List[List[ResultHandle]] – Results distributions.

run (*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Computable – self.

symbolic_evaluate (*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker (*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

class OverlapSquared (*bra_state*, *ket_state*, *kernel=None*)

Bases: ComputableExpression

Computes the overlap squared of states with a kernel operator.

$| < \Phi | P | \Psi(\theta) > |^2$

Note: The kernel operator must be a single Pauli string.

Parameters

- **bra_state** (*GeneralAnsatz*) – Bra part of the expression.
- **ket_state** (*GeneralAnsatz*) – Ket part of the expression.
- **kernel** (Optional[*QubitOperatorString*], default: None) – Pauli operator over qubits.

ALLOWED_PROTOCOLS = (<class 'inquanto.protocols._protocol_overlap.ProtocolVacuum'>, <class 'inquanto.protocols._protocol_overlap.ProtocolDSP'>, <class 'inquanto.protocols._protocol_overlap.ProtocolCSP'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparse'>, <class 'inquanto.protocols._protocol_symbolic.ProtocolSymbolic'>)

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) –

Return type

Any

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns

True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns

Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns

List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns

float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns

Value of the measurement.

evaluate(*parameters=None*)

Evaluate the result of a calculation.

Once the experiment has been run, this function evaluates the computable expression and returns its numerical values.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional symbols to value mapping, if the expression has symbolic part

Returns

complex – Value of the computables

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type
bool

ket_derivative()

Computes the partial derivatives of the expectation value with a Hermitian operator.

$\frac{d}{d\theta}| < \Phi|P|\Psi(\theta) > |^2$

Returns
OverlapSquaredKetDerivative – Ket derivate computable.

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves(*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

classmethod multiple_broadcast(*bra_state*, *ket_state*, **kernels*)

Broadcast numpy arrays to overlap squared computables arrays.

Parameters

- **bra_state** (*GeneralAnsatz*) – Bra in overlap.
- **ket_state** (*GeneralAnsatz*) – Ket in overlap.
- **kernels*** – Arrays of floats or ints representing the overlap kernels.

Returns

Collection of ComputableArrays, each storing computable OverlapSquared objects corresponding to
input array.

classmethod ndarray_broadcast(*bra_state*, *ket_state*, *kernel_array*)

Broadcast a numpy array to an overlap squared computable array.

Parameters

- **bra_state** (*GeneralAnsatz*) – Bra in overlap.
- **ket_state** (*GeneralAnsatz*) – Ket in overlap.
- **kernel_array** (ndarray) – Array of floats or ints representing the overlap kernel.

Returns
ComputableArray of OverlapSquared objects corresponding to input array.

print_tree()

Print the computable dependence tree.

Return type
None

rebuild(*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

retrieve_distributions(*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.

- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns
List[List[ResultHandle]] – Results distributions.

run (*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Computable – self.

symbolic_evaluate (*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker (*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

class OverlapSquaredKetDerivative (*principle_state*, *other_state*, *symbols*, *kernel=None*)

Bases: _Gradient

Computes the partial derivatives: $\frac{d}{d\theta}| < \Phi|P|\Psi(\theta) > |^2$.

Parameters

- **principle_state** (*GeneralAnsatz*) – Symbolic state with variational parameters of interest.

- **other_state** (*GeneralAnsatz*) – Symbolic or numerical state. Will not be differentiated.
- **symbols** (Iterable[Symbol]) – Symbol to differentiate with respect to. These symbols must exist in principle_state.
- **kernel** (Optional[*QubitOperator*], default: None) – Problem kernel. Should only contain one term.

ALLOWED_PROTOCOLS = [<class 'inquanto.protocols._protocol_derivative.ProtocolVacuumPhaseShift'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparseLegacy'>, <class 'inquanto.protocols._protocol_sparse.ProtocolStateVectorSparse'>]

Protocols this computable are compatible with.

approximate_error(*parameters=None*)

Compute the approximate error arising from the statistical nature of the computable's evaluation.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) –

build(*protocols*, *optimize=False*, *unfold_depth=10*)

Apply protocol for all children recursively.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns

Computable – self.

static check_status_ready(*backend*, *all_handles_or_name*)

Checks if any circuits are still being measured.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the launch or the filename of json file.

Returns

True if ready to retrieve results, else false.

clear()

Clear all the protocols recursively.

Returns

Computable – self.

collect_protocols()

Collect all the protocols to be measured.

Returns

List[Protocol] – List of protocols to be measured.

cost_estimate(*backend*, *parameters*, *n_shots=8192*, *compiler_passes=None*, *syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost in quantinuum credits for evaluating the computable on a device/simulator.

Syntax checker will usually be automatically selected, but in some cases needs to be provided.

Parameters

- **backend** (Backend) – Must be an instance of QuantinuumBackend, instantiated with an appropriate device name.
- **parameters** (Optional[*SymbolDict*]) – The parameters of the circuit to be evaluated.
- **n_shots** (int, default: 8192) – The number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – A pytket compilation regime. If None, the default compilation pass is used.
- **syntax_checker** (Optional[str], default: None) – Which syntax checker to use. The default is None.
- **use_websocket** (Optional[bool], default: None) – Whether to use a web connection.

Returns

float – The cost in Quantinuum credits for evaluation of the computable given the input arguments.

default_evaluate(*parameters=None*)

Attempts to evaluate the results immediately for return with default protocols and backends.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns

Value of the measurement.

evaluate(*parameters=None*)

Evaluate the result of a calculation.

Once the experiment has been run, this function evaluates the computable expression and returns its numerical values.

Parameters

parameters (Union[*SymbolDict*, Dict, None], default: None) – Optional symbols to value mapping, if the expression has symbolic part

Returns

ndarray – Value of the computables

generate_circuits(*backend*, *parameters*, *compiler_passes=None*, *custom_prefix=None*)

Generate a sequence of all the measurement circuits. It is recommended to use this only for inspection.

Parameters

- **backend** (Backend) – The backend used for running circuits.
- **parameters** (Union[*SymbolDict*, Dict]) – Set of circuit parameters.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.

- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Iterator – Sequence of all the measurement circuits.

is_leaf()

Is this computable a leaf in a computable expression tree.

Return type
bool

launch(*backend*, *parameters=None*, *n_shots=8192*, *seed=None*, *name=None*, *compiler_passes=None*, *custom_prefix=None*)

Launches hardware measurements and returns handles to them.

If name is set then the handles will be saved into a name.json file.

Parameters

- **backend** (Backend) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **name** (Optional[str], default: None) – filename to store the handles
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User-defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Handles of measurement results.

leaves(*measuarable_type: ClassVar = None*, *condition=None*) → Iterator[Computable]

Loops over leaves with type and condition, if None then loops over all leaves.

Parameters

- **measuarable_type** – Computable types yielded.
- **condition** – Condition that must be satisfied to yield computable.

Yields
All contained computables of requested type, satisfying condition.

print_tree()

Print the computable dependence tree.

Return type
None

rebuild(*protocols*, *optimize=False*, *unfold_depth=10*)

Deletes all existing protocols and applies protocol to all children recursively again.

Parameters

- **protocols** (Union[Protocol, List[Protocol]]) – Protocol to be used for calculating computable.
- **optimize** – Whether to optimize by removing redundant measurements.
- **unfold_depth** (int, default: 10) – Depth to which the computable expression tree will be unfolded.

Returns
Computable – self.

retrieve_distributions (*backend*, *all_handles_or_name*)

Retrieves distributions for computable evaluation.

Given the handles, or the JSON file containing the handles produced by the launch method, this method retrieves the distributions and then the computable can be evaluated.

Parameters

- **backend** – pytket backend.
- **all_handles_or_name** (Union[str, List]) – Returned data of the self.launch(…) or the filename of json file.

Returns
List[List[ResultHandle]] – Results distributions.

run (*backend=None*, *parameters=None*, *n_shots=8192*, *seed=None*, *compiler_passes=None*, *custom_prefix=None*)

Runs the experiments, and holds the results in a form of distributions.

Parameters

- **backend** (Optional[Backend], default: None) – A pytket backend.
- **parameters** (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values
- **n_shots** (int, default: 8192) – Number of shots used for calculation.
- **seed** (Optional[int], default: None) – RNG seed for backend.
- **compiler_passes** (Optional[BasePass], default: None) – Backend compiler passes.
- **custom_prefix** (Optional[str], default: None) – User defined prefix to use to name the circuits generated by computables. Default value is None, which corresponds to a prefix defined by the choice of computable.

Returns
Computable – self.

symbolic_evaluate (*parameters=None*)

Attempts to evaluate the results immediately for return with symbolic protocols.

Parameters
parameters (Union[*SymbolDict*, Dict, None], default: None) – SymbolDict or dict to map symbols to values.

Returns
Value of the measurement.

walker(*depth=0*)

Walks over all computables including self.

Parameters
depth – Depth counter.

Yields
Depth, Computable.

Return type
Iterator[Tuple[int, Computable]]

22.4 inquanto.core

22.4.1 Parameters Classes

class SymbolDict(*initializer=None*, ***kwargs*)

Bases: SymbolTypeKeyDictWrapper

The SymbolDict class holds a Symbol -> Value ordered map.

By means of member functions, provides a general way to keep track of the symbols in a quantum expression. When a string is used as a key, it is converted to a Symbol object.

Parameters

- **initializer** (Union[*SymbolDict*, Dict[Union[str, Symbol], Union[float, complex, Expr, None]], Iterable, None], default: None) – A dict, SymbolDict or an iterable, representing the input map.
- **kwargs** (Union[float, complex, Expr, None]) – Key-value pairs that are used to generate the map if *initialiser* is None.

Examples

```
>>> SymbolDict(a=1,b=2)
SymbolDict({a: 1, b: 2})
```

---

clear()

Remove all symbols.

Return type
None

copy()

Performs a deep copy of self.

Returns
Union[SymbolTypeKeyDictWrapper, *SymbolDict*, *SymbolSet*] – A deep copy of this object.

Examples

```
>>> symbols = SymbolSet(["a", "b", "c"])
>>> other = symbols.copy()
>>> symbols == other
True
>>> symbols is other
False
>>> symbols.add("d")
d
>>> symbols == other
False
```

---

df()

Returns a Pandas dataframe containing information about the symbol map.

discard(**symbols*)

Discard symbols.

Parameters
symbols (Union[str, Symbol]) – Symbols to be discarded.

Returns
SymbolTypeKeyDictWrapper – Updated SymbolDict object.

from_array(*array*)

Sets symbol values from an input array.

The array must have at least as many elements as there are symbols in the dict.

Parameters
array (Union[ndarray, List]) – Array of numeric values.

Returns
SymbolDict – Updated instance of SymbolDict.

classmethod from_circuit(*circuit*)

Instantiate from a circuit.

Parameters
circuit (Circuit) – Circuit object.

Returns
Union[SymbolTypeKeyDictWrapper, *SymbolDict*, *SymbolSet*] – SymbolTypeKeyDictWrapper, SymbolDict or SymbolSet object.

generate_report()

Generates a report on the symbols order and values in json format.

items()

Returns the iterator to the underlying dict key-value pairs.

Return type
ItemsView[Symbol, Union[float, complex, Expr, None]]

keys()

Returns the iterator to the underlying dict keys.

Return type
KeysView[Symbol]

make_hashable()

Alias for str(self).

print_report()

Prints information about the symbol map.

set(*symbol*, *value=None*)

Adds or updates a symbol value.

Parameters

- **symbol** (Union[str, Symbol]) – A symbol to add/update.
- **value** (Union[float, complex, Expr, None], default: None) – Input symbol value.

Returns

None – None.

property symbols: List[Symbol]

Returns a list of all symbols.

Return type

List[Symbol]

to_array()

Converts underlying dict values to an array.

Return type

ndarray

to_dict()

Returns an underlying dict.

Return type

Dict[Symbol, Union[float, complex, Expr, None]]

update(*other*)

Update the symbol values.

The order is defined by the symbols first appearance.

Parameters

other (Union[*SymbolDict*, Dict[Symbol, Union[float, complex, Expr, None]]]) – Symbol map to update from.

Returns

SymbolDict – Updated instance of SymbolDict.

values()

Returns the iterator to the underlying dict values.

Return type

ValuesView[Union[float, complex, Expr, None]]

class SymbolSet(*iterable=None*)

Bases: SymbolTypeKeyDictWrapper

The SymbolSet class holds symbols in an ordered set, with additional meta information.

It can be converted to SymbolDict by adding values to the symbols, using provided classmethods.

Parameters

iterable (Optional[Iterable[Union[str, Symbol]]], default: None) – Iterable of symbols.

Examples

```
>>> SymbolSet(["a", "b", "c"])
SymbolSet([a, b, c])
>>> SymbolSet(["a", Symbol("b"), "c"])
SymbolSet([a, b, c])
```

add(*symbol*)

Adds a symbol to the end of the ordered symbols.

The order is defined by the symbols first appearance.

Parameters

symbol (Union[str, Symbol]) – The symbol to be added.

Returns

Symbol – The added symbol.

Examples

```
>>> symbols = SymbolSet(["a", "b", "c"])
>>> symbols.add("b")
b
>>> symbols
SymbolSet([a, b, c])
>>> symbols.add("a1")
a1
>>> symbols
SymbolSet([a, b, c, a1])
```

clear()

Remove all symbols.

Return type

None

construct_from_array(*array*)

Constructs a SymbolDict object.

SymbolDict object is constructed from the original SymbolSet instance and the input array, such that the elements of the input array become values of each symbol from the set, according to the internal order.

Parameters

array (Union[ndarray, List]) – Array containing the symbols values.

Returns

SymbolDict – New SymbolDict instance.

construct_from_dict(*other*)

Constructs a SymbolDict object.

The new SybmolDict contains all the symbols from the original SymbolSet instance, which values are assigned to the values of the same symbols in the povided SymbolDict or dict object.

Parameters
other (Union[*SymbolDict*, dict]) – A symbol map.

Returns
SymbolDict – A new SymbolDict instance.

Raises

- **RuntimeError** – When the provided SymbolDict (or dict) object does not contain any of the
- **symbols from the SymbolSet instance.** –

Examples

```
>>> sd = SymbolDict(a=1, b=2, c=3)
>>> ss = SymbolSet(['a', 'b'])
>>> ss.construct_from_dict(sd)
SymbolDict({a: 1, b: 2})
>>> sd = {Symbol("b"): 2, Symbol("a"): 1, Symbol("c"): 3}
>>> ss.construct_from_dict(sd)
SymbolDict({a: 1, b: 2})
```

construct_random (*seed=0*, *mu=0.0*, *sigma=1.0*)

Constructs a SymbolDict object.

All the symbol values from the original SymbolSet are drawn randomly from a Gaussian distribution.

Parameters

- **seed** – Seed for random number generation.
- **mu** – Mean for the distribution.
- **sigma** – Sigma for the distribution.

Returns
SymbolDict – New SymbolDict instance.

construct_zeros ()

Constructs a SymbolDict object.

All the symbol values from the original SymbolSet instance are set to 0.

Returns
SymbolDict – New SymbolDict instance.

copy ()

Performs a deep copy of self.

Returns
Union[SymbolTypeKeyDictWrapper, *SymbolDict*, *SymbolSet*] – A deep copy of this object.

Examples

```
>>> symbols = SymbolSet(["a", "b", "c"])
>>> other = symbols.copy()
>>> symbols == other
```

(continues on next page)

(continued from previous page)

```
True
>>> symbols is other
False
>>> symbols.add("d")
d
>>> symbols == other
False
```

---

df()

Returns a Pandas dataframe containing information about the symbol set.

discard(**symbols*)

Discard symbols.

Parameters

symbols (Union[str, Symbol]) – Symbols to be discarded.

Returns

SymbolTypeKeyDictWrapper – Updated SymbolDict object.

classmethod from_circuit(*circuit*)

Instantiate from a circuit.

Parameters

circuit (Circuit) – Circuit object.

Returns

Union[SymbolTypeKeyDictWrapper, *SymbolDict*, *SymbolSet*] – SymbolTypeKeyDictWrapper, SymbolDict or SymbolSet object.

make_hashable()

Alias for str(self).

renamed(*name_function=None*, *prefix='new_'*)

Returns a new set with the same order but renamed with the *name_function*.

Parameters

- **name_function** (Optional[Callable[[str], str]], default: None) – Function to rename symbols.
- **prefix** (str, default: "new_") – If *name_function* not provided, the prefix is added in front of every symbol.

Returns

SymbolSet – A new SymbolSet object.

Examples

```
>>> SymbolSet(["a", "b", "c"]).renamed()
SymbolSet([new_a, new_b, new_c])
>>> SymbolSet(["a", "b", "c"]).renamed(lambda s: f"renamed_{s}")
SymbolSet([renamed_a, renamed_b, renamed_c])
```

---

property symbols: List[Symbol]

Returns a list of all symbols.

Return type
List[Symbol]

update(*other*)

Update the symbol set.

The order is defined by the symbols first appearance.

Parameters
other (Iterable[Union[str, Symbol]]) – Symbols to update from.

Returns
SymbolSet – Updated instance of SymbolSet.

22.4.2 Logging and Timing

class Timer

Bases: object

Simple Timer.

start()

Starts timer.

stop()

Stops timer.

class TimerWith(*name=None*, *process=False*)

Bases: object

Basic timer context manager.

Parameters

- **name** (Optional[str], default: None) – Name for the context.
- **process** (bool, default: False) – Whether to use process timing.

Examples

```
>>> with TimerWith():
...     for i in range(1000000):
...         a = i ** 2
#...
#...
>>> with TimerWith("OUTER"):
...     for i in range(1000000):
...         a = i ** 2
...     with TimerWith("INNER"):
...         for i in range(1000000):
...             a = i ** 2
#...
#...
#...
#...
```

block_counter = 0

class InQuantoContext(*job_name*, *working_file_name=None*, *file_only=False*)

Bases: object

Context to log into a file during a calculation.

Parameters

- **job_name** (str) – Name of the job (this will be part of the file name).
- **working_file_name** (Optional[str], default: None) – Name of this file, for example __file__.
- **file_only** (bool, default: False) – Whether to suppress std outputs.

property base

Filename base.

property prefix

Filename prefix.

22.5 inquanto.embeddings

inquanto.embeddings module provides classes to decompose a system to smaller fragments for which more accurate theories can be applied.

22.5.1 DMET

InQuanto submodule for DMET (Density Matrix Embedding Theory).

This submodule contains a collection of DMET classes and functions to perform a DMET simulation for chemistry Hamiltonian.

class DMETRHF(*hamiltonian_operator*, *one_body_rdm_rhf*, *scf_max_iteration=20*, *scf_tolerance=1e-5*, *newton_maxiter=5*, *newton_tol=1e-5*, *occupation_rtol=1e-5*, *occupation_atol=1e-8*)

Bases: object

Basic RHF-DMET class to drive the DMET computations given the fragment solvers.

All density matrices are represented in spatial orbitals.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian in orthogonal localized basis.
- **one_body_rdm_rhf** (Union[ndarray, *RestrictedOneBodyRDM*]) – RHF RDM in orthogonal localized basis.
- **scf_max_iteration** (int, default: 20) – Max iteration for the outer scf loop.
- **scf_tolerance** (float, default: 1e-5) – SCF convergence tolerance for the outer loop for abs(u - u') < scf_tolerance.
- **newton_maxiter** (int, default: 5) – Max iteration for the newton solver inner loop (chemical potential).
- **newton_tol** (float, default: 1e-5) – Convergence tolerance for the chemical potential, abs(N-N') < newton_tol.

- **occupation_rtol** (float, default: 1e-5) – Relative tolerance (isclose) to check orbital occupation.
- **occupation_atol** (float, default: 1e-8) – Absolute tolerance (isclose) to check orbital occupation.

static construct_random_parameters (*pattern*, *seed=0*, *mu=0.0*, *sigma=0.01*)

Generates an array of random initial parameters with Gaussian distribution.

Parameters

- **pattern** (ndarray) – Pattern for correlation potential matrix.
- **seed** (int, default: 0) – Seed for the random number generator.
- **mu** (float, default: 0.0) – Mean for the distribution.
- **sigma** (float, default: 0.01) – Sigma for the distribution.

Returns

ndarray – Array of random numbers.

static correlation_potential_pattern (*pattern*, *parameters*)

Constructs the correlation potential matrix from the correlation potential matrix pattern and the parameters array.

Parameters

- **pattern** (ndarray) – Correlation potential matrix pattern.
- **parameters** (ndarray) – Array of parameter values.

Returns

Correlation potential matrix.

Examples

```
>>> pattern = numpy.array(
...     [
...         [1, 0, None, None, None, None],
...         [0, 1, None, None, None, None],
...         [None, None, 1, 0, None, None],
...         [None, None, 0, 1, None, None],
...         [None, None, None, None, 1, 0],
...         [None, None, None, None, 0, 1],
...     ]
... )
>>> DMETRHF.correlation_potential_pattern(pattern, [0.5, -0.5])
array([[-0.5,  0.5,  0. ,  0. ,  0. ,  0. ],
       [ 0.5, -0.5,  0. ,  0. ,  0. ,  0. ],
       [ 0. ,  0. , -0.5,  0.5,  0. ,  0. ],
       [ 0. ,  0. ,  0.5, -0.5,  0. ,  0. ],
       [ 0. ,  0. ,  0. ,  0. , -0.5,  0.5],
       [ 0. ,  0. ,  0. ,  0. ,  0.5, -0.5]])
```

---

energy (*fragments*)

Extracting the total energy form the fragment solvers.

Note: This function must be called after run(…).

Parameters

fragments (List[Union[*DMETRHFFragment*, *DMETRHFFragmentDirect*]]) – List of fragment solvers.

Returns

float – The total energy.

static pattern_from_locations(*size*, *locations*)

Convert locations to patterns, see correlation_potential_pattern().

Parameters

- **size** (int) – Size of the correlation potential matrix.
- **locations** (List[List[Tuple[int, int]]]) – For each parameter it store the i, j indices in the correlation potential matrix.

Returns

ndarray – Correlation potential pattern matrix.

Examples

```
>>> locations = [[(0,1), (1,0), (2,3), (3,2), (4,5), (5,4)], [(0,0), (1,1),␣
↪(2,2), (3,3), (4,4), (5,5)]]
>>> DMETRHF.pattern_from_locations(6, locations)
array([[ 1,  0, -1, -1, -1, -1],
       [ 0,  1, -1, -1, -1, -1],
       [-1, -1,  1,  0, -1, -1],
       [-1, -1,  0,  1, -1, -1],
       [-1, -1, -1, -1,  1,  0],
       [-1, -1, -1, -1,  0,  1]])
>>> DMETRHF.correlation_potential_pattern(DMETRHF.pattern_from_locations(6,␣
↪locations), [0.5, -0.5])
array([[-0.5,  0.5,  0. ,  0. ,  0. ,  0. ],
       [ 0.5, -0.5,  0. ,  0. ,  0. ,  0. ],
       [ 0. ,  0. , -0.5,  0.5,  0. ,  0. ],
       [ 0. ,  0. ,  0.5, -0.5,  0. ,  0. ],
       [ 0. ,  0. ,  0. ,  0. , -0.5,  0.5],
       [ 0. ,  0. ,  0. ,  0. ,  0.5, -0.5]])
```

---

run(*fragments*, *parameter_pattern=None*, *initial_parameters=None*, *initial_chemical_potential=0.0*)

Runs the DMET self consistency calculation.

Parameters

- **fragments** (List[Union[*DMETRHFFragment*, *DMETRHFFragmentDirect*]]) – List of fragment solvers.
- **parameter_pattern** (Optional[ndarray], default: None) – Parameter pattern matrix for the correlation potential.
- **initial_parameters** (Optional[ndarray], default: None) – Initial parameters (if None, it is randomly generated).
- **initial_chemical_potential** (float, default: 0.0) – Initial chemical potential.

Returns

Tuple[float, float, ndarray] – Energy, chemical potential, parameters defined by the DMET method.

run_one (*fragments*, *chemical_potential=0.0*)

Runs the DMET at a particular chemical_potential when all the correlation potentials are zero.

Parameters

- **fragments** (List[Union[*DMETRHFFragment*, *DMETRHFFragmentDirect*]]) – List of fragment solvers.
- **chemical_potential** (float, default: 0.0) – Chemical potential.

Returns

float – The total energy at a particular chemical potential.

class DMETRHFFragment (*dmet*, *mask*, *name=None*)

Bases: BaseDMETRHFFragment

Fragment class for DMET RHF.

This class is used to specify an individual fragment within a fragmentation scheme for DMET. Orbitals are specified with a mask, as described below.

This class does not contain logic for solving energetics of the system, as this will typically require an interface to a classical chemistry package. Instead, it forms a base from which fragment classes including this logic may subclass, with the abstract solve() method being implemented. These subclasses may be user-written, or may be provided by InQuanto extensions. For further detail, please see the InQuanto user guide.

Parameters

- **dmet** (DMETRHF) – DMET RHF object that uses this fragment.
- **mask** (ndarray) – Array of boolean, indices correspond to the spatial orbitals of the total system. If the corresponding orbital is in the fragment, then the boolean should be True - else it should be False.
- **name** (str, default: None) – Name of the fragment.

static compute_fragment_energy (*rdm1*, *rdm2*, *one_body*, *two_body*, *veff*, *mask*)

Computes the fragment energy with symmetrization.

Parameters

- **mask** (ndarray) – Fragment orbitals boolean mask.
- **rdm1** (ndarray) – One-body density matrix in the fragment basis.
- **rdm2** (ndarray) – Two-body density matrix in the fragment basis.
- **one_body** (ndarray) – One-body integrals in the fragment basis.
- **two_body** (ndarray) – Two-body integrals in the fragment basis.
- **veff** (ndarray) – Fock matrix in the fragment basis.

Returns

float – Fragment energy.

abstract solve (*hamiltonian_operator*, *n_electron*)

This is an abstract method.

The subclass implementation must return an accurate solution of the embedding system (fragment+bath) that includes the energy and one- and two-particle RDM-s.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian of the embedding system (fragment and bath).

- **n_electron** (int) – Number of electrons in the embedding system.

Returns

Tuple[float, ndarray, ndarray] – Energy, rdm1, rdm2 of the embedding system (fragment and bath).

class DMETRHFFragmentActive(*dmet*, *mask*, *name=None*, *frozen=None*)

Bases: *DMETRHFFragmentDirect*

This is a base class for fragments that reduce the fragment problem for an active space.

This class is used to specify an individual fragment within a fragmentation scheme for DMET. Orbitals are specified with a mask, as described below.

This class does not contain logic for solving energetics of the system, as this will typically require an interface to a classical chemistry package. Instead, it forms a base from which fragment classes including this logic may subclass, with the abstract `solve_active ()` method being implemented. These subclasses may be user-written, or may be provided by InQuanto extensions. For further detail, please see the InQuanto user guide.

Parameters

- **dmet** (DMETRHF) – DMET RHF that uses this fragment.
- **mask** (ndarray) – Array of boolean, indices correspond to the spatial orbitals of the total system, the fragment is masked with True.
- **name** (str, default: None) – Name of the fragment.
- **frozen** (List[int], default: None) – List of indices of frozen spatial orbitals.

static construct_fragment_energy_operator(*one_body*, *two_body*, *veff*, *mask*)

Computes the fragment energy with symmetrization.

Parameters

- **mask** (ndarray) – Fragment orbitals boolean mask.
- **one_body** (ndarray) – One-body integrals in the fragment basis.
- **two_body** (ndarray) – Two-body integrals in the fragment basis.
- **veff** (ndarray) – Fock matrix in the fragment basis.

Returns

ChemistryRestrictedIntegralOperator – Fragment energy operator based on democrating mixing defined by DMET.

solve(*hamiltonian_operator*, *fragment_energy_operator*, *n_electron*)

Solves the fragment system.

This method first runs an RHF for the fragment system, then for the active space it calls the solve_active(…) method.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator of the fragment system.
- **fragment_energy_operator** (*ChemistryRestrictedIntegralOperator*) – Fragment energy operator of the fragment system.
- **n_electron** (int) – Number of electrons in the fragment system.

Returns

Tuple[float, float, *RestrictedOneBodyRDM*] – Energy, fragment_energy, rdm1 of the embedding system (fragment and bath).

solve_active (*hamiltonian_operator*, *fragment_energy_operator*, *fermion_space*, *fermion_state*)

This abstract method requires an implementation to solve the active space problem of the embedding system.

The subclass implementation must return the energy, fragment energy, and the one-particle reduced density matrix (1-RDM) for the active space problem.

Note: This must be implemented by the child class.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator for the active space of the fragment system.
- **fragment_energy_operator** (*ChemistryRestrictedIntegralOperator*) – Fragment energy operator for the active space of the fragment system.
- **fermion_space** (*FermionSpace*) – Fermion space object for the active space of the fragment system.
- **fermion_state** (*FermionState*) – Fock state for the active space of the fragment system.

Returns

Tuple[float, float, *RestrictedOneBodyRDM*] – Energy, fragment energy, rdm1 of the active space of the embedding system (fragment and bath).

class DMETRHFFragmentDirect (*dmet*, *mask*, *name=None*)

Bases: BaseDMETRHFFragment

Fragment class for DMET RHF with direct Fragment energy evaluation.

This class is used to specify an individual fragment within a fragmentation scheme for DMET. Orbitals are specified with a mask, as described below.

This class does not contain logic for solving energetics of the system, as this will typically require an interface to a classical chemistry package. Instead, it forms a base from which fragment classes including this logic may subclass, with the abstract solve() method being implemented. These subclasses may be user-written, or may be provided by InQuanto extensions. For further detail, please see the InQuanto user guide.

Parameters

- **dmet** (DMETRHF) – DMET RHF that uses this fragment.
- **mask** (ndarray) – Array of boolean, indices correspond to the spatial orbitals of the total system, the fragment is masked with True.
- **name** (str, default: None) – Name of the fragment.

static construct_fragment_energy_operator (*one_body*, *two_body*, *veff*, *mask*)

Computes the fragment energy with symmetrization.

Parameters

- **mask** (ndarray) – Fragment orbitals boolean mask.
- **one_body** (ndarray) – One-body integrals in the fragment basis.

- **two_body** (ndarray) – Two-body integrals in the fragment basis.
- **veff** (ndarray) – Fock matrix in the fragment basis.

Returns

ChemistryRestrictedIntegralOperator – Fragment energy operator based on democrating mixing defined by DMET.

abstract solve(*hamiltonian_operator*, *fragment_energy_operator*, *n_electron*)

This is an abstract method.

The subclass implementation must return an accurate solution of the embedding system (fragment+bath) which includes the energy, the fragment energy and 1-RDM.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian of the embedding system (fragment and bath).
- **fragment_energy_operator** (*ChemistryRestrictedIntegralOperator*) – Fragment energy operator of the embedding system (fragment and bath).
- **n_electron** (int) – Number of electrons in the embedding system.

Returns

Tuple[float, float, *RestrictedOneBodyRDM*] – Energy, fragment_energy, rdm1 of the embedding system (fragment and bath).

class DMETRHFFragmentUCCSDVQE(*dmet*, *mask*, *name=None*, *frozen=None*, *backend=None*)

Bases: *DMETRHFFragmentActive*

This is a UCCSD VQE Fragment solver for DMETRHF.

Note: This fragment solver is limited to statevector VQE, if you wish to have a more advanced quantum algorithm it is recommended you subclass DMETRHFFragmentActive or DMETRHFFragmentPySCFActive and implement the solve_active(...) method with the custom quantum algorithm.

Parameters

- **dmet** (DMETRHF) – DMET RHF that uses this fragment.
- **mask** (ndarray) – Array of boolean, indices correspond to the spatial orbitals of the total system, the fragment is masked with True.
- **name** (str, default: None) – Name of the fragment.
- **frozen** (List[int], default: None) – List of indices of frozen spatial orbitals.
- **backend** (Backend, default: None) – A statevector backend.

static construct_fragment_energy_operator(*one_body*, *two_body*, *veff*, *mask*)

Computes the fragment energy with symmetrization.

Parameters

- **mask** (ndarray) – Fragment orbitals boolean mask.
- **one_body** (ndarray) – One-body integrals in the fragment basis.
- **two_body** (ndarray) – Two-body integrals in the fragment basis.
- **veff** (ndarray) – Fock matrix in the fragment basis.

Returns
ChemistryRestrictedIntegralOperator – Fragment energy operator based on democrating mixing defined by DMET.

solve (*hamiltonian_operator*, *fragment_energy_operator*, *n_electron*)

Solves the fragment system.

This method first runs an RHF for the fragment system, then for the active space it calls the solve_active(…) method.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator of the fragment system.
- **fragment_energy_operator** (*ChemistryRestrictedIntegralOperator*) – Fragment energy operator of the fragment system.
- **n_electron** (int) – Number of electrons in the fragment system.

Returns
Tuple[float, float, *RestrictedOneBodyRDM*] – Energy, fragment_energy, rdm1 of the embedding system (fragment and bath).

solve_active (*hamiltonian_operator*, *fragment_energy_operator*, *fermion_space*, *fermion_state*)

This method runs a UCCSD VQE calculation to solve the active space problem for the embedding system.

The implementation uses a statevector backend, and it returns the energy, fragment energy, and the one-particle reduced density matrix (1-RDM) for the active space problem.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator for the active space of the fragment system.
- **fragment_energy_operator** (*ChemistryRestrictedIntegralOperator*) – Fragment energy operator for the active space of the fragment system.
- **fermion_space** (*FermionSpace*) – Fermion space object for the active space of the fragment system.
- **fermion_state** (*FermionState*) – Fock state for the active space of the fragment system.

Returns
Tuple[float, float, *RestrictedOneBodyRDM*] – Energy, fragment energy, rdm1 of the active space of the embedding system (fragment and bath).

class ImpurityDMETROHF (*hamiltonian_operator*, *one_body_rdm_rhf*, *occupation_rtol=1e-5*, *occupation_atol=1e-8*)

Bases: object

Basic Impurity ROHF-DMET class to drive the ImpurityDMET computations given the single impurity fragment solver.

All density matrices are represented in spatial orbitals.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian in orthogonal localized basis.
- **one_body_rdm_rhf** (Union[ndarray, *RestrictedOneBodyRDM*]) – RHF RDM in orthogonal localized basis.

- **occupation_rtol** – Relative tolerance (isclose) to check orbital occupation.
- **occupation_atol** – Absolute tolerance (isclose) to check orbital occupation.

run (*fragment*)

Runs the impurity DMET calculation.

Parameters

fragment (Union[*ImpurityDMETROHFFragment*, *ImpurityDMETROHFFragmentWithoutRDM*]) – Impurity fragment solver.

Returns

float – DMET ground state energy.

class ImpurityDMETROHFFragment (*dmet*, *mask*, *name=None*, *multiplicity=1*)

Bases: BaseImpurityDMETROHFFragment

Fragment class for ImpurityDMETROHFF.

This class is used to specify an individual fragment solver within a fragmentation scheme for DMET. Orbitals are specified with a mask, as described below.

This class does not contain logic for solving energetics of the system, as this will typically require an interface to a classical chemistry package. Instead, it forms a base from which fragment classes including this logic may subclass, with the abstract solve() method being implemented. These subclasses may be user-written, or may be provided by InQuanto extensions. For further detail, please see the InQuanto user guide.

Parameters

- **dmet** (ImpurityDMETROHF) – ImpurityDMET ROHF that uses this fragment.
- **mask** (ndarray) – Array of boolean, indices correspond to the spatial orbitals of the total system, the fragment is masked with True.
- **name** (str, default: None) – Name of the fragment.
- **multiplicity** (int, default: 1) – Multiplicity for the fragment ROHF.

abstract solve (*hamiltonian_operator*, *n_electron*)

This is an abstract method.

The subclass implementation must return an accurate solution of the embedding system (fragment+bath) that includes the energy and the 1-RDM.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian of the embedding system (fragment and bath)
- **n_electron** (int) – Number of electrons in the embedding system

Returns

Any – Energy, rdm1 of the embedding system (fragment and bath).

class ImpurityDMETROHFFragmentActive (*dmet*, *mask*, *name=None*, *multiplicity=1*, *frozen=None*)

Bases: *ImpurityDMETROHFFragmentWithoutRDM*

This is a base class for fragments that reduce the fragment problem for an active space.

This class is used to specify an individual fragment within a fragmentation scheme for DMET. Orbitals are specified with a mask, as described below.

This class does not contain logic for solving energetics of the system, as this will typically require an interface to a classical chemistry package. Instead, it forms a base from which fragment classes including this logic may subclass,

with the abstract `solve_active ()` method being implemented. These subclasses may be user-written, or may be provided by InQuanto extensions. For further detail, please see the InQuanto user guide.

Parameters

- **dmet** (ImpurityDMETROHF) – ImpurityDMET ROHF that uses this fragment.
- **mask** (ndarray) – Array of boolean, indices correspond to the spatial orbitals of the total system, the fragment is masked with True.
- **name** (str, default: None) – Name of the fragment.
- **multiplicity** (int, default: 1) – Multiplicity for the fragment ROHF.
- **frozen** (Any, default: None) – List of indices of frozen spatial orbitals.

solve (*hamiltonian_operator*, *n_electron*)

Solves the fragment system.

This method first runs an ROHF for the fragment system, then for the active space it calls the solve_active(…) method.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator of the fragment system.
- **n_electron** (int) – Number of electrons in the fragment system.

Returns

float – Energy of the embedding system (fragment and bath).

solve_active (*hamiltonian_operator*, *fermion_space*, *fermion_state*)

This is an abstract method.

The subclass implementation must return an accurate ground state energy of the active space of the embedding system (fragment+bath).

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator for the active space of the fragment system.
- **fermion_space** (*FermionSpace*) – Fermion space object for the active space of the fragment system.
- **fermion_state** (*FermionState*) – Fock state for the active space of the fragment system.

Returns

float – Energy of the embedding system (fragment and bath).

class ImpurityDMETROHFFragmentED (*dmet*, *mask*, *name=None*)

Bases: *ImpurityDMETROHFFragmentWithoutRDM*

Exact diagonalization impurity solver for the Impurity DMET method.

This class is used to specify an individual fragment solver within a fragmentation scheme for DMET. Orbitals are specified with a mask, as described below.

Parameters

- **dmet** (ImpurityDMETROHF) – ImpurityDMETROHF instance that uses this fragment.
- **mask** (ndarray) – Array of boolean, indices correspond to the spatial orbitals of the total system, the fragment is masked with True.

- **name** (str, default: None) – Name of the fragment.

solve(*hamiltonian_operator*, *n_electron*)

This method returns the exact ground state energy of the embedding system (fragment+bath).

Note: This method should be used only for small fragments, the computational cost grows exponentially with fragment size.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian of the embedding system (fragment and bath).
- **n_electron** (int) – Number of electrons in the embedding system.

Returns

float – Energy of the embedding system (fragment and bath).

class ImpurityDMETROHFFragmentWithoutRDM(*dmet*, *mask*, *name=None*, *multiplicity=1*)

Bases: BaseImpurityDMETROHFFragment

RDM free fragment class solver for ImpurityDMETROHF.

The notable difference from the base fragment solver that this fragment solver does not require the computation of 1-RDM in its solve(...) method.

This class is used to specify an individual fragment within a fragmentation scheme for DMET. Orbitals are specified with a mask, as described below.

It may be used with a fragment solver (e.g. a quantum algorithm) for which direct rdm1 calculation is difficult.

This class does not contain logic for solving energetics of the system, as this will typically require an interface to a classical chemistry package. Instead, it forms a base from which fragment classes including this logic may subclass, with the abstract solve(...) method being implemented. These subclasses may be user-written, or may be provided by InQuanto extensions. For further detail, please see the InQuanto user guide.

Parameters

- **dmet** (ImpurityDMETROHF) – ImpurityDMET ROHF that uses this fragment.
- **mask** (ndarray) – Array of boolean, indices correspond to the spatial orbitals of the total system, the fragment is masked with True.
- **name** (str, default: None) – Name of the fragment.
- **multiplicity** (int, default: 1) – Multiplicity for the fragment ROHF.

abstract solve(*hamiltonian_operator*, *n_electron*)

This is an abstract method.

The subclass implementation must return an accurate ground state energy of the embedding system (fragment+bath).

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian of the embedding system (fragment and bath).
- **n_electron** (int) – Number of electrons in the embedding system.

Returns

Any – Energy of the embedding system (fragment and bath).

22.6 inquanto.express

Data for example systems for testing.

class DriverGeneralizedHubbard(*e=0.0*, *u=2.0*, *t=-1.0*, *v=0.0*, *n=2*, *ring=False*)

Bases: `GeneralDriver`

Creates Generalized Hubbard Hamiltonian with ring and chain topology.

Parameters

- **e** (`float`, default: `0.0`) – On-site energy.
- **u** (`float`, default: `2.0`) – U on-site repulsion.
- **t** (`float`, default: `-1.0`) – Nearest neighbour coupling.
- **v** (`float`, default: `0.0`) – Nearest neighbour coulomb repulsion.
- **n** (`int`, default: `2`) – Number of sites.
- **ring** (`bool`, default: `False`) – ring (True) or standalone chain (False).

static generate_chain(*n*, *diagonal*, *off_diagonal*)

Generates nearest neighbour connectivity matrix for chain topology.

Parameters

- **n** (`int`) – Size of the matrix.
- **diagonal** (`float`) – Value substituted to the diagonals
- **off_diagonal** (`float`) – Value substituted to the off-diagonals

Returns

The nearest neighbour connectivity matrix as a sparse matrix.

generate_report(**args*, ***kwargs*)

Generates report in a hierarchical dictionary format.

static generate_ring(*n*, *diagonal*, *off_diagonal*)

Generates nearest neighbour connectivity matrix for ring topology.

Parameters

- **n** (`int`) – Size of the matrix.
- **diagonal** (`float`) – Value substituted to the diagonals
- **off_diagonal** (`float`) – Value substituted to the off-diagonals

Returns

The nearest neighbour connectivity matrix as a sparse matrix.

get_system()

Generates the Hubbard system.

Returns

`Tuple`[*FermionOperator*, *FermionSpace*, *FermionState*] – Hamiltonian fermion operator, Fock space, Hartree-Fock state.

property n_electron: int

Returns the number of electrons in the total system.

Return type
int

property n_orb: int

Returns the number of orbitals in the total system.

Return type
int

print_json_report(**args*, ***kwargs*)

Prints report in json format.

class DriverHubbardDimer(*t=0.2*, *u=2.0*)

Bases: GeneralDriver

Driver creating a dimer Hubbard Hamiltonian.

Parameters

- **t** (float, default: 0.2) – Coupling energy. Note that here t is negated in the hamiltonian_operator generated by self.get_system(), which is not the case in DriverGeneralizedHubbard.get_system().
- **u** (float, default: 2.0) – On-site repulsion.

generate_report(**args*, ***kwargs*)

Generates report in a hierarchical dictionary format.

get_system()

Generates the Hubbard dimer.

Returns
Tuple[*FermionOperator*, *FermionSpace*, *FermionState*] – Hamiltonian fermion operator, Fock space, Hartree-Fock state.

property n_electron: int

Returns the number of electrons in the total system.

Return type
int

property n_orb: int

Returns the number of orbitals in the total system.

Return type
int

print_json_report(**args*, ***kwargs*)

Prints report in json format.

list_h5()

Lists the predefined h5 data files in the express module.

load_h5(*name*, *as_tuple=False*)

Loads predefined results from the express module.

Parameters

- **name** (Union[str, Group]) – The name of the h5 file with extension in the express module.

- **as_tuple** (bool, default: False) – If set to True, output will be returned as a namedtuple.

Returns
Union[Dict[str, Any], Tuple] – Dictionary or namedtuple format of the results.

Raises
TypeError – If unsupported operator type is loaded.

run_rhf (*hamiltonian_operator*, *n_electron*, *guess_mo_coeff=None*, *maxit=MAXIT*, *conv=CONV*)

Runs a simple RHF calculation for hamiltonian_operator.

> **Warning:** It is not advised for production calculation, but for testing.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – A Hamiltonian operator.
- **n_electron** (int) – number of electrons.
- **guess_mo_coeff** (Optional[ndarray], default: None) – initial guess for mo_coeff.
- **maxit** (int, default: MAXIT) – Maximum number of iteration.
- **conv** (float, default: CONV) – Criterion for convergence.

Returns
Total energy, orbital energies, mo orbitals, density matrix.

run_rohf (*hamiltonian_operator*, *n_electron*, *spin*, *guess_mo_coeff=None*, *maxit=MAXIT*, *conv=CONV*)

Runs a simple ROHF calculation for hamiltonian_operator.

> **Warning:** It is not advised for production calculation, but for testing.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – A hamitlonian operator.
- **n_electron** (int) – Number of electrons.
- **spin** (int) – Spin.
- **guess_mo_coeff** (Optional[ndarray], default: None) – Initial guess for mo_coeff.
- **maxit** (int, default: MAXIT) – Maximum number of iteration.
- **conv** (float, default: CONV) – Criterion for convergence.

Returns
Total energy, orbital energies, mo orbitals, density matrix.

run_vqe (*ansatz*, *hamiltonian*, *backend*, *with_gradient=True*, *minimizer=MinimizerScipy(method='L-BFGS-B')*, *initial_parameters=None*)

Performs a black-box, state-vector-only variational quantum eigensolver simulation.

This is a wrapper over the instantiation of the AlgorithmVQE class and execution of its build() and run() methods. Most of the parameters have default values.

Parameters

- **ansatz** (*CircuitAnsatz*) – an ansatz object.
- **hamiltonian** (Union[*QubitOperator*, BaseChemistryIntegralOperator]) – a Hamiltonian object.
- **backend** (Backend) – a backend object to perform calculation with.
- **with_gradient** (bool, default: True) – whether to use objective function gradient in the VQE calculation
- **minimizer** (GeneralMinimizer, default: MinimizerScipy(method="L-BFGS-B")) – variational classical minimizer to perform the parameter search.
- **initial_parameters** (Optional[*SymbolDict*], default: None) – a set of initial Ansatz parameters. If not provided, defaulted to all zeros.

Returns

AlgorithmVQE – AlgorithmVQE object after the AlgorithmVQE.build().run() method has been executed.

Examples

```
>>> from inquanto.express import load_h5
>>> from inquanto.states import FermionState
>>> from inquanto.spaces import FermionSpace
>>> from inquanto.ansatzes import FermionSpaceAnsatzUCCSD
>>> from pytket.extensions.qiskit import AerStateBackend
>>> hamiltonian = load_h5("h2_sto3g.h5", as_tuple=True).hamiltonian_operator.
↪qubit_encode()
>>> backend = AerStateBackend()
>>> state = FermionState([1, 1, 0, 0])
>>> space = FermionSpace(4)
>>> ansatz = FermionSpaceAnsatzUCCSD(fermion_space=space, fermion_state=state)
>>> vqe = run_vqe(ansatz, hamiltonian, backend)
# TIMER ...
>>> print(round(vqe.final_value, 8))
-1.13684658
```

22.7 inquanto.geometries

This Module contains classes useful for constructing and manipulating molecular and periodic geometries.

class GeometryMolecular(*geometry=None*, *distance_units='angstrom'*)

Bases: Geometry

Geometry class for handling and manipulating molecular systems.

Various transformations of the system in 3D space are supported. The default units are Angstroms and degrees. Alternatively, one can use Bohrs and radians.

Parameters

- **geometry** (Union[str, List, DataFrame, None], default: None) – Geometrical information - can be a list of lists (e.g [['H', [0, 0, 0]], ['H', [0, 0, 1]]]), a string z-matrix, string xyz coordinates or DataFrame.

- **distance_units** (str, default: "angstrom") – Specification of distance units. Supported units are Angstroms ('angstrom') and Bohrs ('bohr').

add_atom(*element*, *position=None*)

Add an atom to the geometry.

If no geometry.df attribute exists, a new one will be created.

Parameters

- **element** (str) – Element symbol.
- **position** (Optional[ndarray[Any, dtype[float]]], default: None) – Cartesian position vector. If omitted, generates a random position with coordinates between +/-1 Angstroms.

Returns

DataFrame – The updated dataframe object.

align_bond_to_axis(*atom_ids*, *axis*)

Align any bond to an axis.

Rotate and translate the geometry in the Geometry.df attribute to align the bond defined by the elements of the atom_ids list to a co-ordinate axis.

Parameters

- **atom_ids** (List) – Length two list of atom indices which define the bond to align to the axis.
- **axis** (str) – A cartesian co-ordinate axis, options are "x", "y", "z".

Returns

DataFrame – The updated dataframe object.

align_bond_to_vector(*atom_ids*, *vector*)

Align any bond to any user-defined vector.

Rotate and translate the geometry in the Geometry.df attribute to align the bond defined by the elements of the atom_ids list to a vector.

Parameters

- **atom_ids** (List[int]) – Indices of the atoms defining the bond to align to a vector.
- **vector** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – A length 3 array defining a vector.

Returns

DataFrame – The updated dataframe object.

align_to_plane(*atom_ids*, *vectors*)

Rotate and translate the geometry such that 3 atoms lay in the plane defined by the vectors provided.

Parameters

- **atom_ids** (List[int]) – List containing 3 atom indices defining a plane.
- **vectors** (List[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]]) – A list or other iterable containing three vectors.

Returns

DataFrame – The updated dataframe object.

align_to_xy_plane(*atom_ids=None*)

Align any 3 atoms to the xy plane.

Rotate and translate the geometry such that the plane defined by the 3 atom indices in the atom_ids variable becomes the xy plane.

Parameters

atom_ids (Optional[List[int]], default: None) – List containing 3 atom indices defining a plane.

Returns

DataFrame – The updated dataframe object.

align_to_xz_plane(*atom_ids=None*)

Align any three atoms to the xz plane.

Rotate and translate the geometry such that the plane defined by the 3 atom indices in the atom_ids variable becomes the xz plane.

Parameters

atom_ids (Optional[List[int]], default: None) – List containing 3 atom indices defining a plane.

Returns

DataFrame – The updated dataframe object.

align_to_yz_plane(*atom_ids=None*)

Align any 3 atoms to the yz plane.

Rotate and translate the geometry such that the plane defined by the 3 atom indices in the atom_ids variable becomes the yz plane.

Parameters

atom_ids (Optional[List[int]], default: None) – List containing 3 atom indices defining a plane.

Returns

DataFrame – The updated dataframe object.

property atomic_coordinates: ndarray[Any, dtype[ScalarType]]

Get an array of the position vectors of each atom in the geometry.

Each row corresponds to one position vector.

Returns

ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – :

An array of floats, each row is the x, y, z coordinate of the atom which shares the index of the row.

bond_angle(*atom_ids*, *units='deg'*)

Compute the bond angle defined by the three atoms indexed by the elements of atom_ids.

Compute the bond angle defined by the 3 atoms specified in atom_ids. The angle is at the middle atom, or second index in the atom_ids list. For example, to compute the bond angle at the oxygen atom in water, one would pass [index of H1, index of O, index of H2].

Parameters

- **atom_ids** (List[int]) – List or other iterable containing 3 atom indices to use in calculating the angle.
- **units** (str, default: "deg") – Units of the reported bond angle, options are “deg” or “rad”. Default is radians.

Returns
float – The bond angle in the specified units.

bond_length(*atom_ids*)

Compute the inter-nuclear separation between the two atoms indexed by the elements of atom_ids.

Parameters
atom_ids (List[int]) – List or other iterable containing the indices of the atoms defining the bond.

Returns
float – Internuclear separation in the units of the geometry object.

build_2atom_chain(*atom='H'*, *num_pair=4*, *d_intra=0.75*, *d_inter=0.75*)

Construct xyz geometry for a chain of homonuclear diatomics.

Parameters
- **atom** (str, default: "H") – Element symbol.
- **num_pair** (int, default: 4) – Number of diatomics in the chain.
- **d_intra** (float, default: 0.75) – Bond length in each diatomic unit of the chain.
- **d_inter** (float, default: 0.75) – Inter-molecular separation.

Return type
None

build_alternating_ring(*element_a*, *a*, *bb*, *b*, *n*)

Builds a ring geometry of atoms as -A–B–A–B-…

Parameters
- **element_a** (str) – Atomic symbol.
- **a** (float) – Distance between A and B in the AB pairs.
- **bb** (str) – Atomic symbol.
- **b** (float) – B to A distance between AB pairs.
- **n** (int) – Number of A–B atom pairs in the ring.

Return type
None

build_rectangle(*element*, *dx*, *nx*, *dy=0*, *ny=1*)

Builds a rectangular grid geometry of atoms.

Parameters
- **element** (str) – Atomic symbol.
- **dx** (float) – Distance between the atoms in x direction.
- **nx** (int) – Number of atoms in the x direction.
- **dy** (float, default: 0) – Distance between the atoms in y direction.

- **ny** (int, default: 1) – Number of atoms in the y direction.

Return type
None

build_ring(*element*, *d*, *n*)

Builds a ring geometry of atoms.

Parameters

- **element** (str) – Atomic symbol.
- **d** (float) – Distance between the atoms in the ring.
- **n** (int) – Number of atoms in the ring.

Return type
None

compute_distance_matrix()

Compute a distance matrix where each i, jth element is the internuclear separation between atom i and j.

Returns
The distance matrix – a matrix wherein element (i,j) is the distance between atom i and j.

delete_atom(*atom_index*)

Delete an atom from the geometry.

Parameters
atom_index – Index of the atom to delete.

Returns
DataFrame – The updated dataframe object.

df_to_xyz()

Convert the pandas.DataFrame held in self.df to an xyz geometry.

Returns
List[List[Union[str, List[float]]]] – The xyz geometry corresponding to the xyz geometry defined in the dataframe.

dihedral_angle(*atom_ids*, *units='deg'*)

Compute the dihedral angle defined by the 4 atoms specified in atom_ids.

Parameters

- **atom_ids** (List[int]) – The 4 atom indices.
- **units** (str, default: "deg") – The units the angle is reported in, options are “deg”, “rad”.

Returns
float – The dihedral angle defined by the 4 atoms indexed in atom_ids, given in the units specified.

property elements: List[str]

Return a list of the chemical symbols present in the geometry.

Return type
List[str]

static from_xyz_string(*xyz_string*, *distance_units='angstrom'*)

Create a Geometry object from a string containing symbols and x, y and z coordinates on each new line.

Parameters

- **xyz_string** (str) – A string in which each new line contains the element symbol and x, y, and z positions separated by a space.
- **distance_units** (str, default: "angstrom") – The geometrical units of the system, can be "angstrom" or "bohr".

Returns
Geometry – A Geometry object containing the geometry provided in the xyz_string argument.

load_csv (*fn*)

Load a csv file into a pandas.DataFrame object and store it in the Geometry.df attribute.

Parameters
fn (str) – The input filename.

Return type
None

load_json (*fn*)

Load a json file into a pandas.DataFrame object, store in the Geometry.df attribute of the Geometry object.

Parameters
fn (str) – The input filename.

Return type
None

static load_xyz (*filename*, *distance_units='angstrom'*)

Load geometry from an xyz file.

Currently, only the xyz file format is supported.

Parameters

- **filename** (str) – Filename of the xyz file.
- **distance_units** (str, default: "angstrom") – Distance units the geometry is expressed in. Options are "angstrom" or "bohr".

Returns
Geometry – The loaded geometry.

load_zmatrix (*filename*, *distance_units='angstrom'*)

Load a zmatrix from file into the Geometry.zmatrix object attribute.

Parameters

- **filename** (str) – Filename.
- **distance_units** (str, default: "angstrom") – Distance units, 'angstrom' or 'bohr'.

Return type
None

modify_bond_angle (*atom_ids*, *theta*, *units='deg'*)

Change the bond angle at the atom with index matching the second element of atom_ids.

The atom indexed by the third element of atom_ids has its position updated in the Geometry.df attribute.

Parameters

- **atom_ids** (List[int]) – Atom indices.
- **theta** (float) – New bond angle.

- **units** (str, default: "deg") – Units of the bond angle, options are 'rad' or 'deg'.

Returns
DataFrame – The updated dataframe object,

modify_bond_angle_by_group (*atom_ids*, *theta*, *group*, *units='deg'*)

Modify the bond angle between two groups of atoms.

Modify the bond angle at the second element of atom_ids, moving the third element of atom_ids and the subgroup it belongs to by the same angle. Elements 1 and 2 of atom_ids must belong to a different subgroup to the final element. The structure within each subgroup is preserved.

Parameters

- **atom_ids** (List[int]) – Atom indices.
- **theta** (float) – New bond angle.
- **group** (str) – Grouping scheme name.
- **units** (str, default: "deg") – Units of the angle, "deg" or "rad".

Returns
DataFrame – The updated dataframe object.

modify_bond_length (*atom_ids*, *new_bond_length*)

Change the internuclear separation between two atoms.

The first atom in atom_ids is fixed, the second atom is moved.

Parameters

- **atom_ids** (List[int]) – A list of two atom indices.
- **new_bond_length** (float) – Bond length.

Returns
DataFrame – The updated dataframe object.

modify_bond_length_by_group (*atom_ids*, *bond_length*, *group*)

Modify the bond length connecting two groups of atoms.

Modify the bond length connecting two atoms in different subgroups. Each subgroup's internal structure is preserved. The two atoms specified in atom_ids must belong to different subgroups as defined by the group argument. The subgroup translated is the one containing the second atom indexed in atom_ids.

Parameters

- **atom_ids** (List[int]) – Atom indices.
- **bond_length** (float) – New bond length.
- **group** (str) – Group name.

Returns
DataFrame – The updated dataframe object.

modify_dihedral_angle (*atom_ids*, *theta*, *units='deg'*)

Modify a dihedral angle.

Change the dihedral angle defined by the 4 atoms indexed by the elements of atom_ids. The atom indexed by the last element has its position updated in dataframe.

Parameters

- **atom_ids** (List[int]) – Atom indices.

- **theta** (float) – Dihedral angle.
- **units** (str, default: "deg") – Units of the dihedral angle, “rad” or “deg”.

Returns
DataFrame – The modified dataframe object.

modify_dihedral_angle_by_group(*atom_ids*, *theta*, *group*, *units='deg'*)

Modify a dihedral angle between two groups of atoms.

Modify the dihedral angle defined by atoms indexed in atom_ids. The structure within each group is retained. The 3rd and 4th elements of atom_ids must belong to the same label, and be different to the label defined by atoms indexed by the 1st and 2nd elements.

Parameters

- **atom_ids** (List[int]) – Atom indices.
- **theta** (float) – Dihedral angle.
- **group** (str) – Grouping label.
- **units** (str, default: "deg") – Units of the dihedral angle, “deg” or “rad”.

Returns
DataFrame – The updated dataframe object.

randomize_xyz(*sigma=0.05*, *seed=0*, *freeze_atoms=None*)

Add random numbers to the xyz geometry held in the Geometry.df attribute.

The random numbers are sampled from Gaussian distributions centred at each atomic position.

Parameters

- **sigma** (float, default: 0.05) – Sigma for the Gaussian distribution.
- **seed** (int, default: 0) – Random seed.
- **freeze_atoms** (Optional[List], default: None) – List of indices of atoms to freeze.

Returns
DataFrame – The updated dataframe object.

rescale_position_vectors(*factor*)

Multiply the atomic position vectors by a constant factor.

Parameters
factor (float) – Conversion factor to new coordinate frame.

Returns
DataFrame – The updated dataframe object.

rotate_around_axis(*theta*, *axis*, *units='deg'*)

Rotate the geometry stored in the Geometry.df attribute around one of the coordinate axes.

Parameters

- **theta** (float) – The Angle through which to rotate.
- **axis** (str) – the coordinate axis to rotate around.
- **units** – units of the angle specified, options are “deg”, “rad”.

Returns
DataFrame – The updated dataframe object.

rotate_around_vector(*vector*, *theta*, *units='deg'*)

Rotate the geometry held in the Geometry.df attribute around a vector defined by the input by theta.

Parameters

- **vector** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – The vector around which to rotate.
- **theta** (float) – The angle to rotate through.
- **units** (str, default: "deg") – The units of the angle, options are “deg”, “rad”.

Returns

DataFrame – The updated dataframe object.

save_csv(*fn*)

Write the current pandas.DataFrame object held by the geometry object to a csv file with name filename.

Parameters

fn (str) – The output filename.

Return type

None

save_json(*fn*)

Write the current pandas.DataFrame object held by the geometry object to a json file with name filename.

Parameters

fn (str) – The output filename.

Return type

None

save_xyz(*filename*)

Save geometry to an xyz file.

Currently, only the file format xyz is supported.

Parameters

filename (str) – Name of the file.

Return type

None

save_zmatrix(*filename*)

Write the zmatrix held in the Geometry.zmatrix attribute to file with name filename.

Parameters

filename (str) – Output filename.

Return type

None

scan_bond_angle(*atom_ids*, *bond_angles*, *units='deg'*)

Construct many Geometry objects corresponding to a scan of a bond angle.

Create a list of Geometry objects holding df attributes corresponding to a scan of the potential energy surface by bond angle. The modification of the angle moves only one atom - the atom corresponding to the third element of the atom_ids argument by index.

Parameters

- **atom_ids** (List[int]) – The atoms between which the angle is changed.

- **bond_angles** (List[float]) – A list of bond angles.

Returns

List[Geometry] – A list of Geometry objects corresponding the bond lengths specified in the input.

scan_bond_angle_by_group(*atom_ids*, *bond_angles*, *group*, *units='deg'*)

Scan a bond angle between two different groups while retaining the geometry of each group.

Parameters

- **atom_ids** (List[int]) – Atom indices.
- **bond_angles** (List[float]) – Bond angles to create geometry objects for.
- **group** (str) – Group heading.
- **units** (str, default: "deg") – Units of the angles defined in bond_angles.

Returns

List – A list of geometries corresponding to the description and bond_angles passed in.

scan_bond_length(*atom_ids*, *bond_lengths*)

Construct many Geometry objects corresponding to a scan of a bond length.

Create a list of Geometry objects holding df attributes corresponding to a scan of the potential energy surface by bond length. The bond stretching moves only one atom - the atom corresponding to the second element of the atom_ids argument by index.

Parameters

- **atom_ids** (List[int]) – The atoms between which the bond is stretched.
- **bond_lengths** (List[float]) – A list of bond lengths.

Returns

List[Geometry] – A list of Geometry objects corresponding the bond lengths specified in the input.

scan_bond_length_by_group(*atom_ids*, *bond_lengths*, *group*)

Scan a bond length between two different subgroups while retaining the geometry of each subgroup.

Parameters

- **atom_ids** (List[int]) – Atom indices.
- **bond_lengths** (List[float]) – Bond lengths to create geometry objects for.
- **group** (str) – Group heading.

Returns

List[Geometry] – A list of geometries corresponding to the description and bond_lengths passed in.

scan_dihedral_angle(*atom_ids*, *dihedral_angles*, *units='deg'*)

Construct many geometry objects corresponding to a scan of a dihedral angle.

Create a list of Geometry objects holding df attributes corresponding to a scan of the potential energy surface by dihedral angle. The modification of the angle moves only one atom - the atom corresponding to the fourth element of the atom_ids argument by index.

Parameters

- **atom_ids** – The atoms between which the dihedral is changed.

- **dihedral_angles** – A list of dihedral angles.

Returns

List[Geometry] – A list of Geometry objects corresponding the bond lengths specified in the input.

scan_dihedral_angle_by_group(*atom_ids*, *dihedral_angles*, *group*, *units='deg'*)

Scan a dihedral angle between atoms in different groups while retaining the geometry of each group.

Parameters

- **atom_ids** (List[int]) – Atom indices.
- **dihedral_angles** (List[float]) – Dihedral angles to create geometry objects for.
- **group** (str) – Group heading.
- **units** – Units of te angles defined in dihedral_angles.

Returns

List – A list of geometries corresponding to the description and dihedral angles passed in.

set_groups(*target*, *source*, *mapping=None*)

Set group information for the atoms in the system.

Target specifies the heading of the new column created in the underlying dataframe. The source argument can be either a dictionary, where the keys are the labels of the group and the values are the atom indices corresponding to the label, or a string. If source is a dictionary, the mapping argument does nothing. If source is a string, it must be an existing column heading in the dataframe. In which case, the mapping argument must be provided as a callable function which modifies the existing entries in the source column and inserts them into the new target column.

Parameters

- **target** (str) – Column in the table that refer to this grouping.
- **source** (str) – Another column or a dictionary defining the groups.
- **mapping** (Optional[Callable], default: None) – Optional mapping to transform group names.

Return type

None

set_subgroups(*target*, *pattern*, *subgroup*)

Set subgroups on the target grouping via regular expression pattern.

Parameters

- **target** (str) – Target column in the table, a table that has a grouping in it.
- **pattern** (str) – Regex pattern to match.
- **subgroup** (str) – Adding subgroups to the matching groups.

Return type

None

to_angstrom()

Convert the geometry in the Geometry.df attribute from Bohrs to Angstroms.

Returns

DataFrame – The updated dataframe object.

to_bohr()

Convert the geometry in the Geometry.df attribute from Angstroms to Bohrs.

Returns
DataFrame – The updated dataframe object.

to_zmatrix()

Map the geometry held in the .df attribute to a z-matrix in string format.

Atoms are taken in order, and z-matrix sequences are defined backwards through the geometry, for example, the first dihedral is defined by the 4th, 3rd, 2nd and 1st atoms in the geometry. The resulting z-matrix is returned and stored in the Geometry.zmatrix attribute.

Returns
str – The updated z-matrix attribute.

translate_by_vector(*v*)

Translate all atoms in the geometry held in the Geometry.df attribute by a vector, v.

Parameters
v (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – Vector defining the translation.

Returns
DataFrame – The updated dataframe object.

property xyz: List[List[str | List[float]]]

Return the geometry in xyz format.

Returns
List[List[Union[str, List[float]]]] – Geometry as a list of lists of atom symbols and positions

xyz_to_df(*xyz*)

Convert the xyz geometry to a dataframe and overwrites the current Geometry.df attribute accordingly.

Parameters
xyz (List[Any]) – An xyz geometry as a list of lists containing the element as a string and the position as a list of floats.

Returns
DataFrame – The DataFrame corresponding to the xyz argument.

zmatrix_to_df()

Convert the Zmatrix held in the Geometry.zmatrix attribute to a dataframe geometry.

The geometry.df attribute is updated accordingly.

Returns
DataFrame – The updated dataframe object geometry.

class GeometryPeriodic(*geometry=None*, *unit_cell=None*, *distance_units='angstrom'*)

Bases: Geometry

Geometry class for handling periodic systems.

Various transformations and extensions of the system in 3D space are supported. The default units are Angstroms and degrees. Alternatively, one can use bohrs and radians.

Parameters

- **geometry** (Union[str, List, DataFrame, None], default: None) – Geometrical information - can be a list of lists (e.g [['H', [0, 0, 0]], ['H', [0, 0, 1]]]), string xyz coordinates or DataFrame.
- **unit_cell** (Optional[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]], default: None) – The unit cell as a 3x3 array with each row being one cell vector.
- **distance_units** (*str*) – Specification of distance units. Supported units are Angstrom ('angstrom') and Bohrs ('bohr').

add_atom(*element*, *position=None*)

Add an atom to the geometry.

If no geometry.df attribute exists, a new one will be created.

Parameters

- **element** (str) – Element symbol.
- **position** (Optional[ndarray[Any, dtype[float]]], default: None) – Cartesian position vector. If omitted, generates a random position with coordinates between +/-1 Angstroms.

Returns

DataFrame – The updated dataframe object.

align_bond_to_axis(*atom_ids*, *axis*)

Align any bond to an axis.

Rotate and translate the geometry in the Geometry.df attribute to align the bond defined by the elements of the atom_ids list to a co-ordinate axis.

Parameters

- **atom_ids** (List) – Length two list of atom indices which define the bond to align to the axis.
- **axis** (str) – A cartesian co-ordinate axis, options are "x", "y", "z".

Returns

DataFrame – The updated dataframe object.

align_bond_to_vector(*atom_ids*, *vector*)

Align any bond to any user-defined vector.

Rotate and translate the geometry in the Geometry.df attribute to align the bond defined by the elements of the atom_ids list to a vector.

Parameters

- **atom_ids** (List[int]) – Indices of the atoms defining the bond to align to a vector.
- **vector** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – A length 3 array defining a vector.

Returns

DataFrame – The updated dataframe object.

align_to_plane(*atom_ids*, *vectors*)

Rotate and translate the geometry such that 3 atoms lay in the plane defined by the vectors provided.

Parameters

- **atom_ids** (List[int]) – List containing 3 atom indices defining a plane.
- **vectors** (List[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]]) – A list or other iterable containing three vectors.

Returns
DataFrame – The updated dataframe object.

align_to_xy_plane(*atom_ids=None*)

Align any 3 atoms to the xy plane.

Rotate and translate the geometry such that the plane defined by the 3 atom indices in the atom_ids variable becomes the xy plane.

Parameters
atom_ids (Optional[List[int]], default: None) – List containing 3 atom indices defining a plane.

Returns
DataFrame – The updated dataframe object.

align_to_xz_plane(*atom_ids=None*)

Align any three atoms to the xz plane.

Rotate and translate the geometry such that the plane defined by the 3 atom indices in the atom_ids variable becomes the xz plane.

Parameters
atom_ids (Optional[List[int]], default: None) – List containing 3 atom indices defining a plane.

Returns
DataFrame – The updated dataframe object.

align_to_yz_plane(*atom_ids=None*)

Align any 3 atoms to the yz plane.

Rotate and translate the geometry such that the plane defined by the 3 atom indices in the atom_ids variable becomes the yz plane.

Parameters
atom_ids (Optional[List[int]], default: None) – List containing 3 atom indices defining a plane.

Returns
DataFrame – The updated dataframe object.

property atomic_coordinates: ndarray[Any, dtype[ScalarType]]

Get an array of the position vectors of each atom in the geometry.

Each row corresponds to one position vector.

Returns

ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – :

An array of floats, each row is the x, y, z coordinate of the atom which shares the index of the row.

bond_angle(*atom_ids*, *units='deg'*)

Compute the bond angle defined by the three atoms indexed by the elements of atom_ids.

Compute the bond angle defined by the 3 atoms specified in atom_ids. The angle is at the middle atom, or second index in the atom_ids list. For example, to compute the bond angle at the oxygen atom in water, one would pass [index of H1, index of O, index of H2].

Parameters

- **atom_ids** (List[int]) – List or other iterable containing 3 atom indices to use in calculating the angle.
- **units** (str, default: "deg") – Units of the reported bond angle, options are "deg" or "rad". Default is radians.

Returns

float – The bond angle in the specified units.

bond_length(*atom_ids*)

Compute the inter-nuclear separation between the two atoms indexed by the elements of atom_ids.

Parameters

atom_ids (List[int]) – List or other iterable containing the indices of the atoms defining the bond.

Returns

float – Internuclear separation in the units of the geometry object.

build_2atom_chain(*atom='H'*, *num_pair=4*, *d_intra=0.75*, *d_inter=0.75*)

Construct xyz geometry for a chain of homonuclear diatomics.

Parameters

- **atom** (str, default: "H") – Element symbol.
- **num_pair** (int, default: 4) – Number of diatomics in the chain.
- **d_intra** (float, default: 0.75) – Bond length in each diatomic unit of the chain.
- **d_inter** (float, default: 0.75) – Inter-molecular separation.

Return type

None

build_alternating_ring(*element_a*, *a*, *bb*, *b*, *n*)

Builds a ring geometry of atoms as -A–B–A–B-…

Parameters

- **element_a** (str) – Atomic symbol.
- **a** (float) – Distance between A and B in the AB pairs.
- **bb** (str) – Atomic symbol.
- **b** (float) – B to A distance between AB pairs.
- **n** (int) – Number of A–B atom pairs in the ring.

Return type

None

build_rectangle(*element*, *dx*, *nx*, *dy=0*, *ny=1*)

Builds a rectangular grid geometry of atoms.

Parameters

- **element** (str) – Atomic symbol.
- **dx** (float) – Distance between the atoms in x direction.
- **nx** (int) – Number of atoms in the x direction.
- **dy** (float, default: 0) – Distance between the atoms in y direction.
- **ny** (int, default: 1) – Number of atoms in the y direction.

Return type
None

build_ring(*element*, *d*, *n*)

Builds a ring geometry of atoms.

Parameters

- **element** (str) – Atomic symbol.
- **d** (float) – Distance between the atoms in the ring.
- **n** (int) – Number of atoms in the ring.

Return type
None

build_supercell(*dimensions*)

Construct a supercell.

Repeat the unit cell geometry in each direction according to the elements of the provided dimensions list.

Parameters
dimensions (List[int]) – Number of times to repeat the unit cell in each direction.

Returns
DataFrame – The updated Dataframe object.

compute_distance_matrix()

Compute a distance matrix where each i, jth element is the internuclear separation between atom i and j.

Returns
The distance matrix – a matrix wherein element (i,j) is the distance between atom i and j.

delete_atom(*atom_index*)

Delete an atom from the geometry.

Parameters
atom_index – Index of the atom to delete.

Returns
DataFrame – The updated dataframe object.

df_to_xyz()

Convert the pandas.DataFrame held in self.df to an xyz geometry.

Returns
List[List[Union[str, List[float]]]] – The xyz geometry corresponding to the xyz geometry defined in the dataframe.

dihedral_angle(*atom_ids*, *units='deg'*)

Compute the dihedral angle defined by the 4 atoms specified in atom_ids.

Parameters

- **atom_ids** (List[int]) – The 4 atom indices.
- **units** (str, default: "deg") – The units the angle is reported in, options are “deg”, “rad”.

Returns

float – The dihedral angle defined by the 4 atoms indexed in atom_ids, given in the units specified.

property elements: List[str]

Return a list of the chemical symbols present in the geometry.

Return type

List[str]

static from_xyz_string(*xyz_string*, *distance_units='angstrom'*)

Create a Geometry object from a string containing symbols and x, y and z coordinates on each new line.

Parameters

- **xyz_string** (str) – A string in which each new line contains the element symbol and x, y, and z positions separated by a space.
- **distance_units** (str, default: "angstrom") – The geometrical units of the system, can be “angstrom” or “bohr”.

Returns

Geometry – A Geometry object containing the geometry provided in the xyz_string argument.

load_csv(*fn*)

Load a csv file into a pandas.DataFrame object and store it in the Geometry.df attribute.

Parameters

fn (str) – The input filename.

Return type

None

load_json(*fn*)

Load a json file into a pandas.DataFrame object, store in the Geometry.df attribute of the Geometry object.

Parameters

fn (str) – The input filename.

Return type

None

static load_xyz(*filename*, *distance_units='angstrom'*)

Load geometry from an xyz file.

Currently, only the xyz file format is supported.

Parameters

- **filename** (str) – Filename of the xyz file.
- **distance_units** (str, default: "angstrom") – Distance units the geometry is expressed in. Options are “angstrom” or “bohr”.

Returns
`Geometry` – The loaded geometry.

modify_bond_angle(*atom_ids*, *theta*, *units='deg'*)

Change the bond angle at the atom with index matching the second element of atom_ids.

The atom indexed by the third element of atom_ids has its position updated in the Geometry.df attribute.

Parameters

- **atom_ids** (`List[int]`) – Atom indices.
- **theta** (`float`) – New bond angle.
- **units** (`str`, default: `"deg"`) – Units of the bond angle, options are 'rad' or 'deg'.

Returns
`DataFrame` – The updated dataframe object,

modify_bond_angle_by_group(*atom_ids*, *theta*, *group*, *units='deg'*)

Modify the bond angle between two groups of atoms.

Modify the bond angle at the second element of atom_ids, moving the third element of atom_ids and the subgroup it belongs to by the same angle. Elements 1 and 2 of atom_ids must belong to a different subgroup to the final element. The structure within each subgroup is preserved.

Parameters

- **atom_ids** (`List[int]`) – Atom indices.
- **theta** (`float`) – New bond angle.
- **group** (`str`) – Grouping scheme name.
- **units** (`str`, default: `"deg"`) – Units of the angle, "deg" or "rad".

Returns
`DataFrame` – The updated dataframe object.

modify_bond_length(*atom_ids*, *new_bond_length*)

Change the internuclear separation between two atoms.

The first atom in atom_ids is fixed, the second atom is moved.

Parameters

- **atom_ids** (`List[int]`) – A list of two atom indices.
- **new_bond_length** (`float`) – Bond length.

Returns
`DataFrame` – The updated dataframe object.

modify_bond_length_by_group(*atom_ids*, *bond_length*, *group*)

Modify the bond length connecting two groups of atoms.

Modify the bond length connecting two atoms in different subgroups. Each subgroup's internal structure is preserved. The two atoms specified in atom_ids must belong to different subgroups as defined by the group argument. The subgroup translated is the one containing the second atom indexed in atom_ids.

Parameters

- **atom_ids** (`List[int]`) – Atom indices.
- **bond_length** (`float`) – New bond length.

- **group** (str) – Group name.

Returns
DataFrame – The updated dataframe object.

modify_dihedral_angle(*atom_ids*, *theta*, *units='deg'*)

Modify a dihedral angle.

Change the dihedral angle defined by the 4 atoms indexed by the elements of atom_ids. The atom indexed by the last element has its position updated in dataframe.

Parameters

- **atom_ids** (List[int]) – Atom indices.
- **theta** (float) – Dihedral angle.
- **units** (str, default: "deg") – Units of the dihedral angle, “rad” or “deg”.

Returns
DataFrame – The modified dataframe object.

modify_dihedral_angle_by_group(*atom_ids*, *theta*, *group*, *units='deg'*)

Modify a dihedral angle between two groups of atoms.

Modify the dihedral angle defined by atoms indexed in atom_ids. The structure within each group is retained. The 3rd and 4th elements of atom_ids must belong to the same label, and be different to the label defined by atoms indexed by the 1st and 2nd elements.

Parameters

- **atom_ids** (List[int]) – Atom indices.
- **theta** (float) – Dihedral angle.
- **group** (str) – Grouping label.
- **units** (str, default: "deg") – Units of the dihedral angle, “deg” or “rad”.

Returns
DataFrame – The updated dataframe object.

randomize_xyz(*sigma=0.05*, *seed=0*, *freeze_atoms=None*)

Add random numbers to the xyz geometry held in the Geometry.df attribute.

The random numbers are sampled from Gaussian distributions centred at each atomic position.

Parameters

- **sigma** (float, default: 0.05) – Sigma for the Gaussian distribution.
- **seed** (int, default: 0) – Random seed.
- **freeze_atoms** (Optional[List], default: None) – List of indices of atoms to freeze.

Returns
DataFrame – The updated dataframe object.

rescale_position_vectors(*factor*)

Multiply the atomic position vectors by a constant factor.

Parameters
factor (float) – Conversion factor to new coordinate frame.

Returns
DataFrame – The updated dataframe object.

rotate_around_axis(*theta*, *axis*, *units='deg'*)

Rotate the geometry stored in the Geometry.df attribute around one of the coordinate axes.

Parameters

- **theta** (float) – The Angle through which to rotate.
- **axis** (str) – the coordinate axis to rotate around.
- **units** – units of the angle specified, options are “deg”, “rad”.

Returns

DataFrame – The updated dataframe object.

rotate_around_vector(*vector*, *theta*, *units='deg'*)

Rotate the geometry held in the Geometry.df attribute around a vector defined by the input by theta.

Parameters

- **vector** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – The vector around which to rotate.
- **theta** (float) – The angle to rotate through.
- **units** (str, default: "deg") – The units of the angle, options are “deg”, “rad”.

Returns

DataFrame – The updated dataframe object.

save_csv(*fn*)

Write the current pandas.DataFrame object held by the geometry object to a csv file with name filename.

Parameters

fn (str) – The output filename.

Return type

None

save_json(*fn*)

Write the current pandas.DataFrame object held by the geometry object to a json file with name filename.

Parameters

fn (str) – The output filename.

Return type

None

save_xyz(*filename*)

Save geometry to an xyz file.

Currently, only the file format xyz is supported.

Parameters

filename (str) – Name of the file.

Return type

None

scan_bond_angle(*atom_ids*, *bond_angles*, *units='deg'*)

Construct many Geometry objects corresponding to a scan of a bond angle.

Create a list of Geometry objects holding df attributes corresponding to a scan of the potential energy surface by bond angle. The modification of the angle moves only one atom - the atom corresponding to the third element of the atom_ids argument by index.

Parameters

- **atom_ids** (List[int]) – The atoms between which the angle is changed.
- **bond_angles** (List[float]) – A list of bond angles.

Returns

List[Geometry] – A list of Geometry objects corresponding the bond lengths specified in the input.

scan_bond_angle_by_group(*atom_ids*, *bond_angles*, *group*, *units='deg'*)

Scan a bond angle between two different groups while retaining the geometry of each group.

Parameters

- **atom_ids** (List[int]) – Atom indices.
- **bond_angles** (List[float]) – Bond angles to create geometry objects for.
- **group** (str) – Group heading.
- **units** (str, default: "deg") – Units of the angles defined in bond_angles.

Returns

List – A list of geometries corresponding to the description and bond_angles passed in.

scan_bond_length(*atom_ids*, *bond_lengths*)

Construct many Geometry objects corresponding to a scan of a bond length.

Create a list of Geometry objects holding df attributes corresponding to a scan of the potential energy surface by bond length. The bond stretching moves only one atom - the atom corresponding to the second element of the atom_ids argument by index.

Parameters

- **atom_ids** (List[int]) – The atoms between which the bond is stretched.
- **bond_lengths** (List[float]) – A list of bond lengths.

Returns

List[Geometry] – A list of Geometry objects corresponding the bond lengths specified in the input.

scan_bond_length_by_group(*atom_ids*, *bond_lengths*, *group*)

Scan a bond length between two different subgroups while retaining the geometry of each subgroup.

Parameters

- **atom_ids** (List[int]) – Atom indices.
- **bond_lengths** (List[float]) – Bond lengths to create geometry objects for.
- **group** (str) – Group heading.

Returns

List[Geometry] – A list of geometries corresponding to the description and bond_lengths passed in.

scan_dihedral_angle(*atom_ids*, *dihedral_angles*, *units='deg'*)

Construct many geometry objects corresponding to a scan of a dihedral angle.

Create a list of Geometry objects holding df attributes corresponding to a scan of the potential energy surface by dihedral angle. The modification of the angle moves only one atom - the atom corresponding to the fourth element of the atom_ids argument by index.

Parameters

- **atom_ids** – The atoms between which the dihedral is changed.
- **dihedral_angles** – A list of dihedral angles.

Returns

List[Geometry] – A list of Geometry objects corresponding the bond lengths specified in the input.

scan_dihedral_angle_by_group(*atom_ids*, *dihedral_angles*, *group*, *units='deg'*)

Scan a dihedral angle between atoms in different groups while retaining the geometry of each group.

Parameters

- **atom_ids** (List[int]) – Atom indices.
- **dihedral_angles** (List[float]) – Dihedral angles to create geometry objects for.
- **group** (str) – Group heading.
- **units** – Units of te angles defined in dihedral_angles.

Returns

List – A list of geometries corresponding to the description and dihedral angles passed in.

set_groups(*target*, *source*, *mapping=None*)

Set group information for the atoms in the system.

Target specifies the heading of the new column created in the underlying dataframe. The source argument can be either a dictionary, where the keys are the labels of the group and the values are the atom indices corresponding to the label, or a string. If source is a dictionary, the mapping argument does nothing. If source is a string, it must be an existing column heading in the dataframe. In which case, the mapping argument must be provided as a callable function which modifies the existing entries in the source column and inserts them into the new target column.

Parameters

- **target** (str) – Column in the table that refer to this grouping.
- **source** (str) – Another column or a dictionary defining the groups.
- **mapping** (Optional[Callable], default: None) – Optional mapping to transform group names.

Return type

None

set_subgroups(*target*, *pattern*, *subgroup*)

Set subgroups on the target grouping via regular expression pattern.

Parameters

- **target** (str) – Target column in the table, a table that has a grouping in it.
- **pattern** (str) – Regex pattern to match.
- **subgroup** (str) – Adding subgroups to the matching groups.

Return type

None

to_angstrom()

Convert the geometry in the Geometry.df attribute from Bohrs to Angstroms.

Returns
DataFrame – The updated dataframe object.

to_bohr()
Convert the geometry in the Geometry.df attribute from Angstroms to Bohrs.

Returns
DataFrame – The updated dataframe object.

translate_by_vector(*v*)
Translate all atoms in the geometry held in the Geometry.df attribute by a vector, v.

Parameters
v (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – Vector defining the translation.

Returns
DataFrame – The updated dataframe object.

property xyz: List[List[str | List[float]]]
Return the geometry in xyz format.

Returns
List[List[Union[str, List[float]]]] – Geometry as a list of lists of atom symbols and positions

xyz_to_df(*xyz*)
Convert the xyz geometry to a dataframe and overwrites the current Geometry.df attribute accordingly.

Parameters
xyz (List[Any]) – An xyz geometry as a list of lists containing the element as a string and the position as a list of floats.

Returns
DataFrame – The DataFrame corresponding to the xyz argument.

22.8 inquanto.mappings

class QubitMapping
Bases: ABC

Generic class which performs a mapping from fermions to qubits.

This class forms the base for specific mapping strategies , e.g. the Jordan-Wigner or Bravyi-Kitaev transformation. It may also be used to generate custom mappings, provided they follow the "sets of qubits" formalism of Seeley, Richard & Love (arXiv:1208.5986). Subclasses must implement the following, with more detail descriptions given in this class' docstrings:

Required methods:

- flip_set(): the flip qubit set for a given qubit.
- update_set(): the update qubit set for a given qubit.
- parity_set(): the parity qubit set for a given qubit.
- rho_set(): the rho qubit set for a given qubit.
- state_map_matrix(): a matrix which maps binary representations of fermionic orbital indices to qubit indices.

Required attributes:

- _MAPPING_FLAGS: internal flags for characterising mappings. If creating a custom mapping, this should be set to ["ambiguous_qubit_number"] for maximal genericism.

OPERATOR_MAP_TYPES

A dictionary which maps input types to output types - e.g. OPERATOR_MAP_TYPES[FermionOperator] = QubitOperator.

_MAPPING_FLAGS = ['']

Internal flags for manipulating how mapping is performed. If creating a custom mapping, set this to ["ambiguous_qubit_number"].

classmethod flip_set(*cls*, *qubit*, *qubits=None*)

Return the flip set for orbital i.

The flip set is defined as the set of qubits whose parity determines whether a fermionic creation/annihilation operator acting on orbital i is flipped to an annihilation/creation operator.

Parameters

- **qubit** (Union[Qubit, int]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Union[List[Qubit], int, None], default: None) –

Returns

List[Qubit] – List of Qubits comprising the flip set for orbital i.

classmethod operator_map(*operator*, *qubits=None*, *abs_tol=1e-10*)

Map a fermionic operator or set of operators to qubits.

This method functions differently based on the input type:

- FermionOperator and FermionOperatorString will be mapped to QubitOperator and QubitOperatorString respectively. This is the recommended usage.
- Subclasses of FermionOperator and FermionOperatorString will be mapped to their corresponding qubit equivalents.
- FermionOperatorList will be mapped to QubitOperatorList with each term in the list mapped independently.
- BaseChemistryIntegralOperator and its subclasses will be mapped to QubitOperator assuming that the underlying integral arrays correspond to quadratic and quartic creation/annihilation operator products (i.e. a Hamiltonian).
- Lists, ndarrays and more general iterables will be recursed through and returned in the input type.
- QubitOperator and QubitOperatorString are returned trivially for compatibility.

Note that the attribute OPERATOR_MAP_TYPES of this class contains a map of input types to output types for this method.

Parameters

- **operator** (Union[*FermionOperator*, *FermionOperatorString*, *FermionOperatorList*, *QubitOperator*, *QubitOperatorString*, BaseChemistryIntegralOperator, List, ndarray, Iterator[*FermionOperator*]]) – An object representing a fermionic operator or set of operators, with type behaviour specified above.

- **qubits** (Union[List[Qubit], int, None], default: None) – A list of qubits containing the register to be mapped to or an int giving the size of the register. Qubits do not need to be in order. Optional (minimum necessary register indexed 0-len(qubits) will be assumed) but may be needed for some mappings.
- **abs_tol** (Optional[float], default: 1e-10) – Removes terms in the output operator with magnitude lower than this, to avoid floating point errors resulting in erroneous Pauli strings.

Returns

Union[*QubitOperator*, *QubitOperatorList*, List, ndarray] – The mapped operator or set of operators in the output format described above.

classmethod parity_set(*cls*, *qubit*, *qubits=None*)

Return the parity set for orbital i.

The parity set is the set of qubits which must be acted on by a Pauli Z in the first term of the mapped operator (where qubit i is acted on by Pauli X), when mapping a creation or annihilation operator acting on orbital i.

Parameters

- **qubit** (Union[Qubit, int]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Union[List[Qubit], int, None], default: None) –

Returns

List[Qubit] – List of Qubits comprising the parity set for orbital i.

classmethod rho_set(*cls*, *qubit*, *qubits=None*)

Return the rho set for orbital i.

The rho set is the set of qubits which must be acted on by a Pauli Z in the second term of the mapped operator (where qubit i is acted on by Pauli Y), when mapping a creation or annihilation operator acting on orbital i.

Parameters

- **qubit** (Union[Qubit, int]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Union[List[Qubit], int, None], default: None) –

Returns

List[Qubit] – List of Qubits comprising the rho set for orbital i.

classmethod state_map(*state*, *qubits=None*)

Map a fermionic state to a qubit state vector.

This method functions differently based on the input type:

- FermionState and FermionStateString will be mapped to QubitState and QubitStateString respectively. This is the recommended usage.
- Scipy sparse arrays will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned in the input type.
- Numpy arrays containing scalars will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned as a numpy array. If containing objects other than scalars, the method will attempt to recurse.
- Lists will be treated as iterables and the method will recurse. If this is not possible, it will be assumed that the list contains bools giving occupation numbers for a single basis state, and the corresponding qubit basis state will be returned as a QubitState.

Warning: Currently this uses a conventional mapping (e.g. for Jordan-Wigner, fermionic Fock basis states are mapped directly to qubit computational basis states). This may incur phase flips versus generating states by sequences of mapped creation operators depending on intended ordering convention.

Parameters

- **state** (Union[*FermionState*, *FermionStateString*, List, ndarray, spmatrix]) – The fermionic state to be mapped.
- **qubits** (Union[List[Qubit], int, None], default: None) – A list of qubits containing the register to be mapped to or an int giving the size of the register. Qubits do not need to be in order. Optional (minimum necessary register indexed 0-len(qubits) will be assumed) but may be needed for some mappings.

Returns

Union[*QubitState*, *QubitStateString*, ndarray, spmatrix] – The mapped state in the type described above.

classmethod state_map_conventional(*cls*, *state*, *qubits=None*)

Map a fermionic state to a qubit state vector.

This method functions differently based on the input type:

- FermionState and FermionStateString will be mapped to QubitState and QubitStateString respectively. This is recommended usage.
- Scipy sparse arrays will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned in the input type.
- numpy arrays containing scalars will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned as a numpy array. If containing objects other than scalars, the method will attempt to recurse
- lists will be treated as iterables and the method will recurse. If this is not possible, it will be assumed that the list contains bools giving occupation numbers for a single basis state, and the corresponding qubit basis state will be returned as a QubitState.

Warning: This uses a conventional mapping (e.g. for Jordan-Wigner, fermionic Fock basis states are mapped. directly to qubit computational basis states). This may incur phase flips versus generating states by sequences of mapped creation operators depending on intended ordering convention.

Parameters

- **state** (Union[*FermionState*, *FermionStateString*, List, ndarray, spmatrix]) – The fermionic state to be mapped.
- **qubits** (Union[List[Qubit], int, None], default: None) – A list of qubits containing the register to be mapped to or an int giving the size of the register. Qubits do not need to be in order. Optional (minimum necessary register indexed 0-len(qubits) will be assumed) but may be needed for some mappings.

Returns

Union[*QubitState*, ndarray, spmatrix] – The mapped state in the type described above.

classmethod state_map_matrix(*dimension*)

Generate a state map matrix.

The state map matrix is a matrix which maps the binary representation of a fermionic basis state index to the binary representation of the corresponding qubit basis state index.

Parameters

dimension (int) – The dimension of the matrix (i.e. the number of orbitals/qubits).

Returns

ndarray – Numpy array giving the mapping matrix.

classmethod update_set(*cls*, *qubit*, *qubits=None*)

Return the update set for orbital i.

The update set is defined as the qubits which must be updated (i.e. acted on by a Pauli X when a fermionic creation or annihilation operator acts on orbital i. Note this does not include qubit i, which is treated independently.

Parameters

- **qubit** (Union[Qubit, int]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Union[List[Qubit], int, None], default: None) –

Returns

List[Qubit] – List of Qubits comprising the update set for orbital i.

class QubitMappingJordanWigner

Bases: *QubitMapping*

Class to map states and operators in a Fermionic space to states and operators in a qubit space using the Jordan-Wigner transformation.

OPERATOR_MAP_TYPES

A dictionary which maps input types to output types - e.g. OPERATOR_MAP_TYPES[FermionOperator] = QubitOperator.

_MAPPING_FLAGS = ['']

Internal flags for manipulating how mapping is performed. If creating a custom mapping, set this to ["ambiguous_qubit_number"].

classmethod flip_set(*qubit*, *qubits=None*)

Return the flip set for orbital i.

The flip set is defined as the set of qubits whose parity determines whether a fermionic creation/annihilation operator acting on orbital i is flipped to an annihilation/creation operator.

Parameters

- **qubit** (Union[int, Qubit]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Optional[List[Qubit]], default: None) –

Returns

List[Qubit] – List of Qubits comprising the flip set for orbital i.

classmethod operator_map(*operator*, *qubits=None*, *abs_tol=1e-10*)

Map a fermionic operator or set of operators to qubits.

This method functions differently based on the input type:

- FermionOperator and FermionOperatorString will be mapped to QubitOperator and QubitOperatorString respectively. This is the recommended usage.
- Subclasses of FermionOperator and FermionOperatorString will be mapped to their corresponding qubit equivalents.
- FermionOperatorList will be mapped to QubitOperatorList with each term in the list mapped independently.
- BaseChemistryIntegralOperator and its subclasses will be mapped to QubitOperator assuming that the underlying integral arrays correspond to quadratic and quartic creation/annihilation operator products (i.e. a Hamiltonian).
- Lists, ndarrays and more general iterables will be recursed through and returned in the input type.
- QubitOperator and QubitOperatorString are returned trivially for compatibility.

Note that the attribute OPERATOR_MAP_TYPES of this class contains a map of input types to output types for this method.

Parameters

- **operator** (Union[*FermionOperator*, *FermionOperatorString*, *FermionOperatorList*, *QubitOperator*, *QubitOperatorString*, BaseChemistryIntegralOperator, List, ndarray, Iterator[*FermionOperator*]]) – An object representing a fermionic operator or set of operators, with type behaviour specified above.
- **qubits** (Union[List[Qubit], int, None], default: None) – A list of qubits containing the register to be mapped to or an int giving the size of the register. Qubits do not need to be in order. Optional (minimum necessary register indexed 0-len(qubits) will be assumed) but may be needed for some mappings.
- **abs_tol** (Optional[float], default: 1e-10) – Removes terms in the output operator with magnitude lower than this, to avoid floating point errors resulting in erroneous Pauli strings.

Returns

Union[*QubitOperator*, *QubitOperatorList*, List, ndarray] – The mapped operator or set of operators in the output format described above.

classmethod parity_set(*cls*, *qubit*, *qubits=None*)

Return the parity set for orbital i.

The parity set is the set of qubits which must be acted on by a Pauli Z in the first term of the mapped operator (where qubit i is acted on by Pauli X), when mapping a creation or annihilation operator acting on orbital i.

Parameters

- **qubit** (Union[int, Qubit]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Optional[List[Qubit]], default: None) –

Returns

List[Qubit] – List of Qubits comprising the parity set for orbital i.

classmethod rho_set(*cls*, *qubit*, *qubits=None*)

Return the rho set for orbital i.

The rho set is the set of qubits which must be acted on by a Pauli Z in the second term of the mapped operator (where qubit i is acted on by Pauli Y), when mapping a creation or annihilation operator acting on orbital i.

Parameters

- **qubit** (Union[int, Qubit]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Optional[List[Qubit]], default: None) –

Returns

List[Qubit] – List of Qubits comprising the rho set for orbital i.

classmethod state_map(*state*, *qubits=None*)

Map a fermionic state to a qubit state vector.

This method functions differently based on the input type:

- FermionState and FermionStateString will be mapped to QubitState and QubitStateString respectively. This is the recommended usage.
- Scipy sparse arrays will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned in the input type.
- Numpy arrays containing scalars will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned as a numpy array. If containing objects other than scalars, the method will attempt to recurse.
- Lists will be treated as iterables and the method will recurse. If this is not possible, it will be assumed that the list contains bools giving occupation numbers for a single basis state, and the corresponding qubit basis state will be returned as a QubitState.

Warning: Currently this uses a conventional mapping (e.g. for Jordan-Wigner, fermionic Fock basis states are mapped directly to qubit computational basis states). This may incur phase flips versus generating states by sequences of mapped creation operators depending on intended ordering convention.

Parameters

- **state** (Union[*FermionState*, *FermionStateString*, List, ndarray, spmatrix]) – The fermionic state to be mapped.
- **qubits** (Union[List[Qubit], int, None], default: None) – A list of qubits containing the register to be mapped to or an int giving the size of the register. Qubits do not need to be in order. Optional (minimum necessary register indexed 0-len(qubits) will be assumed) but may be needed for some mappings.

Returns

Union[*QubitState*, *QubitStateString*, ndarray, spmatrix] – The mapped state in the type described above.

classmethod state_map_conventional(*cls*, *state*, *qubits=None*)

Map a fermionic state to a qubit state vector.

This method functions differently based on the input type:

- FermionState and FermionStateString will be mapped to QubitState and QubitStateString respectively. This is recommended usage.
- Scipy sparse arrays will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned in the input type.

- numpy arrays containing scalars will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned as a numpy array. If containing objects other than scalars, the method will attempt to recurse
- lists will be treated as iterables and the method will recurse. If this is not possible, it will be assumed that the list contains bools giving occupation numbers for a single basis state, and the corresponding qubit basis state will be returned as a QubitState.

Warning: This uses a conventional mapping (e.g. for Jordan-Wigner, fermionic Fock basis states are mapped. directly to qubit computational basis states). This may incur phase flips versus generating states by sequences of mapped creation operators depending on intended ordering convention.

Parameters

- **state** (Union[*FermionState*, *FermionStateString*, List, ndarray, spmatrix]) – The fermionic state to be mapped.
- **qubits** (Union[List[Qubit], int, None], default: None) – A list of qubits containing the register to be mapped to or an int giving the size of the register. Qubits do not need to be in order. Optional (minimum necessary register indexed 0-len(qubits) will be assumed) but may be needed for some mappings.

Returns

Union[*QubitState*, ndarray, spmatrix] – The mapped state in the type described above.

static state_map_matrix(*dimension*)

Generate a state map matrix.

The state map matrix is a matrix which maps the binary representation of a fermionic basis state index to the binary representation of the corresponding qubit basis state index.

Parameters

dimension (int) – The dimension of the matrix (i.e. the number of orbitals/qubits).

Returns

Numpy array giving the mapping matrix.

classmethod update_set(*qubit*, *qubits=None*)

Return the update set for orbital i.

The update set is defined as the qubits which must be updated (i.e. acted on by a Pauli X when a fermionic creation or annihilation operator acts on orbital i. Note this does not include qubit i, which is treated independently.

Parameters

- **qubit** (Union[int, Qubit]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Optional[List[Qubit]], default: None) –

Returns

List[Qubit] – List of Qubits comprising the update set for orbital i.

class QubitMappingBravyiKitaev

Bases: *QubitMapping*

Class to map states and operators in a Fermionic space to states and operators in a qubit space using the Bravyi-Kitaev mapping.

The Bravyi-Kitaev mapping encodes both parity and occupation number information in a manner wherein it may be accessed with a logarithmic number of operations with respect to the system size. For further detail, see S. Bravyi, A. Kitaev, Ann. Phys. 298, 210 (2002).

OPERATOR_MAP_TYPES

A dictionary which maps input types to output types - e.g. OPERATOR_MAP_TYPES[FermionOperator] = QubitOperator.

_MAPPING_FLAGS = ['ambiguous_qubit_number']

Internal flags for manipulating how mapping is performed. If creating a custom mapping, set this to ["ambiguous_qubit_number"].

classmethod flip_set(*cls*, *qubit*, *qubits=None*)

Return the flip set for orbital i.

The flip set is defined as the set of qubits whose parity determines whether a fermionic creation/annihilation operator acting on orbital i is flipped to an annihilation/creation operator.

Parameters

- **qubit** (Union[Qubit, int]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Union[List[Qubit], int, None], default: None) –

Returns

List[Qubit] – List of Qubits comprising the flip set for orbital i.

classmethod operator_map(*cls*, *operator*, *qubits=None*, *abs_tol=1e-10*)

Map a fermionic operator or set of operators to qubits.

This method functions differently based on the input type:

- FermionOperator and FermionOperatorString will be mapped to QubitOperator and QubitOperatorString respectively. This is the recommended usage.
- Subclasses of FermionOperator and FermionOperatorString will be mapped to their corresponding qubit equivalents.
- FermionOperatorList will be mapped to QubitOperatorList with each term in the list mapped independently.
- BaseChemistryIntegralOperator and its subclasses will be mapped to QubitOperator assuming that the underlying integral arrays correspond to quadratic and quartic creation/annihilation operator products (i.e. a Hamiltonian).
- Lists, ndarrays and more general iterables will be recursed through and returned in the input type.
- QubitOperator and QubitOperatorString are returned trivially for compatibility.

Note that the attribute OPERATOR_MAP_TYPES of this class contains a map of input types to output types for this method.

Parameters

- **operator** (Union[*FermionOperator*, *FermionOperatorString*, *FermionOperatorList*, *QubitOperator*, *QubitOperatorString*, BaseChemistryIntegralOperator, List, ndarray]) – An object representing a fermionic operator or set of operators, with type behaviour specified above.

- **qubits** (Union[List[Qubit], int, None], default: None) – A list of qubits containing the register to be mapped to or an int giving the size of the register. Qubits do not need to be in order. Optional (minimum necessary register indexed 0-len(qubits) will be assumed) but may be needed for some mappings.
- **abs_tol** (Optional[float], default: 1e-10) – Removes terms in the output operator with magnitude lower than this, to avoid floating point errors resulting in erroneous Pauli strings.

Returns
The mapped operator or set of operators in the output format described above.

classmethod parity_set(*cls*, *qubit*, *qubits=None*)

Return the parity set for orbital i.

The parity set is the set of qubits which must be acted on by a Pauli Z in the first term of the mapped operator (where qubit i is acted on by Pauli X), when mapping a creation or annihilation operator acting on orbital i.

Parameters

- **qubit** (Union[Qubit, int]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Union[List[Qubit], int, None], default: None) –

Returns
List[Qubit] – List of Qubits comprising the parity set for orbital i.

classmethod remainder_set(*cls*, *qubit*, *qubits=None*)

Return the remainder set (the set difference of the parity set and the flip set).

Parameters

- **qubit** (Union[Qubit, int]) –
- **qubits** (Union[List[Qubit], int, None], default: None) –

Return type
List[Qubit]

classmethod rho_set(*cls*, *qubit*, *qubits=None*)

Return the rho set for orbital i.

The rho set is the set of qubits which must be acted on by a Pauli Z in the second term of the mapped operator (where qubit i is acted on by Pauli Y), when mapping a creation or annihilation operator acting on orbital i.

Parameters

- **qubit** (Union[Qubit, int]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Union[List[Qubit], int, None], default: None) –

Returns
List[Qubit] – List of Qubits comprising the rho set for orbital i.

classmethod state_map(*state*, *qubits=None*)

Map a fermionic state to a qubit state vector.

This method functions differently based on the input type:

- FermionState and FermionStateString will be mapped to QubitState and QubitStateString respectively. This is the recommended usage.

- Scipy sparse arrays will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned in the input type.
- Numpy arrays containing scalars will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned as a numpy array. If containing objects other than scalars, the method will attempt to recurse.
- Lists will be treated as iterables and the method will recurse. If this is not possible, it will be assumed that the list contains bools giving occupation numbers for a single basis state, and the corresponding qubit basis state will be returned as a QubitState.

Warning: Currently this uses a conventional mapping (e.g. for Jordan-Wigner, fermionic Fock basis states are mapped directly to qubit computational basis states). This may incur phase flips versus generating states by sequences of mapped creation operators depending on intended ordering convention.

Parameters

- **state** (Union[*FermionState*, *FermionStateString*, List, ndarray, spmatrix]) – The fermionic state to be mapped.
- **qubits** (Union[List[Qubit], int, None], default: None) – A list of qubits containing the register to be mapped to or an int giving the size of the register. Qubits do not need to be in order. Optional (minimum necessary register indexed 0-len(qubits) will be assumed) but may be needed for some mappings.

Returns

Union[*QubitState*, *QubitStateString*, ndarray, spmatrix] – The mapped state in the type described above.

classmethod state_map_conventional(*cls*, *state*, *qubits=None*)

Map a fermionic state to a qubit state vector.

This method functions differently based on the input type:

- FermionState and FermionStateString will be mapped to QubitState and QubitStateString respectively. This is recommended usage.
- Scipy sparse arrays will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned in the input type.
- numpy arrays containing scalars will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned as a numpy array. If containing objects other than scalars, the method will attempt to recurse
- lists will be treated as iterables and the method will recurse. If this is not possible, it will be assumed that the list contains bools giving occupation numbers for a single basis state, and the corresponding qubit basis state will be returned as a QubitState.

Warning: This uses a conventional mapping (e.g. for Jordan-Wigner, fermionic Fock basis states are mapped. directly to qubit computational basis states). This may incur phase flips versus generating states by sequences of mapped creation operators depending on intended ordering convention.

Parameters

- **state** (Union[*FermionState*, *FermionStateString*, List, ndarray, spmatrix]) – The fermionic state to be mapped.

- **qubits** (Union[List[Qubit], int, None], default: None) – A list of qubits containing the register to be mapped to or an int giving the size of the register. Qubits do not need to be in order. Optional (minimum necessary register indexed 0-len(qubits) will be assumed) but may be needed for some mappings.

Returns
Union[*QubitState*, ndarray, spmatrix] – The mapped state in the type described above.

classmethod state_map_matrix(*dimension*)

Generate a state map matrix.

The state map matrix is a matrix which maps the binary representation of a fermionic basis state index to the binary representation of the corresponding qubit basis state index.

Parameters
dimension (int) – The dimension of the matrix (i.e. the number of orbitals/qubits).

Returns
Numpy array giving the mapping matrix.

classmethod update_set(*cls*, *qubit*, *qubits=None*)

Return the update set for orbital i.

The update set is defined as the qubits which must be updated (i.e. acted on by a Pauli X when a fermionic creation or annihilation operator acts on orbital i. Note this does not include qubit i, which is treated independently.

Parameters

- **qubit** (Union[Qubit, int]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Union[List[Qubit], int, None], default: None) –

Returns
List[Qubit] – List of Qubits comprising the update set for orbital i.

class QubitMappingParity

Bases: *QubitMapping*

Class to map states and operators in a Fermionic space to states and operators in a qubit space using the parity mapping.

OPERATOR_MAP_TYPES

A dictionary which maps input types to output types - e.g. OPERATOR_MAP_TYPES[FermionOperator] = QubitOperator.

_MAPPING_FLAGS = ['ambiguous_qubit_number']

Internal flags for manipulating how mapping is performed. If creating a custom mapping, set this to ["ambiguous_qubit_number"].

classmethod flip_set(*cls*, *qubit*, *qubits=None*)

Return the flip set for orbital i.

The flip set is defined as the set of qubits whose parity determines whether a fermionic creation/annihilation operator acting on orbital i is flipped to an annihilation/creation operator.

Parameters

- **qubit** (Union[Qubit, int]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.

- **qubits** (Union[List[Qubit], int, None], default: None) –

Returns

List[Qubit] – List of Qubits comprising the flip set for orbital i.

classmethod operator_map(*cls*, *operator*, *qubits=None*, *abs_tol=1e-10*)

Map a fermionic operator or set of operators to qubits.

This method functions differently based on the input type:

- FermionOperator and FermionOperatorString will be mapped to QubitOperator and QubitOperatorString respectively. This is the recommended usage.
- Subclasses of FermionOperator and FermionOperatorString will be mapped to their corresponding qubit equivalents.
- FermionOperatorList will be mapped to QubitOperatorList with each term in the list mapped independently.
- BaseChemistryIntegralOperator and its subclasses will be mapped to QubitOperator assuming that the underlying integral arrays correspond to quadratic and quartic creation/annihilation operator products (i.e. a Hamiltonian).
- Lists, ndarrays and more general iterables will be recursed through and returned in the input type.
- QubitOperator and QubitOperatorString are returned trivially for compatibility.

Note that the attribute OPERATOR_MAP_TYPES of this class contains a map of input types to output types for this method.

Parameters

- **operator** (Union[*FermionOperator*, *FermionOperatorString*, *FermionOperatorList*, *QubitOperator*, *QubitOperatorString*, BaseChemistryIntegralOperator, List, ndarray]) – An object representing a fermionic operator or set of operators, with type behaviour specified above.
- **qubits** (Union[List[Qubit], int, None], default: None) – A list of qubits containing the register to be mapped to or an int giving the size of the register. Qubits do not need to be in order. Optional (minimum necessary register indexed 0-len(qubits) will be assumed) but may be needed for some mappings.
- **abs_tol** (Optional[float], default: 1e-10) – Removes terms in the output operator with magnitude lower than this, to avoid floating point errors resulting in erroneous Pauli strings.

Returns

The mapped operator or set of operators in the output format described above.

classmethod parity_set(*cls*, *qubit*, *qubits=None*)

Return the parity set for orbital i.

The parity set is the set of qubits which must be acted on by a Pauli Z in the first term of the mapped operator (where qubit i is acted on by Pauli X), when mapping a creation or annihilation operator acting on orbital i.

Parameters

- **qubit** (Union[Qubit, int]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Union[List[Qubit], int, None], default: None) –

Returns

List[Qubit] – List of Qubits comprising the parity set for orbital i.

classmethod rho_set(*qubit*, *qubits=None*)

Return the rho set for orbital i.

The rho set is the set of qubits which must be acted on by a Pauli Z in the second term of the mapped operator (where qubit i is acted on by Pauli Y), when mapping a creation or annihilation operator acting on orbital i.

Parameters

- **qubit** (Union[Qubit, int]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Union[List[Qubit], int, None], default: None) –

Returns

List[Qubit] – List of Qubits comprising the rho set for orbital i.

classmethod state_map(*state*, *qubits=None*)

Map a fermionic state to a qubit state vector.

This method functions differently based on the input type:

- FermionState and FermionStateString will be mapped to QubitState and QubitStateString respectively. This is the recommended usage.
- Scipy sparse arrays will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned in the input type.
- Numpy arrays containing scalars will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned as a numpy array. If containing objects other than scalars, the method will attempt to recurse.
- Lists will be treated as iterables and the method will recurse. If this is not possible, it will be assumed that the list contains bools giving occupation numbers for a single basis state, and the corresponding qubit basis state will be returned as a QubitState.

Warning: Currently this uses a conventional mapping (e.g. for Jordan-Wigner, fermionic Fock basis states are mapped directly to qubit computational basis states). This may incur phase flips versus generating states by sequences of mapped creation operators depending on intended ordering convention.

Parameters

- **state** (Union[*FermionState*, *FermionStateString*, List, ndarray, spmatrix]) – The fermionic state to be mapped.
- **qubits** (Union[List[Qubit], int, None], default: None) – A list of qubits containing the register to be mapped to or an int giving the size of the register. Qubits do not need to be in order. Optional (minimum necessary register indexed 0-len(qubits) will be assumed) but may be needed for some mappings.

Returns

Union[*QubitState*, *QubitStateString*, ndarray, spmatrix] – The mapped state in the type described above.

classmethod state_map_conventional(*cls*, *state*, *qubits=None*)

Map a fermionic state to a qubit state vector.

This method functions differently based on the input type:

- FermionState and FermionStateString will be mapped to QubitState and QubitStateString respectively. This is recommended usage.

- Scipy sparse arrays will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned in the input type.
- numpy arrays containing scalars will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned as a numpy array. If containing objects other than scalars, the method will attempt to recurse
- lists will be treated as iterables and the method will recurse. If this is not possible, it will be assumed that the list contains bools giving occupation numbers for a single basis state, and the corresponding qubit basis state will be returned as a QubitState.

Warning: This uses a conventional mapping (e.g. for Jordan-Wigner, fermionic Fock basis states are mapped. directly to qubit computational basis states). This may incur phase flips versus generating states by sequences of mapped creation operators depending on intended ordering convention.

Parameters

- **state** (Union[*FermionState*, *FermionStateString*, List, ndarray, spmatrix]) – The fermionic state to be mapped.
- **qubits** (Union[List[Qubit], int, None], default: None) – A list of qubits containing the register to be mapped to or an int giving the size of the register. Qubits do not need to be in order. Optional (minimum necessary register indexed 0-len(qubits) will be assumed) but may be needed for some mappings.

Returns

Union[*QubitState*, ndarray, spmatrix] – The mapped state in the type described above.

static state_map_matrix(*dimension*)

Generate a state map matrix.

The state map matrix is a matrix which maps the binary representation of a fermionic basis state index to the binary representation of the corresponding qubit basis state index.

Parameters

dimension (int) – The dimension of the matrix (i.e. the number of orbitals/qubits).

Returns

ndarray – Numpy array giving the mapping matrix.

classmethod update_set(*cls*, *qubit*, *qubits=None*)

Return the update set for orbital i.

The update set is defined as the qubits which must be updated (i.e. acted on by a Pauli X when a fermionic creation or annihilation operator acts on orbital i. Note this does not include qubit i, which is treated independently.

Parameters

- **qubit** (Union[Qubit, int]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Union[List[Qubit], int, None], default: None) –

Returns

List[Qubit] – List of Qubits comprising the update set for orbital i.

class QubitMappingParaparticular

Bases: *QubitMapping*

Class to map fermions to qubits a paraparticular mapping - i.e. without encoding Fermionic statistics.

This mapping does not encode the fermionic anticommutation relations. Fermionic operators and states are treated as though they were operators and states on a Hilbert space of paraparticles – i.e. particles which obey bosonic commutation relations, but are explicitly restricted to an occupation number of 0 or 1. Qubits can be considered to be paraparticles, so all this mapping does is convert a creation/annihilation operator (σ^+, σ^-) decomposition to a Pauli decomposition.

Warning: This will lead to erroneous results if used as a drop-in replacement for conventional fermion-qubit mapping schemes. The intended use-case for this mapping is for instances where the lack of encoded fermionic anticommutation relations is harmless due to their encoding elsewhere in the problem - for instance, when creating an ansatz used in a variational optimisation (arXiv:2101.11607).

OPERATOR_MAP_TYPES

A dictionary which maps input types to output types - e.g. OPERATOR_MAP_TYPES[FermionOperator] = QubitOperator.

_MAPPING_FLAGS = ['']

Internal flags for manipulating how mapping is performed. If creating a custom mapping, set this to ["ambiguous_qubit_number"].

classmethod flip_set(*qubit*, *qubits=None*)

Return the flip set for orbital i.

The flip set is defined as the set of qubits whose parity determines whether a fermionic creation/annihilation operator acting on orbital i is flipped to an annihilation/creation operator.

Parameters

- **qubit** (Union[int, Qubit]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Optional[List[Qubit]], default: None) –

Returns

List[Qubit] – List of Qubits comprising the flip set for orbital i.

classmethod operator_map(*operator*, *qubits=None*, *abs_tol=1e-10*)

Map a fermionic operator or set of operators to qubits.

This method functions differently based on the input type:

- FermionOperator and FermionOperatorString will be mapped to QubitOperator and QubitOperatorString respectively. This is the recommended usage.
- Subclasses of FermionOperator and FermionOperatorString will be mapped to their corresponding qubit equivalents.
- FermionOperatorList will be mapped to QubitOperatorList with each term in the list mapped independently.
- BaseChemistryIntegralOperator and its subclasses will be mapped to QubitOperator assuming that the underlying integral arrays correspond to quadratic and quartic creation/annihilation operator products (i.e. a Hamiltonian).
- Lists, ndarrays and more general iterables will be recursed through and returned in the input type.

- QubitOperator and QubitOperatorString are returned trivially for compatibility.

Note that the attribute OPERATOR_MAP_TYPES of this class contains a map of input types to output types for this method.

Parameters

- **operator** (Union[*FermionOperator*, *FermionOperatorString*, *FermionOperatorList*, *QubitOperator*, *QubitOperatorString*, BaseChemistryIntegralOperator, List, ndarray, Iterator[*FermionOperator*]]) – An object representing a fermionic operator or set of operators, with type behaviour specified above.
- **qubits** (Union[List[Qubit], int, None], default: None) – A list of qubits containing the register to be mapped to or an int giving the size of the register. Qubits do not need to be in order. Optional (minimum necessary register indexed 0-len(qubits) will be assumed) but may be needed for some mappings.
- **abs_tol** (Optional[float], default: 1e-10) – Removes terms in the output operator with magnitude lower than this, to avoid floating point errors resulting in erroneous Pauli strings.

Returns

Union[*QubitOperator*, *QubitOperatorList*, List, ndarray] – The mapped operator or set of operators in the output format described above.

classmethod parity_set(*qubit*, *qubits=None*)

Return the parity set for orbital i.

The parity set is the set of qubits which must be acted on by a Pauli Z in the first term of the mapped operator (where qubit i is acted on by Pauli X), when mapping a creation or annihilation operator acting on orbital i.

Parameters

- **qubit** (Union[int, Qubit]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Optional[List[Qubit]], default: None) –

Returns

List[Qubit] – List of Qubits comprising the parity set for orbital i.

classmethod rho_set(*qubit*, *qubits=None*)

Return the rho set for orbital i.

The rho set is the set of qubits which must be acted on by a Pauli Z in the second term of the mapped operator (where qubit i is acted on by Pauli Y), when mapping a creation or annihilation operator acting on orbital i.

Parameters

- **qubit** (Union[int, Qubit]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Optional[List[Qubit]], default: None) –

Returns

List[Qubit] – List of Qubits comprising the rho set for orbital i.

classmethod state_map(*state*, *qubits=None*)

Map a fermionic state to a qubit state vector.

This method functions differently based on the input type:

- FermionState and FermionStateString will be mapped to QubitState and QubitStateString respectively. This is the recommended usage.
- Scipy sparse arrays will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned in the input type.
- Numpy arrays containing scalars will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned as a numpy array. If containing objects other than scalars, the method will attempt to recurse.
- Lists will be treated as iterables and the method will recurse. If this is not possible, it will be assumed that the list contains bools giving occupation numbers for a single basis state, and the corresponding qubit basis state will be returned as a QubitState.

Warning: Currently this uses a conventional mapping (e.g. for Jordan-Wigner, fermionic Fock basis states are mapped directly to qubit computational basis states). This may incur phase flips versus generating states by sequences of mapped creation operators depending on intended ordering convention.

Parameters

- **state** (Union[*FermionState*, *FermionStateString*, List, ndarray, spmatrix]) – The fermionic state to be mapped.
- **qubits** (Union[List[Qubit], int, None], default: None) – A list of qubits containing the register to be mapped to or an int giving the size of the register. Qubits do not need to be in order. Optional (minimum necessary register indexed 0-len(qubits) will be assumed) but may be needed for some mappings.

Returns

Union[*QubitState*, *QubitStateString*, ndarray, spmatrix] – The mapped state in the type described above.

classmethod state_map_conventional(*cls*, *state*, *qubits=None*)

Map a fermionic state to a qubit state vector.

This method functions differently based on the input type:

- FermionState and FermionStateString will be mapped to QubitState and QubitStateString respectively. This is recommended usage.
- Scipy sparse arrays will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned in the input type.
- numpy arrays containing scalars will be assumed to be a fermionic state vector, and the corresponding qubit state vector will be returned as a numpy array. If containing objects other than scalars, the method will attempt to recurse
- lists will be treated as iterables and the method will recurse. If this is not possible, it will be assumed that the list contains bools giving occupation numbers for a single basis state, and the corresponding qubit basis state will be returned as a QubitState.

Warning: This uses a conventional mapping (e.g. for Jordan-Wigner, fermionic Fock basis states are mapped. directly to qubit computational basis states). This may incur phase flips versus generating states by sequences of mapped creation operators depending on intended ordering convention.

Parameters

- **state** (Union[*FermionState*, *FermionStateString*, List, ndarray, spmatrix]) – The fermionic state to be mapped.
- **qubits** (Union[List[Qubit], int, None], default: None) – A list of qubits containing the register to be mapped to or an int giving the size of the register. Qubits do not need to be in order. Optional (minimum necessary register indexed 0-len(qubits) will be assumed) but may be needed for some mappings.

Returns
Union[*QubitState*, ndarray, spmatrix] – The mapped state in the type described above.

static state_map_matrix(*dimension*)

Generate a state map matrix.

The state map matrix is a matrix which maps the binary representation of a fermionic basis state index to the binary representation of the corresponding qubit basis state index.

Parameters
dimension (int) – The dimension of the matrix (i.e. the number of orbitals/qubits).

Returns
Numpy array giving the mapping matrix.

classmethod update_set(*qubit*, *qubits=None*)

Return the update set for orbital i.

The update set is defined as the qubits which must be updated (i.e. acted on by a Pauli X when a fermionic creation or annihilation operator acts on orbital i. Note this does not include qubit i, which is treated independently.

Parameters
- **qubit** (Union[int, Qubit]) – Index i of fermionic orbital being operated on, or a Qubit sharing the index.
- **qubits** (Optional[List[Qubit]], default: None) –

Returns
List[Qubit] – List of Qubits comprising the update set for orbital i.

22.9 inquanto.minimizers

Module provides access to minimizers compatible with variational experiments.

class MinimizerRotosolve(*max_iterations=20*, *tolerance=1e-4*, *disp=False*)

Bases: GeneralMinimizer

The Rotosolve minimizer, introduced in Quantum 5, 391 (2021).

This learns the minimum of an estimator with a sinusoidal energy landscape.

Parameters
- **max_iterations** (int, default: 20) – Maximum number of iterations allowed before the minimization is terminated.
- **tolerance** (float, default: 1e-4) – Tolerance for convergence.
- **disp** (bool, default: False) – If True, print information to the screen throughout minimization.

generate_report()

Generate a summary of the minimization.

Returns

A dict containing the number of iterations (n_iterations), the final value (final_value) and final parameters (final_parameters).

minimize(*function*, *initial*)

Minimize the function provided.

Minimization starts at the the parameters provided by the initial argument.

Parameters

- **function** (Callable[[ndarray], float]) – Sinusoidal objective function to minimize.
- **initial** (ndarray) – Initial parameters to optimize.

Returns

Tuple[float, ndarray] – A tuple containing the final value and parameters obtained by the minimization.

class MinimizerSGD(*learning_rate=0.01*, *decay_rate=0.05*, *max_iterations=100*, *disp=False*, *callback=None*)

Bases: GeneralMinimizer

Uses the gradient and geometry of the objective function to accelerate minimization.

Introduced in Quantum 4, 269 (2020).

Parameters

- **learning_rate** (float, default: 0.01) – Stepsize in direction of descent.
- **decay_rate** (float, default: 0.05) – User defined decay rate.
- **max_iterations** (int, default: 100) – Maximum number of iterations allowed before variational loop is terminated.
- **disp** (bool, default: False) – If True, display minimization history.
- **callback** (Optional[Callable], default: None) – Custom callback for minimizer.

generate_report()

Generate a report summarizing the minimization.

Includes the final value, the final parameters and the number of iterations peformed.

Returns

A dictionary containing the final value, parameters and number of iterations obtained by the minimizer.

minimize(*function*, *initial*, *gradient*)

Minimize the objective function, starting at the initial parameters provided by the user.

Parameters

- **function** (Callable[[Union[ndarray, List[float]]], float]) – Objective function to minimize.
- **initial** (Union[ndarray, List[float]]) – Initial parameters to optimize.
- **gradient** (Callable[[Union[ndarray, List[float]]], Union[ndarray, List[float]]]) – Gradient function to assist minimization.

Returns
Tuple[float, Union[ndarray, List[float]]] – A tuple containing the final value and parameters obtained by the minimizer.

class MinimizerScipy(*method='L-BFGS-B'*, *options=None*, *disp=False*, *callback=None*)

Bases: GeneralMinimizer

A simple wrapper for Scipy minimization routines.

Parameters

- **method** (str, default: "L-BFGS-B") – The method to use. Popular methods to choose from include "CG", "BFGS", "L-BFGS-B", and "SLSQP".
- **options** (Optional[dict], default: None) – Options used for calibration which are passed through to the scipy minimization.
- **disp** (bool, default: False) – Set to True to print minimization history.
- **callback** (Optional[Callable], default: None) – Custom callback function.

generate_report()

Generate a report containing a summary of the minimization.

Returns
Dict – A dictionary containing the final value and location of the final value from the minimization.

property method: str

Get the method being used by the optimizer as a string.

Returns
str – The name of the minimization algorithm used by the minimizer.

minimize(*function*, *initial*, *gradient=None*)

Minimize the function provided.

The minimization starts at the parameters provided in the initial argument, and the gradient (if provided) is used to aid the minimization and is evaluated by calling the gradient argument.

Parameters

- **function** (Callable[[ndarray], float]) – The objective function to minimize.
- **initial** (ndarray) – Initial parameters to optimize.
- **gradient** (Optional[Callable[[ndarray], ndarray]], default: None) – Gradient function to assist minimization.

Returns
Tuple[float, ndarray] – The value of the function at the minimum and the location of the minimum.

property options: Dict

Get the options passed to scipy by the minimizer internally.

Returns
Dict – A dict containing the options used by scipy to perform the minimization.

class NaiveEulerIntegrator(*time_eval*, *disp=False*, *callback=None*, *linear_solver=GeneralIntegrator.linear_solver_scipy_linalg*)

Bases: GeneralIntegrator

A simple Euler integrator to solve time evolution problems.

Parameters

- **time_eval** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) –
- **disp** (bool, default: False) –
- **callback** (Optional[Callable[[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], float, ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]], Any]], default: None) –
- **linear_solver** (Callable[[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]], default: GeneralIntegrator.linear_solver_scipy_linalg) –

static linear_solver_scipy_linalg(*a*, *b*)

Wraps scipy.linalg.solve method.

Parameters

- **a** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) –
- **b** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) –

Return type
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]

static linear_solver_scipy_pinvh(*a*, *b*)

Wraps scipy.linalg.solve method.

Parameters

- **a** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) –
- **b** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) –

Return type
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]

solve(*linear_problem*, *initial*, **args*, ***kwargs*)

Solve the differential equation.

Parameters

- **linear_problem** (Callable[[Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]], float], Tuple[Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]], Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]]]]) – A function f(p, t) -> A,b which takes the parameters and time and returns the linear problem (matrix A(t), vector b(t) of A*x=b) at a time.

- **initial** (Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]]) – Initial parameters.

Returns

Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – The solution to the differential equation.

class ScipyIVPIntegrator(*time_eval*, *disp=False*, *callback=None*, *linear_solver=GeneralIntegrator.linear_solver_scipy_linalg*)

Bases: GeneralIntegrator

Wrapper class for scipy solve_ivp for linear problems.

Parameters

- **time_eval** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) –
- **disp** (bool, default: False) –
- **callback** (Optional[Callable[[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], float, ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]], Any]], default: None) –
- **linear_solver** (Callable[[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]], default: GeneralIntegrator.linear_solver_scipy_linalg) –

static linear_solver_scipy_linalg(*a*, *b*)

Wraps scipy.linalg.solve method.

Parameters

- **a** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) –
- **b** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) –

Return type

ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]

static linear_solver_scipy_pinvh(*a*, *b*)

Wraps scipy.linalg.solve method.

Parameters

- **a** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) –
- **b** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) –

Return type

ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]

solve(*linear_problem*, *initial*, **args*, ***kwargs*)

Solve the differential equation.

Parameters

- **linear_problem** (Callable[[Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]], float], Tuple[Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]], Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]]]]) – A function f(p, t) -> A,b which takes the parameters and time and returns the linear problem (matrix A(t), vector b(t) of A*x=b) at a time.
- **initial** (Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]]) – Initial parameters.

Returns

Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – The solution to the differential equation.

class ScipyODEIntegrator(*time_eval*, *disp=False*, *callback=None*, *linear_solver=GeneralIntegrator.linear_solver_scipy_linalg*)

Bases: GeneralIntegrator

Wrapper class for scipy ODEINT for linear problems.

Parameters

- **time_eval** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) –
- **disp** (bool, default: False) –
- **callback** (Optional[Callable[[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], float, ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]], Any]], default: None) –
- **linear_solver** (Callable[[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]], default: GeneralIntegrator.linear_solver_scipy_linalg) –

static linear_solver_scipy_linalg(*a*, *b*)

Wraps scipy.linalg.solve method.

Parameters

- **a** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) –
- **b** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) –

Return type
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]

static linear_solver_scipy_pinvh(*a*, *b*)

Wraps scipy.linalg.solve method.

Parameters

- **a** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) –
- **b** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) –

Return type
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]

solve(*linear_problem*, *initial*, **args*, ***kwargs*)

Solve the differential equation.

Parameters

- **linear_problem** (Callable[[Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]], float], Tuple[Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]], Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]]]]) – A function f(p, t) -> A,b which takes the parameters and time and returns the linear problem (matrix A(t), vector b(t) of A*x=b) at a time.
- **initial** (Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]]) – Initial parameters.

Returns
Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – The solution to the differential equation.

22.10 inquanto.operators

InQuanto representation of quantum operators.

class ChemistryRestrictedIntegralOperator(*constant*, *one_body*, *two_body*, *dtype=None*)

Bases: BaseChemistryIntegralOperator

Handles a (restricted-orbital) chemistry integral operator.

Stores constant, one- and two-body spatial integral values following chemistry notation, two_body[p, q, r, s] = (pq|rs).

Parameters

- **constant** (float) – Constant energy term, electron independent.

- **one_body** (ndarray) – One-body energy integrals.
- **two_body** (ndarray) – Two-body electron repulsion integrals (ERI).

TOLERANCE = 1e-08

Internal tolerance used when iterating over terms. Terms with magnitude smaller than this are not returned by *items()*.

approx_equal(*other*, *rtol=1.0e-5*, *atol=1.0e-8*, *equal_nan=False*)

Test for approximate equality with another instance of this class by comparing integral arrays.

Arguments are passed directly to numpy.allclose for equality comparison - see numpy documentation for further details.

Parameters

- **other** (*ChemistryRestrictedIntegralOperator*) – The other operator to compare for approximate equality.
- **rtol** (float, default: 1.0e-5) – The relative tolerance parameter, as defined by numpy.allclose.
- **atol** (float, default: 1.0e-8) – The absolute tolerance parameter, as defined by numpy.allclose.
- **equal_nan** – Whether to compare NaN's as equal, as defined by numpy.allclose.

Returns

bool – True if other instance is approximately equal to this, otherwise False.

astype(*dtype*)

Returns a copy of the current integral operator, cast to a new dtype.

Parameters

dtype (Any) – The dtype to cast into.

Returns

ChemistryRestrictedIntegralOperator – New integral operator with components cast to dtype.

copy()

Performs a deep copy of object.

Return type

BaseChemistryIntegralOperator

df()

Returns a pandas.DataFrame object showing all terms in the operator.

Return type

DataFrame

double_factorize(*tol1=-1*, *tol2=None*, *diagonalize_one_body=True*, *diagonalize_one_body_offset=True*, *combine_one_body_terms=True*)

Double factorizes the two-electron integrals and returns the hamiltonian in diagonal form.

Writes the hamiltonian as $H = H_0 + H_1 + S + V$ where $S + V$ is the coulomb interaction. $V = (1/2)\sum_{ijkl}(ij|kl)a_i^\dagger a_j a_k^\dagger a_l$ is a reordered two-body operator and S is a one-body-like energy offset given by $S = -(1/2)\sum_{ij} s_{ij} a_i^\dagger a_j$ where $s_{ij} = \sum_k (ik|kj)$. H_0 and H_1 are the constant and one-electron terms respectively.

First level of factorization is an eigenvalue decomposition of the ERI tensor: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. The second factorization is diagonalization of the V_{pq}^t matrix for each t: $V_{pq}^t = \sum_u^{N_\lambda^t} U_{pu}^t \lambda_u^t U_{qu}^t$. At each diagonalization truncation is performed by discarding eigenvalues, starting from the smallest, until the sum of those discarded exceeds the threshold.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} - s_{pq}/2 = \sum_r W_{pr} \omega_r W_{qr}$. One-body diagonalization is not truncated.

Warning: Not intended for reduction of classical memory usage, only for truncating the two-body terms of the hamiltonian for quantum simulation.

Parameters

- **tol1** (float, default: -1) – Truncation threshold for first diagonalization of ERI matrix. If negative, no truncation is performed.
- **tol2** (Optional[float], default: None) – Truncation threshold for second diagonalization of ERI matrix. If None, same as tol1. If negative, no truncation is performed.
- **diagonalize_one_body** (Optional[bool], default: True) – Whether to diagonalize the physical one-body integrals h_pq.
- **diagonalize_one_body_offset** (Optional[bool], default: True) – Whether to diagonalize the one-body offset integrals s_{pq}
- **combine_one_body_terms** (Optional[bool], default: True) – Whether to consolidate the one-body and one-body offset integrals into effective one-body integrals. Requires diagonalize_one_body == diagonalize_one_body_offset.

Returns

DoubleFactorizedHamiltonian –:

Hamiltonian operator storing two-body integrals in double factorized form and, optionally, diagonalised
one-body integrals.

property dtype: dtype

Returns numpy data type of the two-body integral terms.

Return type
dtype

effective_potential(*rdm1*)

Calculates the effective Coulombic potential for a given 1-RDM.

Parameters
rdm1 (*RestrictedOneBodyRDM*) – Restricted, one-body reduced density matrix object.

Returns
ndarray – Effective potential matrix.

effective_potential_spin(*rdm1*)

Calculates the contribution to the effective Coulomb potential due to a spin imbalance.

RDM1a - RDM1b contribution to the effective potential matrix.

Parameters
rdm1 (*RestrictedOneBodyRDM*) – Restricted, one-body reduced density matrix object.

Returns
ndarray – Effective potential matrix.

energy(*rdm1*, *rdm2=None*)

Calculate total energy based on the one- and two-body reduced density matrices.

If rdm2 is not given, this method returns the mean-field energy.

Parameters

- **rdm1** (*RestrictedOneBodyRDM*) – Restricted, one-body reduced density matrix object.
- **rdm2** (Optional[*RestrictedTwoBodyRDM*], default: None) – Restricted, two-body reduced density matrix object.

Returns
float – Total energy.

energy_electron_mean_field(*rdm1*)

Calculates the electronic energy in the mean-field approximation.

Parameters
rdm1 (*RestrictedOneBodyRDM*) – Restricted, one-body reduced density matrix object.

Returns
Tuple[float, float] – Mean-field electronic energy (1e + 2e), and Mean-field Coulomb contribution (2e).

classmethod from_FermionOperator(*operator*, *dtype=float*)

Convert a 2-body fermionic operator to restricted-orbital integral operator.

The one-body terms are stored in a matrix, _one_body[p, q], and the two-body terms are stored in a tensor, _two_body[p, q, r, s].

Parameters

- **operator** (*FermionOperator*) – Input fermion operator.
- **dtype** – Desired term value dtype.

Returns
ChemistryRestrictedIntegralOperator – Output integral operator.

> **Warning:** Currently, this converter assumes all the terms in the integral operator are represented by the aabb and bbaa blocks of fermion operator terms, and they are supposed to overwrite the (contracted) aaaa and bbbb terms - it might give faulty result if this is not the case

static from_fcidump(*filename*)

Generate a ChemistryRestrictedIntegralOperator object from an FCIDUMP file.

FCIDUMP files typically contain information regarding the molecular orbital integrals, number of orbitals, number of electrons, spin and spatial symmetry. Only the symmetry information is discarded by this method.

Parameters
filename (str) – The FCIDUMP filename.

Returns
operator – The operator generated from the FCIDUMP file. n_orb: The number of orbitals in the system. n_elec: The number of electrons in the system. multiplicity: The spin multiplicity of the system.

property imag: *ChemistryRestrictedIntegralOperator*

Extract the imaginary part of the integral operator.

Returns
ChemistryRestrictedIntegralOperator – New integral operator object with only the imaginary parts of all elements remaining.

items (*yield_constant=True*, *yield_one_body=True*, *yield_two_body=True*)

Generates the constant, one- and two-body operator terms contained in the operator object.

Parameters

- **yield_constant** (bool, default: True) – Whether to generate a constant term.
- **yield_one_body** (bool, default: True) – Whether to generate one-body terms.
- **yield_two_body** (bool, default: True) – Whether to generate two-body terms.

Yields
Next requested *FermionOperatorString* and constant/integral value.

Return type
Generator[Tuple[*FermionOperatorString*, float], None, None]

classmethod load_h5 (*name*)

Loads operator object from .h5 file.

Parameters
name (Union[str, Group]) – Name of .h5 file to be loaded.

Returns
BaseChemistryIntegralOperator – Loaded integral operator object.

norm (**args*, ***kwargs*)

Calculates the norm of the integral operator.

Sums the norm of the constant, one-body and two-body parts using numpy.linalg.norm().

Parameters

- ***args** – Additional arguments passed to numpy.linalg.norm().
- ****kwargs** – Additional keyword arguments passed to numpy.linalg.norm().

Returns
float – Norm of integral operator.

print_table ()

Prints operator terms in a table format.

Return type
None

qubit_encode (*mapping=None*)

Performs qubit encoding (mapping) using the provided mapping class of the current integral operator.

Parameters
mapping (Optional[QubitMapping], default: None) – Mapping function. Default mapping procedure is Jordan-Wigner.

Returns
Union[QubitOperator, QubitOperatorList, List, ndarray] – Mapped *QubitOperator* object.

property real: *ChemistryRestrictedIntegralOperator*

Extract the real part of the integral operator.

Returns
ChemistryRestrictedIntegralOperator – New integral operator object with only the real parts of all elements remaining.

rotate(*rotation*)

Performs an in-place unitary rotation of the chemistry integrals.

Parameters
rotation (ndarray) – Unitary rotation matrix.

Returns
ChemistryRestrictedIntegralOperator – self after rotation.

save_h5(*name*)

Dumps operator object to .h5 file.

Parameters
name (Union[str, Group]) – Destination filename of .h5 file.

Return type
None

to_FermionOperator(*yield_constant=True*, *yield_one_body=True*, *yield_two_body=True*)

Converts chemistry integral operator to *FermionOperator*.

Returns
FermionOperator – Fermion operator form of integral operator.

Parameters

- **yield_constant** (bool, default: True) –
- **yield_one_body** (bool, default: True) –
- **yield_two_body** (bool, default: True) –

to_compact_integral_operator(*symmetry*)

Converts two-body integrals into compact form.

Compacts the two-body integrals into a CompactTwoBodyIntegrals object.

Returns
ChemistryRestrictedIntegralOperatorCompact – Equivalent integral operator with two-body integrals stored in compact form.

Parameters
symmetry (Union[int, str]) –

two_body_iijj()

Returns pair-diagonal elements of two-body integrals, (ii|jj) in chemist's notation.

Returns
ndarray – 2D array of pair-diagonal two body-integrals.

class ChemistryRestrictedIntegralOperatorCompact(*constant*, *one_body*, *two_body*, *dtype=None*)

Bases: BaseChemistryIntegralOperator

Handles a (restricted-orbital) chemistry integral operator, with integrals stored in a compact form.

Stores constant, one- and two-body spatial integral values following chemistry notation, i.e. `two_body[p, q, r, s]` = (pqlrs). Two-body integrals are stored as a `CompactTwoBodyIntegrals` object rather than a `numpy.ndarray`, which reduces memory load, at the expense of some operations taking longer.

Parameters

- **constant** (`float`) – Constant energy term, electron independent.
- **one_body** (`ndarray`) – One-body energy integrals.
- **two_body** (`Union[CompactTwoBodyIntegralsS4, CompactTwoBodyIntegralsS8]`) – Two-body electron repulsion integrals (ERI).

TOLERANCE = 1e-08

Internal tolerance used when iterating over terms. Terms with magnitude smaller than this are not returned by *items()*.

approx_equal(*other*, *rtol=1.0e-5*, *atol=1.0e-8*, *equal_nan=False*)

Test for approximate equality with another instance of this class by comparing integral arrays.

Arguments are passed directly to `numpy.allclose` for equality comparison - see numpy documentation for further details.

Parameters

- **other** (*ChemistryRestrictedIntegralOperatorCompact*) – The other operator to compare for approximate equality.
- **rtol** (`float`, default: `1.0e-5`) – The relative tolerance parameter, as defined by `numpy.allclose`.
- **atol** (`float`, default: `1.0e-8`) – The absolute tolerance parameter, as defined by `numpy.allclose`.
- **equal_nan** (`bool`, default: `False`) – Whether to compare NaN's as equal, as defined by `numpy.allclose`.

Returns

`bool` – True if other instance is approximately equal to this, otherwise False.

astype(*dtype*)

Returns a copy of the current integral operator, cast to a new dtype.

Parameters

dtype (`Any`) – The dtype to cast into.

Returns

ChemistryRestrictedIntegralOperatorCompact – New integral operator with components cast to dtype.

copy()

Performs a deep copy of object.

Return type

`BaseChemistryIntegralOperator`

df()

Returns a `pandas.DataFrame` object showing all terms in the operator.

Return type

`DataFrame`

double_factorize(*tol1=-1*, *tol2=None*, *diagonalize_one_body=True*, *diagonalize_one_body_offset=True*, *combine_one_body_terms=True*)

Double factorizes the two-electron integrals and returns the hamiltonian in diagonal form.

Writes the hamiltonian as $H = H_0 + H_1 + S + V$ where $S + V$ is the coulomb interaction. $V = (1/2)\sum_{ijkl}(ij|kl)a_i^\dagger a_j a_k^\dagger a_l$ is a reordered two-body operator and S is a one-body-like energy offset given by $S = -(1/2)\sum_{ij} s_{ij} a_i^\dagger a_j$ where $s_{ij} = \sum_k (ik|kj)$. H_0 and H_1 are the constant and one-electron terms respectively.

First level of factorization is an eigenvalue decomposition of the ERI tensor: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. The second factorization is diagonalization of the V_{pq}^t matrix for each t: $V_{pq}^t = \sum_u^{N_\lambda^t} U_{pu}^t \lambda_u^t U_{qu}^t$. At each diagonalization truncation is performed by discarding eigenvalues, starting from the smallest, until the sum of those discarded exceeds the threshold.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} - s_{pq}/2 = \sum_r W_{pr}\omega_r W_{qr}$. One-body diagonalization is not truncated.

Warning: Not intended for reduction of classical memory usage, only for truncating the two-body terms of the hamiltonian for quantum simulation.

Parameters

- **tol1** (`float`, default: `-1`) – Truncation threshold for first diagonalization of ERI matrix. If negative, no truncation is performed.
- **tol2** (`Optional[float]`, default: `None`) – Truncation threshold for second diagonalization of ERI matrix. If `None`, same as `tol1`. If negative, no truncation is performed.
- **diagonalize_one_body** (`Optional[bool]`, default: `True`) – Whether to diagonalize the physical one-body integrals h_pq.
- **diagonalize_one_body_offset** (`Optional[bool]`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq}
- **combine_one_body_terms** (`Optional[bool]`, default: `True`) – Whether to consolidate the one-body and one-body offset integrals into effective one-body integrals. Requires `diagonalize_one_body == diagonalize_one_body_offset`.

Returns

DoubleFactorizedHamiltonian –:

Hamiltonian operator storing two-body integrals in double factorized form and, optionally, diagonalised
one-body integrals.

property dtype: dtype

Returns numpy data type of the two-body integral terms.

Return type
`dtype`

effective_potential(*rdm1*)

Calculates the effective Coulombic potential for a given 1-RDM.

Parameters
rdm1 (*RestrictedOneBodyRDM*) – Restricted, one-body reduced density matrix object.

Returns
ndarray – Effective potential matrix.

effective_potential_spin(*rdm1*)

Calculates the contribution to the effective Coulomb potential due to a spin imbalance.

RDM1a - RDM1b contribution to the effective potential matrix.

Parameters
rdm1 (*RestrictedOneBodyRDM*) – Restricted, one-body reduced density matrix object.

Returns
ndarray – Effective potential matrix.

energy(*rdm1*, *rdm2=None*)

Calculate total energy based on the one- and two-body reduced density matrices.

If rdm2 is not given, this method returns the mean-field energy.

Parameters

- **rdm1** (*RestrictedOneBodyRDM*) – Restricted, one-body reduced density matrix object.
- **rdm2** (Optional[*RestrictedTwoBodyRDM*], default: None) – Restricted, two-body reduced density matrix object.

Returns
float – Total energy.

energy_electron_mean_field(*rdm1*)

Calculates the electronic energy in the mean-field approximation.

Parameters
rdm1 (*RestrictedOneBodyRDM*) – Restricted, one-body reduced density matrix object.

Returns
Tuple[float, float] – Mean-field electronic energy (1e + 2e), and Mean-field Coulomb contribution (2e).

classmethod from_FermionOperator(*operator*, *dtype=float*, *symmetry='s4'*)

Convert a 2-body fermionic operator to compact, restricted-orbital integral operator.

The one-body terms are stored in a matrix, _one_body[p, q], and the two-body terms are stored in a CompactTwoBodyIntegrals object, _two_body[p, q, r, s]

Parameters

- **operator** (*FermionOperator*) – Input fermion operator.
- **dtype** – Desired term value dtype.

Returns
Output integral operator.

Warning: Currently, this converter assumes all the terms in the integral operator are represented by the aabb and bbaa blocks of fermion operator terms, and they are supposed to overwrite the (contracted) aaaa and bbbb terms - it might give faulty results if this is not the case.

Parameters
symmetry (Union[str, int], default: "s4") –

property imag

Extract the imaginary part of the integral operator.

Returns
New integral operator object with only the imaginary parts of all elements remaining.

items(*yield_constant=True*, *yield_one_body=True*, *yield_two_body=True*)

Generates the constant, one- and two-body operator terms contained in the operator object.

Parameters

- **yield_constant** (bool, default: True) – Whether to generate a constant term.
- **yield_one_body** (bool, default: True) – Whether to generate one-body terms.
- **yield_two_body** (bool, default: True) – Whether to generate two-body terms.

Yields
Next requested *FermionOperatorString* and constant/integral value.

Return type
Generator[Tuple[*FermionOperatorString*, float], None, None]

classmethod load_h5(*name*)

Loads operator object from .h5 file.

Parameters
name (Union[str, Group]) – Name of .h5 file to be loaded.

Returns
BaseChemistryIntegralOperator – Loaded integral operator object.

norm()

Calculates the Frobenius norm of the integral operator.

Sums the norm of the constant, one-body and two-body parts.

Returns
Norm of integral operator.

print_table()

Prints operator terms in a table format.

Return type
None

qubit_encode(*mapping=None*)

Performs qubit encoding (mapping) using the provided mapping class of the current integral operator.

Parameters
mapping (Optional[QubitMapping], default: None) – Mapping function. Default mapping procedure is Jordan-Wigner.

Returns
Union[QubitOperator, QubitOperatorList, List, ndarray] – Mapped *QubitOperator* object.

property real

Extract the real part of the integral operator.

Returns
New integral operator object with only the real parts of all elements remaining.

rotate(*rotation*)

Performs an in-place unitary rotation of the chemistry integrals.

Rotation must be real-valued (orthogonal) for compact integrals.

Raises
ValueError – If dimensions of rotation matrix are not compatible with integrals.

Parameters
rotation (ndarray) – Real, unitary rotation matrix.

Returns
ChemistryRestrictedIntegralOperatorCompact – self after rotation.

save_h5(*name*)

Dumps operator object to .h5 file.

Parameters
name (Union[str, Group]) – Destination filename of .h5 file.

Return type
None

to_FermionOperator(*yield_constant=True*, *yield_one_body=True*, *yield_two_body=True*)

Converts chemistry integral operator to *FermionOperator*.

Returns
FermionOperator – Fermion operator form of integral operator.

Parameters

- **yield_constant** (bool, default: True) –
- **yield_one_body** (bool, default: True) –
- **yield_two_body** (bool, default: True) –

to_uncompacted_integral_operator()

Convert to a *ChemistryRestrictedIntegralOperator* object.

Unpacks the compact two-body integrals into a four-dimensional numpy.ndarray.

Returns
Equivalent integral operator object with un-compacted integrals.

two_body_iijj()

Returns pair-diagonal elements of two-body integrals, (ii|jj) in chemist's notation.

Returns
2D array of pair-diagonal two body-integrals.

class ChemistryUnrestrictedIntegralOperator(*constant*, *one_body_aa*, *one_body_bb*, *two_body_aaaa*, *two_body_bbbb*, *two_body_aabb*, *two_body_bbaa*)

Bases: BaseChemistryIntegralOperator

Handles a (unrestricted-orbital) chemistry integral operator.

Stores constant, one- and two-body spatial integral values following chemistry notation, i.e. two_body[p, q, r, s] = (pq|rs)

Parameters

- **constant** (float) – Constant energy term, electron independent.

- **one_body_aa** (ndarray) – One-body energy integrals for the alpha (a) spin channel.
- **one_body_bb** (ndarray) – One-body energy integrals for the beta (b) spin channel.
- **two_body_aaaa** (ndarray) – Two-body electron repulsion integrals (ERI) with spatial orbitals in the aaaa spin channels respectively.
- **two_body_bbbb** (ndarray) – ERI with spatial orbitals in the bbbb spin channels.
- **two_body_aabb** (ndarray) – ERI with spatial orbitals in the aabb spin channels.
- **two_body_bbaa** (ndarray) – ERI with spatial orbitals in the bbaa spin channels.

TOLERANCE = 1e-08

Internal tolerance used when iterating over terms. Terms with magnitude smaller than this are not returned by *items()*.

approx_equal(*other*, *rtol=1.0e-5*, *atol=1.0e-8*, *equal_nan=False*)

Test for approximate equality with another instance of this class by comparing integral arrays.

Arguments are passed directly to `numpy.allclose` for equality comparison - see numpy documentation for further details.

Parameters

- **other** (*ChemistryUnrestrictedIntegralOperator*) – The other operator to compare for approximate equality.
- **rtol** (float, default: 1.0e-5) – The relative tolerance parameter, as defined by numpy.allclose.
- **atol** (float, default: 1.0e-8) – The absolute tolerance parameter, as defined by numpy.allclose.
- **equal_nan** (bool, default: False) – Whether to compare NaN's as equal, as defined by numpy.allclose.

Returns

bool – True if other instance is approximately equal to this, otherwise False.

copy()

Performs a deep copy of object.

Return type

BaseChemistryIntegralOperator

df()

Returns a `pandas.DataFrame` object showing all terms in the operator.

Return type

DataFrame

double_factorize(*tol1=-1*, *tol2=None*, *diagonalize_one_body=True*, *diagonalize_one_body_offset=True*, *combine_one_body_terms=True*)

Double factorizes the two-electron integrals and returns the hamiltonian in diagonal form.

Writes the hamiltonian as $H = H_0 + H_1 + S + V$ where $S + V$ is the coulomb interaction. $V = (1/2)\sum_{ijkl}(ij|kl)a_i^\dagger a_j a_k^\dagger a_l$ is a reordered two-body operator and S is a one-body-like energy offset given by $S = -(1/2)\sum_{ij} s_{ij} a_i^\dagger a_j$ where $s_{ij} = \sum_k (ik|kj)$. H_0 and H_1 are the constant and one-electron terms respectively.

First level of factorization is an eigenvalue decomposition of the ERI tensor: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. The second factorization is diagonalization of the V_{pq}^t matrix for each t: $V_{pq}^t = \sum_u^{N_\lambda^t} U_{pu}^t \lambda_u^t U_{qu}^t$. At each

diagonalization truncation is performed by discarding eigenvalues, starting from the smallest, until the sum of those discarded exceeds the threshold.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} - s_{pq}/2 = \sum_r W_{pr}\omega_r W_{qr}$. One-body diagonalization is not truncated.

Warning: Not intended for reduction of classical memory usage, only for truncating the two-body terms of the hamiltonian for quantum simulation.

Parameters

- **tol1** (`float`, default: `-1`) – Truncation threshold for first diagonalization of ERI matrix. If negative, no truncation is performed.
- **tol2** (`Optional[float]`, default: `None`) – Truncation threshold for second diagonalization of ERI matrix. If `None`, same as `tol1`. If negative, no truncation is performed.
- **diagonalize_one_body** (`Optional[bool]`, default: `True`) – Whether to diagonalize the physical one-body integrals h_pq.
- **diagonalize_one_body_offset** (`Optional[bool]`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq}
- **combine_one_body_terms** (`Optional[bool]`, default: `True`) – Whether to consolidate the one-body and one-body offset integrals into effective one-body integrals. Requires `diagonalize_one_body == diagonalize_one_body_offset`.

Returns

DoubleFactorizedHamiltonian –:

Hamiltonian operator storing two-body integrals in double factorized form and, optionally, diagonalised
one-body integrals.

effective_potential(*rdm1*)

Calculates the effective Coulombic potential for a given 1-RDM.

Parameters
rdm1 (*UnrestrictedOneBodyRDM*) – Unrestricted, one-body reduced density matrix.

Returns
`List[ndarray]` – Effective potentials for the alpha and beta spin channels.

energy(*rdm1*, *rdm2=None*)

Calculate total energy based on the one- and two-body reduced density matrices.

If `rdm2` is not given, this method returns the mean-field energy.

Parameters

- **rdm1** (*UnrestrictedOneBodyRDM*) – Unrestricted, one-body reduced density matrix object.
- **rdm2** (`Optional[`*UnrestrictedTwoBodyRDM*`]`, default: `None`) – Unrestricted, two-body reduced density matrix object.

Returns
`float` – Total energy.

energy_electron_mean_field(*rdm1*)

Calculates the electronic energy in the mean-field approximation.

Parameters
rdm1 (*UnrestrictedOneBodyRDM*) – Unrestricted, one-body reduced density matrix object.

Returns
Tuple[float, float] – Mean-field electronic energy (1e + 2e), and Mean-field Coulomb contribution (2e).

classmethod from_FermionOperator(*operator*)

Convert a 2-body fermionic operator to unrestricted-orbital integral operator.

The one-body terms are stored in matrices, `_one_body_aa[p, q]` and `_one_body_bb`, and the two-body terms are stored in tensors, `_two_body_aaaa[p, q, r, s]`, `_two_body_bbbb[...]`, `_two_body_aabb[...]` and `_two_body_bbaa[...]`.

Parameters
operator (*FermionOperator*) – Input fermion operator.

Returns
ChemistryUnrestrictedIntegralOperator – Output integral operator.

Warning: The same-spin (aaaa and bbbb) blocks returned are contracted - this is fine if you use them in quantum chemistry calculations with density matrices having unit elements (case of Slater Determinants) - for other purposes one should be careful.

items(*yield_constant=True*, *yield_one_body=True*, *yield_two_body=True*)

Generates the constant, one- and two-body operator terms contained in the operator object.

Parameters

- **yield_constant** (bool, default: True) – Whether to generate a constant term.
- **yield_one_body** (bool, default: True) – Whether to generate one-body terms.
- **yield_two_body** (bool, default: True) – Whether to generate two-body terms.

Yields
Next requested *FermionOperatorString* and constant/integral value.

Return type
Generator[Tuple[*FermionOperatorString*, float], None, None]

classmethod load_h5(*name*)

Loads operator object from .h5 file.

Parameters
name (Union[str, Group]) – Name of .h5 file to be loaded.

Returns
BaseChemistryIntegralOperator – Loaded integral operator object.

print_table()

Prints operator terms in a table format.

Return type
None

qubit_encode (*mapping=None*)

Performs qubit encoding (mapping) using the provided mapping class of the current integral operator.

Parameters

mapping (Optional[QubitMapping], default: None) – Mapping function. Default mapping procedure is Jordan-Wigner.

Returns

Union[QubitOperator, QubitOperatorList, List, ndarray] – Mapped *QubitOperator* object.

rotate (*rotation_aa*, *rotation_bb*)

Performs an in-place unitary rotation of the chemistry integrals.

Each spin block is rotated separately.

Parameters

- **rotation_aa** (ndarray) – Unitary rotation matrix for the alpha spin block.
- **rotation_bb** (ndarray) – Unitary rotation matrix for the beta spin block.

Returns

ChemistryUnrestrictedIntegralOperator – self after rotation.

save_h5 (*name*)

Dumps operator object to .h5 file.

Parameters

name (Union[str, Group]) – Destination filename of .h5 file.

Return type

None

to_FermionOperator (*yield_constant=True*, *yield_one_body=True*, *yield_two_body=True*)

Converts chemistry integral operator to *FermionOperator*.

Returns

FermionOperator – Fermion operator form of integral operator.

Parameters

- **yield_constant** (bool, default: True) –
- **yield_one_body** (bool, default: True) –
- **yield_two_body** (bool, default: True) –

to_compact_integral_operator (*symmetry*)

Converts two-body integrals into compact form.

Compacts the two-body integrals into a CompactTwoBodyIntegrals object.

Parameters

symmetry (Union[int, str]) – Target symmetry. Four-fold symmetry (4, "4" or s4) or eight-fold symmetry (8, "8" or s8) are supported.

Returns

ChemistryUnrestrictedIntegralOperatorCompact – Equivalent integral operator with two-body integrals stored in compact form.

class ChemistryUnrestrictedIntegralOperatorCompact(*constant*, *one_body_aa*, *one_body_bb*, *two_body_aaaa*, *two_body_bbbb*, *two_body_aabb*, *two_body_bbaa*, *dtype=None*)

Bases: `BaseChemistryIntegralOperator`

Handles a (unrestricted-orbital) chemistry integral operator, with integrals stored in a compact form.

Stores constant, one- and two-body spatial integral values following chemistry notation, i.e. `two_body[p, q, r, s]` = (pqlrs). Two-body integrals are stored as a `CompactTwoBodyIntegrals` object rather than a `numpy.ndarray`, which reduces memory load, at the expense of some operations taking longer.

Note: The `symmetry` class instance attribute is determined by the symmetry of the `two_body_aaaa` input. The aabb and bbaa integral tensors are incompatible with s8 symmetry.

Parameters

- **constant** (`float`) – Constant energy term, electron independent.
- **one_body_aa** (`ndarray`) – One-body energy integrals for the alpha (a) spin channel.
- **one_body_bb** (`ndarray`) – One-body energy integrals for the beta (b) spin channel.
- **two_body_aaaa** (`Union`[*CompactTwoBodyIntegralsS4*, *CompactTwoBodyIntegralsS8*]) – Two-body electron repulsion integrals (ERI) with spatial orbitals in the aaaa spin channels respectively.
- **two_body_bbbb** (`Union`[*CompactTwoBodyIntegralsS4*, *CompactTwoBodyIntegralsS8*]) – ERI with spatial orbitals in the bbbb spin channels.
- **two_body_aabb** (*CompactTwoBodyIntegralsS4*) – ERI with spatial orbitals in the aabb spin channels.
- **two_body_bbaa** (*CompactTwoBodyIntegralsS4*) – ERI with spatial orbitals in the bbaa spin channels.

TOLERANCE = 1e-08

Internal tolerance used when iterating over terms. Terms with magnitude smaller than this are not returned by *items()*.

approx_equal(*other*, *rtol=1.0e-5*, *atol=1.0e-8*, *equal_nan=False*)

Test for approximate equality with another instance of this class by comparing integral arrays.

Arguments are passed directly to `numpy.allclose` for equality comparison - see numpy documentation for further details.

Parameters

- **other** (*ChemistryUnrestrictedIntegralOperatorCompact*) – The other operator to compare for approximate equality.
- **rtol** (`float`, default: `1.0e-5`) – The relative tolerance parameter, as defined by `numpy.allclose`.
- **atol** (`float`, default: `1.0e-8`) – The absolute tolerance parameter, as defined by `numpy.allclose`.
- **equal_nan** (`bool`, default: `False`) – Whether to compare NaN's as equal, as defined by `numpy.allclose`.

Returns
bool – True if other instance is approximately equal to this, otherwise False.

copy()

Performs a deep copy of object.

Return type
BaseChemistryIntegralOperator

df()

Returns a pandas.DataFrame object showing all terms in the operator.

Return type
DataFrame

double_factorize(*tol1=-1*, *tol2=None*, *diagonalize_one_body=True*, *diagonalize_one_body_offset=True*, *combine_one_body_terms=True*)

Double factorizes the two-electron integrals and returns the hamiltonian in diagonal form.

Writes the hamiltonian as $H = H_0 + H_1 + S + V$ where $S + V$ is the coulomb interaction. $V = (1/2)\sum_{ijkl}(ij|kl)a_i^\dagger a_j a_k^\dagger a_l$ is a reordered two-body operator and S is a one-body-like energy offset given by $S = -(1/2)\sum_{ij} s_{ij} a_i^\dagger a_j$ where $s_{ij} = \sum_k (ik|kj)$. H_0 and H_1 are the constant and one-electron terms respectively.

First level of factorization is an eigenvalue decomposition of the ERI tensor: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. The second factorization is diagonalization of the V_{pq}^t matrix for each t: $V_{pq}^t = \sum_u^{N_\lambda^t} U_{pu}^t \lambda_u^t U_{qu}^t$. At each diagonalization truncation is performed by discarding eigenvalues, starting from the smallest, until the sum of those discarded exceeds the threshold.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} - s_{pq}/2 = \sum_r W_{pr}\omega_r W_{qr}$. One-body diagonalization is not truncated.

Warning: Not intended for reduction of classical memory usage, only for truncating the two-body terms of the hamiltonian for quantum simulation.

Parameters

- **tol1** (float, default: -1) – Truncation threshold for first diagonalization of ERI matrix. If negative, no truncation is performed.
- **tol2** (Optional[float], default: None) – Truncation threshold for second diagonalization of ERI matrix. If None, same as tol1. If negative, no truncation is performed.
- **diagonalize_one_body** (Optional[bool], default: True) – Whether to diagonalize the physical one-body integrals h_pq.
- **diagonalize_one_body_offset** (Optional[bool], default: True) – Whether to diagonalize the one-body offset integrals s_{pq}
- **combine_one_body_terms** (Optional[bool], default: True) – Whether to consolidate the one-body and one-body offset integrals into effective one-body integrals. Requires diagonalize_one_body == diagonalize_one_body_offset.

Returns

DoubleFactorizedHamiltonian –:

Hamiltonian operator storing two-body integrals in double factorized form and, optionally, diagonalised
one-body integrals.

effective_potential(*rdm1*)

Calculates the effective Coulombic potential for a given 1-RDM.

Parameters
rdm1 (*UnrestrictedOneBodyRDM*) – Unrestricted, one-body reduced density matrix.

Returns
List[ndarray] – Effective potentials for the alpha and beta spin channels.

energy(*rdm1*, *rdm2=None*)

Calculate total energy based on the one- and two-body reduced density matrices.

If rdm2 is not given, this method returns the mean-field energy.

Parameters

- **rdm1** (*UnrestrictedOneBodyRDM*) – Unrestricted, one-body reduced density matrix object.
- **rdm2** (Optional[*UnrestrictedTwoBodyRDM*], default: None) – Unrestricted, two-body reduced density matrix object.

Returns
float – Total energy.

energy_electron_mean_field(*rdm1*)

Calculates the electronic energy in the mean-field approximation.

Parameters
rdm1 (*UnrestrictedOneBodyRDM*) – Unrestricted, one-body reduced density matrix object.

Returns
Tuple[float, float] – Mean-field electronic energy (1e + 2e), and Mean-field Coulomb contribution (2e).

classmethod from_FermionOperator(*operator*, *symmetry='s4'*)

Convert a 2-body fermionic operator to unrestricted-orbital integral operator.

The one-body terms are stored in matrices, _one_body_aa[p, q] and _one_body_bb, and the two-body terms are stored in CompactTwoBodyIntegrals objects, _two_body_aaaa[p, q, r, s], _two_body_bbbb[...], _two_body_aabb[...] and _two_body_bbaa[...].

Parameters
operator (*FermionOperator*) – Input fermion operator.

Returns
ChemistryUnrestrictedIntegralOperatorCompact – Output integral operator.

Warning: The same-spin (aaaa and bbbb) blocks returned are contracted - this is fine if you use them in quantum chemistry calculations with density matrices having unit elements (case of Slater Determinants) - for other purposes one should be careful.

Parameters
symmetry (Union[str, int], default: "s4") –

items (*yield_constant=True*, *yield_one_body=True*, *yield_two_body=True*)

Generates the constant, one- and two-body operator terms contained in the operator object.

Parameters

- **yield_constant** (bool, default: True) – Whether to generate a constant term.
- **yield_one_body** (bool, default: True) – Whether to generate one-body terms.
- **yield_two_body** (bool, default: True) – Whether to generate two-body terms.

Yields
Next requested *FermionOperatorString* and constant/integral value.

Return type
Generator[Tuple[*FermionOperatorString*, float], None, None]

classmethod load_h5 (*name*)

Loads operator object from .h5 file.

Parameters
name (Union[str, Group]) – Name of .h5 file to be loaded.

Returns
BaseChemistryIntegralOperator – Loaded integral operator object.

print_table ()

Prints operator terms in a table format.

Return type
None

qubit_encode (*mapping=None*)

Performs qubit encoding (mapping) using the provided mapping class of the current integral operator.

Parameters
mapping (Optional[QubitMapping], default: None) – Mapping function. Default mapping procedure is Jordan-Wigner.

Returns
Union[QubitOperator, QubitOperatorList, List, ndarray] – Mapped *QubitOperator* object.

rotate (*rotation_aa*, *rotation_bb*)

Performs an in-place unitary rotation of the chemistry integrals.

Each spin block is rotated separately. Rotation must be real-valued (orthogonal) for compact integrals.

Parameters

- **rotation_aa** (ndarray) – Real, unitary rotation matrix for the alpha spin basis.
- **rotation_bb** (ndarray) – Real, unitary rotation matrix for the beta spin basis.

Returns
ChemistryUnrestrictedIntegralOperatorCompact – self after rotation.

save_h5 (*name*)

Dumps operator object to .h5 file.

Parameters
name (Union[str, Group]) – Destination filename of .h5 file.

Return type
None

to_FermionOperator(*yield_constant=True*, *yield_one_body=True*, *yield_two_body=True*)

Converts chemistry integral operator to *FermionOperator*.

Returns
FermionOperator – Fermion operator form of integral operator.

Parameters

- **yield_constant** (bool, default: True) –
- **yield_one_body** (bool, default: True) –
- **yield_two_body** (bool, default: True) –

to_uncompacted_integral_operator()

Convert to a *ChemistryUnrestrictedIntegralOperator* object.

Unpacks the compact two-body integrals into a four-dimensional numpy.ndarray.

Returns
ChemistryUnrestrictedIntegralOperator – Equivalent integral operator object with un-compacted integrals.

class CompactTwoBodyIntegralsS4(*compact_array*)

Bases: BaseCompactTwoBodyIntegrals

Stores a two-body integral tensor in s4 symmetry reduced form.

Allows simple, 4-indexed [i,j,k,l] array-like access through index transformation.

Parameters
compact_array (ndarray) – Two-body integrals in a four-fold symmetry-reduced form. A numpy.ndarray with shape (n_pair_ij, n_pair_kl) where n_pair = n_orb * (n+orb + 1) / 2. This array may be rectangular to account for different numbers of alpha/beta spin orbitals in the h_aabb ERI tensor in an unrestricted spin picture.

astype(*dtype*)

Returns a copy of the current compact integrals, with the compact array cast to a new type.

Parameters
dtype (Any) – The dtype to cast into.

Returns
CompactTwoBodyIntegralsS4 – New compact integrals object with compact array cast to dtype.

static check_s4_symmetry(*uncompact_array*, *rtol=1.0e-5*, *atol=1.0e-8*, *equal_nan=False*)

Tests whether the input ERI tensor has four-fold (s4) index symmetries.

Checks for symmetries under index swaps i<->j and k<->l.

Parameters

- **uncompact_array** (ndarray) – Four-dimensional ERI tensor.
- **rtol** (float, default: 1.0e-5) – Relative tolerance.
- **atol** (float, default: 1.0e-8) – Absolute tolerance.

- **equal_nan** (bool, default: False) – Whether to compare NaN's as equal.

Returns
bool – True if the input array has s4 symmetry, false otherwise.

property dtype: dtype

Returns numpy data type of the compact array.

Return type
dtype

classmethod from_uncompacted_integrals(*two_body*, *symmetry*, *check_symmetry=False*, *rtol=1.0e-5*, *atol=1.0e-8*, *equal_nan=False*)

Builds a compact integrals object from an uncompacted set of two body integrals (a rank-4 tensor).

Parameters

- **two_body** (ndarray) – 4D array.
- **symmetry** (Union[str, int]) – Code to specify target symmetry. Uses the same convention as pyscf. Currently supports s4 and s8 symmetry.
- **check_symmetry** (bool, default: False) – Whether the input array should be checked for the requested symmetry.
- **rtol** (float, default: 1.0e-5) – Relative tolerance on symmetry checks.
- **atol** (float, default: 1.0e-8) – Absolute tolerance.
- **equal_nan** (bool, default: False) – Whether to compare NaN's as equal.

Returns
Union[*CompactTwoBodyIntegralsS4*, *CompactTwoBodyIntegralsS8*] – Object containing the same information as the input array in a compact form.

property imag: *CompactTwoBodyIntegralsS4*

Extract the imaginary part of the two-body integrals.

Returns
CompactTwoBodyIntegralsS4 – New compact object containing only the imaginary part of the integrals.

static pairs(*n_orb*)

Yields unique index pairs, and their pair index.

Does not return equivalent pairs i.e. {2, 1} will appear, but {1, 2} will not.

Parameters
n_orb (int) – Number of orbitals over which pairs will be iterated.

Yields
Pair index, first orbital index, second orbital index.

Return type
Iterator[Tuple[int, int, int]]

property real: *CompactTwoBodyIntegralsS4*

Extract the real part of the two-body integrals.

Returns
CompactTwoBodyIntegralsS4 – New compact object containing only the real part of the integrals.

rotate(*u_ij*, *u_kl=None*)

Perform a rotation of the compact two-body integrals.

Parameters

- **u_ij** (ndarray) – A real, orthogonal matrix of dimensions (n_orb_ij, n_orb_ij). For rotating the first two indices of the two-body integrals tensor.
- **u_kl** (Optional[ndarray], default: None) – A real, orthogonal matrix of dimensions (n_orb_kl, n_orb_kl). For rotating the second two indices of the two-body integrals tensor.

Return type
None

property shape: Tuple[int, int, int, int]

The shape of the corresponding rank-4 tensor.

Returns
Tuple[int, int, int, int] – The shape (n_p, n_q, n_r, n_s) of the two body tensor (pq|rs).

class CompactTwoBodyIntegralsS8(*compact_array*)

Bases: BaseCompactTwoBodyIntegrals

Stores a two-body integral tensor in s8 symmetry reduced form.

Allows simple, 4-indexed [i,j,k,l] array-like access through index transformation.

Note: This symmetry class can only be used for ERI tensors that describe orbitals with the same spin i.e the restricted spin ERI tensor, or the aaaa and bbbb unrestricted ERI tensors. It *cannot* be used for the aabb unrestricted tensor, because it doesn't have ij<->kl swap symmetry.

Parameters
compact_array (ndarray) – Two-body integrals in an eight-fold symmetry-reduced form. A numpy.ndarray with shape (n_pp) where n_pp = n_pair * (n_pair + 1) / 2 and n_pair = n_orb * (n+orb + 1) / 2.

astype(*dtype*)

Returns a copy of the current compact integrals, with the compact array cast to a new type.

Parameters
dtype (Any) – The dtype to cast into.

Returns
CompactTwoBodyIntegralsS8 – New compact integrals object with compact array cast to dtype.

static check_s8_symmetry(*uncompact_array*, *rtol=1.0e-5*, *atol=1.0e-8*, *equal_nan=False*)

Tests whether the input ERI tensor has eight-fold (s8) index symmetries.

Checks for symmetries under index swaps i<->j and k<->l, and pair index swaps ij<->kl.

Parameters

- **uncompact_array** (ndarray) – Four-dimensional ERI tensor.
- **rtol** (float, default: 1.0e-5) – Relative tolerance.
- **atol** (float, default: 1.0e-8) – Absolute tolerance.
- **equal_nan** (bool, default: False) – Whether to compare NaN's as equal.

Returns
bool – True if the input array has s8 symmetry, false otherwise.

property dtype: dtype

Returns numpy data type of the compact array.

Return type
dtype

classmethod from_uncompacted_integrals(*two_body*, *symmetry*, *check_symmetry=False*, *rtol=1.0e-5*, *atol=1.0e-8*, *equal_nan=False*)

Builds a compact integrals object from an uncompacted set of two body integrals (a rank-4 tensor).

Parameters

- **two_body** (ndarray) – 4D array.
- **symmetry** (Union[str, int]) – Code to specify target symmetry. Uses the same convention as pyscf. Currently supports s4 and s8 symmetry.
- **check_symmetry** (bool, default: False) – Whether the input array should be checked for the requested symmetry.
- **rtol** (float, default: 1.0e-5) – Relative tolerance on symmetry checks.
- **atol** (float, default: 1.0e-8) – Absolute tolerance.
- **equal_nan** (bool, default: False) – Whether to compare NaN's as equal.

Returns
Union[*CompactTwoBodyIntegralsS4*, *CompactTwoBodyIntegralsS8*] – Object containing the same information as the input array in a compact form.

property imag: *CompactTwoBodyIntegralsS8*

Extract the imaginary part of the two-body integrals.

Returns
CompactTwoBodyIntegralsS8 – New compact object containing only the imaginary part of the integrals.

static pairs(*n_orb*)

Yields unique index pairs, and their pair index.

Does not return equivalent pairs i.e. {2, 1} will appear, but {1, 2} will not.

Parameters
n_orb (int) – Number of orbitals over which pairs will be iterated.

Yields
Pair index, first orbital index, second orbital index.

Return type
Iterator[Tuple[int, int, int]]

property real: *CompactTwoBodyIntegralsS8*

Extract the real part of the two-body integrals.

Returns
CompactTwoBodyIntegralsS8 – New compact object containing only the real part of the integrals.

rotate(*u*)

Perform a rotation of the compact two-body integrals.

Parameters

u (ndarray) – A real, orthogonal matrix of dimensions (n_orb, n_orb).

Return type

None

property shape: Tuple[int, int, int, int]

The shape of the corresponding rank-4 tensor.

Returns

Tuple[int, int, int, int] – The shape (n_p, n_q, n_r, n_s) of the two body tensor (pq|rs).

class DoubleFactorizedHamiltonian(*constant*, *one_body*, *one_body_offset*, *two_body*)

Bases: object

Restricted-spin hamiltonian operator, with two-body integrals stored in double factorized form.

Stores a hamiltonian of the form $H = H_0 + H_1 + S + V$ where $S + V$ is the coulomb interaction. $V = (1/2)\sum_{ijkl}(ij|kl)a_i^\dagger a_j a_k^\dagger a_l$ is a reordered two-body operator and S is a one-body-like offset to the two-body operator given by $S = \sum_{ij} s_{ij} a_i^\dagger a_j$ where $s_{ij} = -(1/2)\sum_k (ik|kj)$. H_0 and H_1 are the constant and one-electron terms respectively.

Double factorized two-body operator has the form: $V = (1/2)\sum_t^{N_\gamma} R(U_t) A_t R(U_t)^\dagger$ where $A_t = \gamma^t \left(\sum_u^{N_\lambda^t} \lambda_u^t a_u^\dagger a_u\right)^2$ and $R(U_t)$ is a basis rotation operator.

One-body-like terms may be consolidated into a single effective-one-body term: $H_1' = H_1 + S$. One-body term(s) may be stored as arrays, or diagonalized. When diagonalized, one-body terms are given by: $H_1 = R(W)\left(\sum_i^N \omega_i a_i^\dagger a_i\right) R(W)^\dagger$.

Parameters

- **constant** (float) – Constant (electron-independent) energy.
- **one_body** (Union[DiagonalizedOneBodyIntegrals, ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]]) – One-body integrals. These are the physical one-body integrals H_1 if one_body_offset != None, or effective one-body integrals H_1' if one_body_offset == None.
- **one_body_offset** (Union[DiagonalizedOneBodyIntegrals, ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], None]) – Contracted ERI tensor s_{ij}. If None, they are assumed to be part of one_body.
- **two_body** (DoubleFactorizedTwoBodyIntegrals) – Double factorized two-body integrals.

get_all_operators(*include_constant=True*, *include_one_body=True*, *include_one_body_offset=True*, *include_two_body=True*, *square_two_body_sum=True*)

Return all fermion operators alongside accompanying rotation matrices.

Parameters

- **include_constant** (bool, default: True) – Whether to include the constant energy term.
- **include_one_body** (bool, default: True) – Whether to include the physical/effective one-body terms. Calls *get_one_body_operator()*.

- **include_one_body_offset** (bool, default: True) – Whether to include the one-body offset terms. Calls *get_one_body_offset_operator()*. If one_body_offset == None, this term is omitted anyway.
- **include_two_body** (bool, default: True) – Whether to include the two-body terms. Calls *get_two_body_operators()*.
- **square_two_body_sum** (bool, default: True) – Whether to square the linear combination of number operators in the two-body term. Passed to *get_two_body_operators()*.

Returns

Tuple[List[*FermionOperator*], List[Optional[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]]]] – List of all fermion operators terms, list of corresponding rotation matrices.

get_one_body_offset_operator()

Generates the *FermionOperator* and rotation matrix for the one-body offset terms.

If one_body_offset is diagonalized, returns the operator $\sum_i \omega_i a_i^\dagger a_i$, alongside the eigenvector/rotation matrix for the one-body offset integrals. Otherwise returns $\sum_{ij} s_{ij} a_i^\dagger a_j$ and no rotation matrix.

Returns

Tuple[Optional[*FermionOperator*], Optional[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]]] – One-body fermion operator and accompanying rotation matrix (if required). If all one-body-like terms are consolidated into an effective-one-body, returns (None, None).

get_one_body_operator()

Generates the *FermionOperator* and rotation matrix for the one-body/effective-one-body terms.

If one_body is diagonalized, returns the operator $\sum_i \omega_i a_i^\dagger a_i$, alongside the eigenvector/rotation matrix for the one-body integrals, W. Otherwise returns $\sum_{ij} h_{ij} a_i^\dagger a_j$ and no rotation matrix.

Returns

Tuple[*FermionOperator*, Optional[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]]] – One-body fermion operator and accompanying rotation matrix (if required).

get_two_body_operators(*square_two_body_sum=True*)

Generates the *FermionOperator* and rotation matrices U^t for the two-body terms.

Parameters

square_two_body_sum (bool, default: True) – Whether to square the linear combination of FermionOperators.

If square_sum is True, returned operators have the full form: $\gamma^t \left(\sum_u^{N_\lambda^t} \lambda_u^t a_u^\dagger a_u\right)^2 /2$ for all t. If False, returns $\sqrt{\gamma^t \sum_u^{N_\lambda^t} \lambda_u^t a_u^\dagger a_u /2}$ for all t.

Returns

Tuple[List[*FermionOperator*], List[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]]] – List of two-body fermion operators, list of rotation matrices.

class FCIDumpRestricted

Bases: object

This class reads FCIDUMP files and facilitates transformations to native inquanto objects.

Symmetry information is not extracted from the fcidump files. To be used in inquanto, symmetry information should be passed by the user to the relevant space object. The vanilla information is, however, stored in the specification attribute, so a user can map from FCIDUMP integer values to irrep labels using their own code.

Notes

Currently this functionality is tested against pyscf, psi4 and nwchem FCIDUMP files for restricted systems only. User experience may differ when using this with files generated by packages not listed here.

Parameters
filename – The name of the FCIDUMP file to be handled by the object.

get_system_specification()

Read the system specification block of the FCIDUMP file and process the contents into a dictionary.

Parameters
filename – The name of the FCIDUMP file.

Returns
Dict – A dictionary where each key is a named element of the system specification block.

load(*filename*)

Load the contents of the FCIDUMP file in one IO operation.

This method populates the constant, one_body_list, and two_body_list attributes.

Parameters
filename (str) –

Return type
None

one_body_to_array()

Process the one-body data into a NxN matrix, where N is the number of spatial orbitals.

Elements which are permutationally equivalent to those in the fcidump file are filled with the appropriate value.

Returns
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – The one-electron integrals as a matrix, as read from the fcidump file.

to_ChemistryRestrictedIntegralOperator()

Generate a ChemistryRestrictedIntegralOperator object from the provided file.

The integrals from the file are unpacked and distributed into the full NxN one-body matrix and NxNxNxN two-body tensor.

Parameters
filename – The name of the FCIDUMP file.

Returns
ChemistryRestrictedIntegralOperator – The ChemistryRestrictedIntegralOperator corresponding to the information contained in the FCIDUMP file with name filename.

to_arrays()

Transform the one- and two-body data into an NxN matrix and an NxNxNxN tensor, respectively.

Chemists' notation is followed, and permutationally equivalent elements to those in the FCIDUMP file are filled.

Returns

Tuple[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]] – The one- and two-body integral arrays as a tuple.

two_body_to_tensor()

Process the one-body data into an NxNxNxN tensor, where N is the number of spatial orbitals.

Elements which are permutationally equivalent to those in the fcidump file are filled with the appropriate value.

Returns

ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – The two-electron integrals as a tensor, as read from the fcidump file.

write(*filename*, *constant*, *one_body*, *two_body*, *norb*, *nelec*, *multiplicity*, *orbsym=None*, *isym=None*, *tolerance=1e-12*)

Write an FCIDUMP file corresponding to the provided constant, one body and two body quantities.

Parameters

- **filename** (str) – The name of the file to write the FCIDUMP to.
- **constant** (float) – The constant term of the hamiltonian.
- **one_body** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – The spatial orbital, one-body integrals as a numpy array.
- **two_body** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – The spatial orbital, two-body integrals as a numpy array.
- **norb** (int) – The number of spatial orbitals in the system.
- **nelec** (int) – The number of electrons in the system.
- **multiplicity** (int) – The multiplicity of the system.
- **orbsym** (Optional[List[int]], default: None) – The integer value of the irreps of the spatial orbitals.
- **isym** (Optional[int], default: None) – The integer value of the point group.
- **tolerance** (float, default: 1e-12) – Write integrals to the file with a value greater than this number.

Return type

None

class FermionOperator(*data=None*, *coeff=1.0*)

Bases: Operator

Handles fermionic operator representation in a Fock space.

Operator is stored as a dict, with the keys being FermionOperatorString objects (terms) and values being the corresponding term coefficients. Directly instantiate with a tuple, a list of tuples or a dict.

Parameters

- **data** (Union[Tuple, *FermionOperatorString*, List[Tuple[Union[int, float, complex, Expr], Union[*FermionOperatorString*, str]]], Dict[*FermionOperatorString*, Union[int, float, complex, Expr]],

None], default: None) – Input data from which the fermion operator is built. Multiple input formats are supported.

- **coeff** (Union[int, float, complex, Expr], default: 1.0) – Multiplicative scalar coefficient. Used only when data is of type tuple or *FermionOperatorString*.

Examples

```
>>> fos = FermionOperatorString(((1, 0), (2, 1)))
>>> op = FermionOperator(fos, 1.0)
>>> print(op)
(1.0, F1  F2^)
>>> op = FermionOperator({fos: 1.0})
>>> print(op)
(1.0, F1  F2^)
```

class TrotterizeCoefficientsLocation(*value*)

Bases: str, Enum

Determines where coefficients will be stored upon performing Trotterization.

INNER = 'inner'

All coefficients will be stored in the "inner" coefficients, within the component QubitOperators in the Trotterized result.

MIXED = 'mixed'

The Trotter step factor will be stored in the "outer" coefficients, whereas the original coefficients of the original Operator will remain in the component Operators.

OUTER = 'outer'

All coefficients will be stored in the "outer" coefficients, the coefficients stored directly in the generated OperatorList.

apply_bra(*fock_state*)

Performs an operation on a bra FermionState state.

Parameters
fock_state (*FermionState*) – FermionState object (bra state).

Returns
FermionState – New bra state.

apply_ket(*fock_state*)

Performs an operation on a ket FermionState state.

Parameters
fock_state (*FermionState*) – FermionState object (ket state).

Returns
FermionState – New ket state.

approx_equal_to(*other*, *abs_tol=1e-10*)

Checks for equality between two FermionOperators.

First, dictionary equivalence is tested for. If false, operators are compared in normal-ordered form, and difference is compared to *abs_tol*.

Parameters

- **other** (*FermionOperator*) – An operator to test for equality with.
- **abs_tol** (float, default: 1e-10) – Tolerance threshold for negligible terms in comparison.

Returns

bool – True if this operator is equal to other, otherwise False.

Danger: This method may use the *normal_ordered()* method internally, which may be exponentially costly.

approx_equal_to_by_random_subs(*other*, *order=1*, *abs_tol=1e-10*)

Checks if object's dictionary values are numerically identical to the other object values.

Symbols contained in the difference of the two objects, if any, are substituted by random numeric values prior to norm check.

Parameters

- **other** (LinearDictCombiner) – Object to compare to.
- **order** (int, default: 1) – Parameter specifying the norm formula (see numpy.linalg.norm documentation).
- **abs_tol** (float, default: 1e-10) – Threshold vs which the norm of the difference is checked.

Return type

bool

as_scalar(*abs_tol=None*)

If the operator is a sum of identity terms or zero, return the sum of the coefficients, otherwise return None.

Note that this does not perform combination on terms and will return zero only if all coefficients are zero.

Parameters

abs_tol (Optional[float], default: None) – Tolerance for checking if coefficients are zero. Set to None to test using a standard python == comparison.

Returns

Union[float, complex, None] – The operator as a scalar if it can be represented as such, otherwise None.

classmethod ca(*a*, *b*)

Return object with a single one-body creation-annihilation pair with a unit coefficient in a dict.

Parameters

- **a** (int) – Index of fermionic mode to which a creation operator is applied.
- **b** (int) – Index of fermionic mode to which an annihilation operator is applied.

Returns

The creation-annihilation operator pair.

Examples

```
>>> print(FermionOperator.ca(2, 0))
(1.0, F2^ F0 )
```

classmethod caca(*a*, *b*, *c*, *d*)

Return object with a single two-body creation-annihilation pair with a unit coefficient in a dict.

Ordered as creation-annihilation-creation-annihilation.

Parameters

- **a** (int) – Index of fermionic mode to which the first creation operator is applied.
- **b** (int) – Index of fermionic mode to which the first annihilation operator is applied.
- **c** (int) – Index of fermionic mode to which the second creation operator is applied.
- **d** (int) – Index of fermionic mode to which the second annihilation operator is applied.

Returns

The composed two-body operator.

Examples

```
>>> print(FermionOperator.caca(2, 0, 3, 1))
(1.0, F2^ F0  F3^ F1 )
```

---

classmethod ccaa(*a*, *b*, *c*, *d*)

Return object with a single two-body creation-annihilation pair with a unit coefficient in a dict.

Ordered as creation-creation-annihilation-annihilation.

Parameters

- **a** (int) – Index of fermionic mode to which the first creation operator is applied.
- **b** (int) – Index of fermionic mode to which the second creation operator is applied.
- **c** (int) – Index of fermionic mode to which the first annihilation operator is applied.
- **d** (int) – Index of fermionic mode to which the second annihilation operator is applied.

Returns

The composed two-body operator.

Examples

```
>>> print(FermionOperator.ccaa(3, 2, 1, 0))
(1.0, F3^ F2^ F1  F0 )
```

---

clone()

Performs shallow copy of the object.

Return type

TypeVar(SYMBOLICTYPE, bound= Symbolic)

property coefficients: List[int | float | complex | Expr]

Returns dictionary values.

Return type

List[Union[int, float, complex, Expr]]

commutator(*other_operator*)

Return the commutator of self with other_operator.

This calculation is performed by explicit calculation through multiplication.

Parameters
other_operator (*FermionOperator*) – The other fermionic operator.

Returns
FermionOperator – The commutator.

commutes_with(*other_operator*, *abs_tol=1e-10*)

Returns True if self commutes with other_operator (within tolerance), otherwise False.

Parameters

- **other_operator** (*FermionOperator*) – The other fermionic operator.
- **abs_tol** (float, default: 1e-10) – Tolerance threshold for negligible terms in the commutator.

Returns
bool – True if operators commute, within tolerance, otherwise False.

compress(*abs_tol=1e-10*, *symbol_sub_type=CompressSymbolSubType.NONE*)

Combines duplicate terms, removing those with negligible coefficient.

Parameters

- **abs_tol** (float, default: 1e-10) – Tolerance for comparing values to zero.
- **symbol_sub_type** (Union[CompressSymbolSubType, str], default: CompressSymbolSubType.NONE) – Defines the behaviour for dealing with symbolic expressions in coefficients. If “none”, symbolic expressions are left intact. If “unity”, substitutes all free symbols with 1, and removes all imaginary and real components below tolerance. If “random”, substitutes all free symbols with a random number between 0 and 1, and removes imaginary and real components below tolerance.

Warning: When :code: *symbol_sub_type != “none”*, this method assumes significant expression structure is known a priori, and is best suited to operators which have simple product expressions, such as excitation operators for VQE ansätze and digital quantum simulation. Otherwise, it may remove terms relevant to computation. Each expression is of the form $f(a_1, a_2, \ldots, a_n)$ for some symbols a_i. $|f(a_1, a_2, \ldots, a_n)|$ is assumed to monotonically increase in both real and imaginary components for all $a_i \in [0, 1]$.

Returns
LinearDictCombiner – Updated instance of LinearDictCombiner.

classmethod constant(*value*)

Return object with a provided constant entry in a dict.

Returns
Operator representation of provided constant scalar.

Examples

```
>>> print(FermionOperator.constant(0.5))
(0.5, )
```

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

dagger()

Returns the Hermitian conjugate of an operator.

Return type
FermionOperator

df()

Returns a Pandas DataFrame object of the dictionary.

Return type
DataFrame

empty()

Checks if dictionary is empty.

Return type
bool

evalf(**args*, ***kwargs*)

Evaluates symbolic expressions stored in dict values and replaces them with the results.

Parameters

- **args** (Any) – Args to be passed to *sympy.evalf()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.evalf()*.

Returns
LinearDictCombiner – Updated instance of LinearDictCombiner.

free_symbols()

Returns the free symbols in the coefficient values.

free_symbols_ordered()

Returns the free symbols in the dict, ordered alphabetically.

Returns
SymbolSet – Ordered set of symbols.

freeze(*index_map*, *occupation*)

Adaptation of OpenFermion's freeze_orbitals method with mask and consistent index pruning.

Parameters

- **index_map** (List[int]) – A list of integers or None entries, whose size is equal to the number of spin-orbitals, where None indicates orbitals to be frozen and the remaining sequence of integers is expected to be continuous.
- **occupation** (List[int]) – A list of 1s and 0s of the same length as index_map, indicating occupied and unoccupied orbitals.

Returns
FermionOperator – New operator with frozen orbitals removed.

classmethod from_list(*data*)

Construct *FermionOperator* from a list of tuples.

Parameters

data (List[Tuple[Union[int, float, complex, Expr], *FermionOperatorString*]]) – List data representing a fermion operator term or sum of fermion operator terms. Each term in the list must be a tuple containing a scalar multiplicative coefficient, followed by a *FermionOperatorString* or tuple (see *FermionOperator.from_tuple()*).

Returns

FermionOperator – Fermion operator object corresponding to a valid input.

Examples

```
>>> fos0 = FermionOperatorString(((1, 0), (2, 1)))
>>> fos1 = FermionOperatorString(((1, 0), (3, 1)))
>>> op = FermionOperator.from_list([(0.9, fos0), (0.1, fos1)])
>>> print(op)
(0.9, F1  F2^), (0.1, F1  F3^)
>>> op = FermionOperator.from_list([(0.9, ((1, 0), (2, 1))), (0.1, fos1)])
>>> print(op)
(0.9, F1  F2^), (0.1, F1  F3^)
```

classmethod from_string(*input_string*)

Constructs a child class instance from a string.

Parameters

input_string (str) – String in the format `coeff1 [(coeff1_1, term1_1), ..., (coeff1_n, term1_n)], ..., coeffn [(coeffn_1, termn_1), ...]`.

Returns

Child class object.

classmethod from_tuple(*data*, *coeff=1.0*)

Construct *FermionOperator* from a tuple of terms.

Parameters

- **data** (Tuple) – Data representing a fermion operator term, which may be a product of single fermion creation or anihilation operators. Creation and annihilation operators acting on orbital index `q` are given by tuples `(q, 0)` and `(q, 1)` respectively. A product of single operators is given by a tuple of tuples; for example, the number operator: `((q, 1), (q, 0))`.
- **coeff** (Union[int, float, complex, Expr], default: `1.0`) – Multiplicative scalar coefficient.

Returns

FermionOperator – Fermion operator object corresponding to a valid input.

Examples

```
>>> op = FermionOperator.from_tuple(((1, 0), (2, 1)), 1.0)
>>> print(op)
(1.0, F1  F2^)
```

classmethod identity()

Return object with an identity entry in a dict.

Examples

```
>>> print(FermionOperator.identity())
(1.0, )
```

infer_num_spin_orbs()

Returns the number of modes that this operator acts upon, inferring the existence of modes with index from 0 to the maximum index.

Returns

int – The minimum number of spin orbitals in the Fock space to which this *FermionOperator* operates on.

Examples

```
>>> op = FermionOperator.from_string("(1.0, F1 F2^)")
>>> print(op.infer_num_spin_orbs())
3
```

is_all_coeff_complex()

Check if all coefficients are complex values.

Warning: Returns None if there is a free symbol in a coefficient.

Return type

bool

is_all_coeff_imag()

Check if all coefficients are complex values.

Warning: Returns None if there is a free symbol in a coefficient.

Return type

bool

is_all_coeff_real()

Check if all coefficients are real values.

Warning: Returns None if there is a free symbol in a coefficient.

Return type

bool

is_all_coeff_symbolic()

Check if all coefficients contain free symbols.

Return type
bool

is_antihermitian(*abs_tol=1e-10*)

Returns True if operator is anti-Hermitian (within a tolerance), else False.

This explicitly calculates the Hermitian conjugate, multiplies by -1 and tests for equality. Normal-ordering is performed before comparison.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold for negligible terms in comparison.

Returns
bool – True if operator is antihermitian, otherwise False.

Danger: This method uses the *normal_ordered()* method internally, which may be exponentially costly.

is_any_coeff_complex()

Check if any coefficient is a complex value.

Warning: Returns None if there is a free symbol in a coefficient.

Return type
bool

is_any_coeff_imag()

Check if any coefficient is an imaginary value.

Warning: Returns None if there is a free symbol in a coefficient.

Return type
bool

is_any_coeff_real()

Check if any coefficient is a real value.

Warning: Returns None if there is a free symbol in a coefficient.

Return type
bool

is_any_coeff_symbolic()

Check if any coefficient contains a free symbol.

Return type
bool

is_commuting_operator()

Return True if every term in operator commutes with every other term, otherwise False.

Return type
bool

is_hermitian(*abs_tol=1e-10*)

Returns True if operator is Hermitian (within a tolerance), else False.

This explicitly calculates the Hermitian conjugate and tests for equality. Normal-ordering is performed before comparison.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold for negligible terms in comparison.

Returns
bool – True if operator is Hermitian, otherwise False.

Danger: This method uses the *normal_ordered()* method internally, which may be exponentially costly.

is_normal_ordered()

Return whether or not term is in normal-ordered form.

This method is a (modified) version of the OpenFermion *is_normal_ordered()* method. In our convention, a normal-ordered operator operator has all creation operators to the left of annihilation operators, and each "block" of creation/annihilation operators are ordered with orbital indices in descending order (from left to right).

Returns
bool – True if operator is normal-ordered, false otherwise.

Examples

```
>>> op = FermionOperator.from_string("(3.5, F1 F2^)")
>>> print(op.is_normal_ordered())
False
>>> op = FermionOperator.from_string("(3.5, F1^ F2^)")
>>> print(op.is_normal_ordered())
False
```

is_parallel_with(*other*, *abs_tol=1e-10*)

Returns True if other is parallel with this (i.e. a scalar multiple of this), otherwise False.

Parameters

- **other** (LinearDictCombiner) – The other object to compare against
- **abs_tol** (Optional[float], default: 1e-10) – Tolerance threshold for comparison. Set to None to test for exact equivalence.

Returns
bool – True if other is parallel with this, otherwise False.

is_self_inverse(*abs_tol=1e-10*)

Returns True if operator is its own inverse (within a tolerance), False otherwise.

This explicitly calculates the square of the operator and compares to the identity. Normal-ordering is performed before comparison.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold for negligible terms in comparison.

Returns
bool – True if operator is self-inverse, otherwise False.

Danger: This method uses the *normal_ordered()* method internally, which may be exponentially costly.

is_two_body_number_conserving(*check_spin_symmetry=False*)

Query whether operator has correct form to be from a molecule.

This method is a copy of the OpenFermion *is_two_body_number_conserving()* method.

Require that term is particle-number conserving (same number of raising and lowering operators). Require that term has 0, 2 or 4 ladder operators. Require that term conserves spin (parity of raising operators equals parity of lowering operators).

Parameters
check_spin_symmetry (*bool*) – Whether to check if operator conserves spin.

Returns
bool – True if operator conserves electron number (and, optionally, spin), False otherwise.

Examples

```
>>> op = FermionOperator.from_string("(1.0, F1^ F0)")
>>> print(op.is_two_body_number_conserving())
True
>>> op = FermionOperator.from_string("(1.0, F0^)")
>>> print(op.is_two_body_number_conserving())
False
>>> op = FermionOperator.from_string("(1.0, F1^ F0^)")
>>> print(op.is_two_body_number_conserving())
False
>>> op = FermionOperator.from_string("(1.0, F1^ F0)")
>>> print(op.is_two_body_number_conserving(check_spin_symmetry=True))
False
```

is_unit_1norm(*abs_tol=1e-10*)

Returns True if operator has unit 1-norm, else False.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold for comparison with unity.

Return type
bool

is_unit_2norm(*abs_tol=1e-10*)

Returns True if operator has unit 1-norm, else False.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold for comparison with unity.

Return type
bool

is_unit_norm(*order=2*, *abs_tol=1e-10*)

Returns True if operator has unit p-norm, else False.

Parameters

- **order** (int, default: 2) – Norm order.
- **abs_tol** (float, default: 1e-10) – Tolerance threshold for comparison with unity.

Raises
ValueError – Coefficients contain free symbols.

Return type
bool

is_unitary(*abs_tol=1e-10*)

Returns True if operator is unitary (within a tolerance), False otherwise.

This explicitly calculates the Hermitian conjugate, right-multiplies by the initial operator and tests for equality to the identity. Normal-ordering is performed before comparison.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold for negligible terms in comparison.

Returns
bool – True if operator is unitary, otherwise False.

Danger: This method uses the *normal_ordered()* method internally, which may be exponentially costly.

items()

Returns dictionary items.

Return type
ItemsView[Any, Union[int, float, complex, Expr]]

static key_from_str(*key_str*)

Returns a FermionOperatorString instance initiated from the input string.

Parameters
key_str (str) – An input string describing the FermionOperatorString to be created - see FermionOperatorString.from_string() for more detail.

Returns
FermionOperatorString – A generated FermionOperatorString.

list_class

alias of *FermionOperatorList*

make_hashable()

Return a hashable representation of the object.

Currently simply returns a stringified object.

Return type
str

map(*mapping*)

Updates dictionary values, using a mapping function provided.

Parameters

mapping (Callable[[Union[int, float, complex, Expr]], Union[int, float, complex, Expr]]) – Mapping function to update the dict.

Returns

LinearDictCombiner – None.

property n_symbols: int

Returns the number of free symbols in the object.

Return type

int

norm_coefficients(*order=2*)

Returns the p-norm of the coefficients.

Parameters

order (int, default: 2) – Norm order.

Return type

Union[complex, float]

normal_ordered()

Returns a normal-ordered version of *FermionOperator*.

Normal-ordering the operator moves all creation operators to the left of annihilation operators, and orders orbital indices in descending order (from left to right).

Notes

Makes use of OpenFermion _normal_ordered_ladder_term() function.

Examples

```
>>> op = FermionOperator.from_string("(0.5, F1 F2^), (0.2, F1 F2 F3^ F4^)")
>>> op_no = op.normal_ordered()
>>> print(op_no)
(-0.5, F2^ F1 ), (0.2, F4^ F3^ F2  F1 )
```

Return type

FermionOperator

normalized(*norm_value=1.0*, *norm_order=2*)

Returns a copy of this object with normalised coefficients.

Parameters

- **norm_value** (float, default: 1.0) – The desired norm of the returned operator.
- **norm_order** (int, default: 2) – The order of the norm to be used.

Returns

LinearDictCombiner – A copy of the object with coefficients normalised to the desired value.

property num_spin_orbs: int

Return the number of spin-orbitals that this operator explicitly acts on.

Return type
int

permuted_operator(*permutation*)

Permutes the indices in a FermionOperator.

Permutation is according to a list or a dict of indices, mapping the old to a new operator order.

Parameters
permutation (Union[Dict, List[int]]) – List of integers or a dict, mapping the old operator terms indices to the new ones. In case if a list is given, list index acts as a key (old index) and list value corresponds to the new index of an operator. If a dict is given, it can only contain the indices to be permuted for higher efficiency.

Returns
FermionOperator – Permuted FermionOperator object.

Examples

```
>>> op = FermionOperator.ccaa(1,4,5,6) + FermionOperator.ca(0,4)
>>> print(op)
(1.0, F1^ F4^ F5  F6 ), (1.0, F0^ F4 )
>>> print(op.permuted_operator([0,4,2,3,1,5,6]))
(1.0, F4^ F1^ F5  F6 ), (1.0, F0^ F1 )
>>> print(op.permuted_operator({1:4, 4:1}))
(1.0, F4^ F1^ F5  F6 ), (1.0, F0^ F1 )
```

---

print_table()

Print dictionary formatted as a table.

Return type
NoReturn

qubit_encode(*mapping=None*, *qubits=None*)

Performs qubit encoding (mapping), using provided mapping function, of the current FermionOperator.

Parameters

- **mapping** (QubitMapping, default: None) – Mapping class. Default mapping procedure is Jordan-Wigner.
- **qubits** (Optional[List[Qubit]], default: None) – The qubit register. If left as None, a default register will be assumed if possible. See the mapping class documentation for further details.

Returns
Mapped QubitOperator object.

remove_global_phase(*phase=0.0*)

Returns a copy with a global phase applied such that the first element has the desired phase.

Parameters
phase (float, default: 0.0) – The phase to yield on the first element, in half-turns (i.e. multiples of pi).

Returns
LinearDictCombiner – A copy of the object with the desired global phase applied.

reversed_order()

Reverses the order of terms and returns it as a new object.

Return type
LinearDictCombiner

simplify(**args*, ***kwargs*)

Simplifies expressions stored in dictionary values.

Parameters

- **args** (Any) – Args to be passed to *sympy.simplify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.simplify()*.

Returns
LinearDictCombiner – Updated instance of LinearDictCombiner.

split()

Generates single-term FermionOperator objects.

subs(*symbol_map*)

Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution(*symbol_map=None*)

Substitutes free symbols for numerical values according to a map.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) –

Return type
LinearDictCombiner

sympify(**args*, ***kwargs*)

Sympifies dictionary values.

Replaces values with their corresponding symbolic expressions.

Parameters

- **args** (Any) – Args to be passed to *sympy.sympify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.sympify()*.

Returns
LinearDictCombiner – Updated instance of LinearDictCombiner.

Raises
RuntimeError – When sympification fails.

property terms: List[Any]

Returns dictionary keys.

Return type
List[Any]

to_ChemistryRestrictedIntegralOperator()

Convert fermion operator into a restricted integral operator.

Uses the *ChemistryRestrictedIntegralOperator.from_FermionOperator()* method internally.

Return type
ChemistryRestrictedIntegralOperator

to_ChemistryUnrestrictedIntegralOperator()

Convert fermion operator into an unrestricted integral operator.

Uses the *ChemistryUnrestrictedIntegralOperator.from_FermionOperator()* method internally.

Return type
ChemistryUnrestrictedIntegralOperator

to_latex(*imaginary_unit='\\\\text{i}'*, ***kwargs*)

Generate a LaTeX representation of the operator.

Parameters

- **imaginary_unit** (str, default: r"\\text{i}") – Symbol to use for the imaginary unit.
- ****kwargs** – Keyword arguments passed to the *to_latex()* method of component operator strings (*FermionOperatorString* or *QubitOperatorString*).

Returns
str – LaTeX compilable equation string.

Examples

```
>>> from inquanto.operators import FermionOperator
>>> from sympy import sympify
>>> c = sympify("c")
>>> fo = FermionOperator([(-c, "F1 F2^"), (c**2, "F1^"), (c, "F0 F5 F3^")])
>>> print(fo.to_latex())
- c a_{1} a_{2}^{\dagger} + c^{2} a_{1}^{\dagger} + c a_{0} a_{5} a_{3}^{\
↪dagger}
>>> fo = FermionOperator([(1.0, "F0^ F1^ F3^"), (-1.0j, "F3 F1 F0")])
>>> print(fo.to_latex())
a_{0}^{\dagger} a_{1}^{\dagger} a_{3}^{\dagger} -\text{i} a_{3} a_{1} a_{0}
>>> from sympy import sqrt
>>> fo = FermionOperator([(sqrt(2), "F0^ F1"), (0.5-8j, "F5 F5^"), (2j, "F1^␣
↪F1^")])
>>> print(fo.to_latex(imaginary_unit=r"\text{j}", operator_symbol="f"))
\sqrt{2} f_{0}^{\dagger} f_{1} + (0.5-8.0\text{j}) f_{5} f_{5}^{\dagger} + 2.
↪0\text{j} f_{1}^{\dagger} f_{1}^{\dagger}
```

```
>>> from inquanto.operators import QubitOperator, QubitOperatorString
>>> qos1 = QubitOperatorString.from_string("X1 Y2 Z3")
```

(continues on next page)

(continued from previous page)

```
>>> qos2 = QubitOperatorString.from_string("Z0 Y1 X4")
>>> qos3 = QubitOperatorString.from_list([(("a", [0]), "I"), (("b", [1]), "Z
↪"), (("c", [2]), "Z")])
>>> qo = QubitOperator({qos1: -1.0, qos2: 3+4j, qos3: c})
>>> print(qo.to_latex())
- X_{1} Y_{2} Z_{3} + (3.0+ 4.0\text{i}) Z_{0} Y_{1} X_{4} + c I_{0} Z_{1} Z_
↪{2}
>>> qo = QubitOperator({qos3: 3j})
>>> print(qo.to_latex(imaginary_unit="j", show_labels=True))
3.0j I^{\text{a}}_{0} Z^{\text{b}}_{1} Z^{\text{c}}_{2}
```

trotterize(*trotter_number=1*, *trotter_order=1*, *constant=1.0*, *coefficients_location=TrotterizeCoefficientsLocation.OUTER*)

Trotterizes the operator, treating the operator as an exponent.

Assuming that this operator is an exponent, this will generate an OperatorList with each element in the OperatorList corresponding to a single exponent in the Trotter product of exponentials.

Parameters

- **trotter_number** (int, default: 1) – The number of time-slices in the Trotter approximation.
- **trotter_order** (int, default: 1) – The order of the Trotter-Suzuki approximation being used. Currently, this only supports 1, i.e. ABABAB.
- **constant** (Union[float, complex], default: 1.0) – An additional constant factor to multiply each exponent by, which may be useful when, for example, constructing a time evolution operator.
- **coefficients_location** (*TrotterizeCoefficientsLocation*, default: TrotterizeCoefficientsLocation.OUTER) – By default, the coefficient of each term in the input operator is multiplied by the Trotter factor and stored in the outer coefficient of the generated OperatorList, with the coefficient of each inner operator set to 1. This behaviour can be controlled with this argument - set to "outer" for default behaviour. Setting this parameter to "inner" will store all scalars in the inner operator coefficient, with the outer coefficients of the generated OperatorList set to 1. Setting this parameter to "mixed" will store the Trotter factor in the outer coefficients of the generated OperatorList, with the inner coefficients of each term left untouched. See examples for a comparison.

Returns

TypeVar(OperatorListT, bound= OperatorList) – A Trotterized form of the exponential product, where each element is an individual exponent.

Examples

```
>>> from inquanto.operators import QubitOperator
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6) + QubitOperator("Z1 Z2 Z3 Z5", -5.6j)
>>> trotter_operator = op1.trotterize(trotter_number=2)
>>> print(trotter_operator)
2.3       [(1.0, X0 Y1 Z3)],
-2.8j     [(1.0, Z1 Z2 Z3 Z5)],
2.3       [(1.0, X0 Y1 Z3)],
-2.8j     [(1.0, Z1 Z2 Z3 Z5)]
>>> from inquanto.operators import QubitOperator
```

(continues on next page)

(continued from previous page)

```
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6) + QubitOperator("Z1 Z2 Z3 Z5", -5.6j)
>>> trotter_operator = op1.trotterize(trotter_number=2,coefficients_location=
↪"inner")
>>> print(trotter_operator)
1.0       [(2.3, X0 Y1 Z3)],
1.0       [(-2.8j, Z1 Z2 Z3 Z5)],
1.0       [(2.3, X0 Y1 Z3)],
1.0       [(-2.8j, Z1 Z2 Z3 Z5)]
>>> from inquanto.operators import QubitOperator
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6) + QubitOperator("Z1 Z2 Z3 Z5", -5.6j)
>>> trotter_operator = op1.trotterize(trotter_number=2,coefficients_location=
↪"mixed")
>>> print(trotter_operator)
0.5       [(4.6, X0 Y1 Z3)],
0.5       [(-5.6j, Z1 Z2 Z3 Z5)],
0.5       [(4.6, X0 Y1 Z3)],
0.5       [(-5.6j, Z1 Z2 Z3 Z5)]
```

truncated (*tolerance=1e-8*, *normal_ordered=True*)

Prunes *FermionOperator* terms with coefficients below provided threshold.

Parameters

- **tolerance** – Threshold below which terms are removed.
- **normal_ordered** – Should the operator be returned in normal-ordered form.

Returns

FermionOperator – New, modified operator.

Examples

```
>>> op = FermionOperator.from_string("(0.999, F0 F1^), (0.001, F2^ F1 )")
>>> print(op.truncated(tolerance=0.005, normal_ordered=False))
(0.999, F0  F1^)
>>> print(op.truncated(tolerance=0.005))
(-0.999, F1^ F0 )
```

unsympify()

Unsympifies dicionary values.

Replaces symbolic expressions with their corresponding numeric values.

Returns

LinearDictCombiner – Updated instance of LinearDictCombiner.

Raises

TypeError – When unsympification fails.

classmethod zero()

Return object with a zero dict entry.

Examples

```
>>> print(LinearDictCombiner.zero())
(0)
```

class FermionOperatorList(*data=None*, *coeff=1.0*)

Bases: OperatorList

Stores tuples of *FermionOperator* objects and corresponding coefficient values in an ordered list.

In contrast to *FermionOperator*, this class is not assumed to comprise a linear combination. Typically this will be of use when considering sequences of operators upon which some nonlinear operation is performed. For instance, this may be used to store a Trotterised combination of exponentiated *FermionOperator* objects. This class may be instantiated from a *FermionOperator* or a *FermionOperatorString* object and a scalar or symbolic coefficient. It may also be instantiated with a list of tuples of scalar or symbolic coefficients and *FermionOperator* objects.

Parameters

- **data** (Union[*FermionOperator*, *FermionOperatorString*, List[Tuple[Union[int, float, complex, Expr], *FermionOperator*]], None], default: None) – Input data from which the list of fermion operators is built. See *FermionOperator* for methods of constructing terms.
- **coeff** (Union[int, float, complex, Expr], default: 1.0) – Multiplicative scalar coefficient. Used only if data is not of type list.

Examples

```
>>> from sympy import sympify
>>> op1 = FermionOperator(FermionOperatorString(((1, 0), (2, 1))), 3.5)
>>> op2 = FermionOperator(FermionOperatorString(((0, 0), (3, 1))), 1.5)
>>> fto = FermionOperatorList([(sympify("a"), op1), (sympify("b"), op2)])
>>> print(fto)
a         [(3.5, F1  F2^)],
b         [(1.5, F0  F3^)]
>>> fto = FermionOperatorList(op1)
>>> print(fto)
1.0       [(3.5, F1  F2^)]
>>> fto = FermionOperatorList(FermionOperatorString(((1, 0), (2, 1))))
>>> print(fto)
1.0       [(1.0, F1  F2^)]
```

class CompressScalarsBehavior(*value*)

Bases: str, Enum

Governs compression of scalars method behaviour.

ALL = 'all'

Combine all coefficients possible, simplifying inner terms.

ONLY_IDENTITIES_AND_ZERO = 'simple'

Only compress based on terms which are a scalar multiple of the identity operator, or zero.

OUTER = 'outer'

Combine all “outer” coefficients (coefficients stored directly in the top-level OperatorList) into one.

class FactoryCoefficientsLocation(*value*)

Bases: str, Enum

Determines where the from_Operator method places coefficients.

INNER = 'inner'

Coefficients are left within the component operators.

OUTER = 'outer'

Coefficients are moved to be directly stored at the top-level of the OperatorList.

clone()

Performs shallow copy of the object.

Return type

TypeVar(SYMBOLICTYPE, bound= Symbolic)

collapse_as_linear_combination(*ignore_outer_coefficients=False*)

Treating the OperatorList as a linear combination, return it in the form of a Operator.

By default, each term is multiplied by its corresponding scalar coefficient, then all such multiplied terms are summed to yield a single Operator. The first step may be skipped (i.e. the scalar coefficients associated with each component Operator may be ignored) by setting `ignore_outer_coefficients` to True.

Parameters

ignore_outer_coefficients (bool, default: False) – Set to True to skip multiplication by the "outer" coefficients in the OperatorList.

Returns

TypeVar(OperatorT, bound= Operator) – The sum of all terms in the OperatorList, multiplied by their associated coefficients if requested.

collapse_as_product(*reverse=False*, *ignore_outer_coefficients=False*)

Treating the OperatorList as a product of separate terms, return the full product as an Operator.

By default, each Operator in the OperatorList is multiplied by its corresponding coefficient, and the product is taken sequentially with the leftmost term given by the first element of the OperatorList. This behaviour can be reversed with the `reverse` parameter - if set to True, the leftmost term will be given by the last element of OperatorList.

If ignore_outer_coefficients is set to True, the first step (the multiplication of Operator terms by their corresponding coefficients) is skipped - i.e. the "outer" coefficients stored in the OperatorList are ignored.

> **Danger:** In the general case, the number of terms in the expansion will blow up exponentially (and thus the runtime of this method will also blow up exponentially).

Parameters

- **reverse** (bool, default: False) – Set to True to reverse the order of the product.
- **ignore_outer_coefficients** (bool, default: False) – Set to True to skip multiplication by the "outer" coefficients in the OperatorList.

Returns

TypeVar(OperatorT, bound= Operator) – The product of each component operator.

compress_scalars_as_product(*abs_tol=1e-12*, *inner_coefficient=False*, *coefficients_to_compress=CompressScalarsBehavior.ONLY_IDENTITIES_AND_ZERO*)

Treating the OperatorList as a product, compress identity terms or resolve to zero if possible.

To do this, we iterate through the (coefficient, operator) pairs in the OperatorList. If any coefficient or operator is zero, then we return an empty OperatorList, as the product will be zero. If the iteration operator is an identity, it will be treated as a scalar multiplier, itself multiplied by its associated coefficient, with the operator itself removed from the OperatorList. These multipliers are multiplied together and – if they do not equate to 1 – are prepended to the OperatorList as a separate identity term.

By default, (coefficient,operator) pairs which are not identity or zero will be ignored. This behaviour may be controlled with the coefficients_to_compress parameter. This can be set to "outer" to include all "outer" coefficients (of the (coefficient,operator) pairs) in the prepended identity term. It can also be set to "all" to additionally bring coefficients within the non-identity operators into the prepended identity term – i.e. for operators where all the "inner" coefficients are equal. See examples for a comparison.

The prepended identity term will, by default, store the multiplied scalar factor in the "outer" coefficient of the (coefficient, operator) pair. The inner_coefficient parameter may be set to True to instead store it within the operator.

Note: Term simplification is not performed on component operators. It is possible that a component operator will resolve on simplification to the identity or zero. This method will not catch these occurrences.

Parameters

- **abs_tol** (`float`, default: `1e-12`) – Numerical threshold for comparison of numbers to 0 and 1. Set to None to use exact identity.
- **inner_coefficient** (`bool`, default: `False`) – Set to True to store generated scalar factors within the identity term, as described above.
- **coefficients_to_compress** (*CompressScalarsBehavior*, default: `CompressScalarsBehavior.ONLY_IDENTITIES_AND_ZERO`) – Controls which scalar factors will be combined, as described above.

Returns

`TypeVar`(`OperatorListT`, bound= OperatorList) – The OperatorList with identity and zero terms combined, as described above.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0", 2) + QubitOperator("Z1", 2)
>>> op2 = 3 * QubitOperator.identity()
>>> qol = QubitOperatorList([(5, op1), (7, op2)])
>>> result = qol.compress_scalars_as_product()
>>> print(result)
21.0      [(1.0, )],
5         [(2, X0), (2.0, Z1)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0", 2) + QubitOperator("Z1", 2)
>>> op2 = 3 * QubitOperator.identity()
>>> qol = QubitOperatorList([(5, op1), (7, op2)])
>>> result = qol.compress_scalars_as_product(inner_coefficient=True)
>>> print(result)
```

(continues on next page)

(continued from previous page)

```
1          [(21.0, )],
5          [(2, X0), (2.0, Z1)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0", 2) + QubitOperator("Z1", 2)
>>> op2 = 3 * QubitOperator.identity()
>>> qol = QubitOperatorList([(5, op1), (7, op2)])
>>> result = qol.compress_scalars_as_product(coefficients_to_compress="outer")
>>> print(result)
105.0      [(1.0, )],
1.0        [(2, X0), (2.0, Z1)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0", 2) + QubitOperator("Z1", 2)
>>> op2 = 3 * QubitOperator.identity()
>>> qol = QubitOperatorList([(5, op1), (7, op2)])
>>> result = qol.compress_scalars_as_product(coefficients_to_compress="all")
>>> print(result)
210.0      [(1.0, )],
1.0        [(1.0, X0), (1.0, Z1)]
```

copy()

Returns a deep copy of self.

Return type
LinearListCombiner

df()

Returns a Pandas DataFrame object of the dictionary.

empty()

Checks if internal list is empty.

Return type
bool

evalf(**args*, ***kwargs*)

Numerically evaluates symbolic expressions stored in the left and right values of list items and replaces them with the results.

Parameters

- **args** (Any) – Args to be passed to *sympy.evalf()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.evalf()*.

Returns
LinearListCombiner – Updated instance of LinearListCombiner.

free_symbols()

Returns the free symbols in the coefficient values.

Return type
set

free_symbols_ordered()

Returns the free symbols in the list, ordered alphabetically.

Returns
SymbolSet – Ordered set of symbols.

classmethod from_Operator(*input*, *additional_coefficient=1.0*, *coefficients_location=FactoryCoefficientsLocation.INNER*)

Converts an Operator to an OperatorList with terms in arbitrary order.

Each term in the Operator is split into a separate component Operator in the OperatorList. The resulting location of each scalar coefficient in the input Operator can be controlled with the coefficients_location parameter. Setting this to 'inner' will leave coefficients stored as part of the component operators, 'outer' will move the coefficients to the 'outer' level.

Parameters

- **input** (`Operator`) – The input operator to split into an OperatorList.
- **additional_coefficient** (`Union[int, float, complex, Expr]`, default: `1.0`) – An additional factor to include in the "outer" coefficients of the generated OperatorList.
- **coefficients_location** (*FactoryCoefficientsLocation*, default: `FactoryCoefficientsLocation.INNER`) – The destination of the coefficients of the input operator, as described above.

Returns

`TypeVar(OperatorListT, bound= OperatorList)` – An OperatorList as described above.

Raises

ValueError – On invalid input to the coefficients_location parameter.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op = QubitOperator("X0", 2.) + QubitOperator("Z1", 2.)
>>> qol = QubitOperatorList.from_Operator(op)
>>> print(qol)
1.0       [(2.0, X0)],
1.0       [(2.0, Z1)]
>>> op = QubitOperator("X0", 2.) + QubitOperator("Z1", 2.)
>>> qol = QubitOperatorList.from_Operator(op,coefficients_location='outer')
>>> print(qol)
2.0       [(1.0, X0)],
2.0       [(1.0, Z1)]
```

classmethod from_string(*input_string*)

Constructs a child class instance from a string.

Parameters

input_string (`str`) – String in the format `coeff1 [(coeff1_1, term1_1), ..., (coeff1_n, term1_n)], ..., coeffn [(coeffn_1, termn_1), ...]`.

Returns

Child class object.

infer_num_spin_orbs()

Returns the number of modes that the component operators act upon, inferring the existence of modes with index from 0 to the maximum index.

Returns

`int` – The minimum number of spin orbitals in the Fock space to which this operator list operates on.

Examples

```
>>> op1 = FermionOperator(FermionOperatorString(((1, 0), (2, 1))), 1.)
>>> op2 = FermionOperator(FermionOperatorString(((0, 0), (6, 1))), 1.)
>>> fto = FermionOperatorList([(1., op1), (1., op2)])
>>> print(fto.infer_num_spin_orbs())
7
```

items()

Returns internal list.

Return type
List[Tuple[Any, Union[int, float, complex, Expr]]]

make_hashable()

Return a hashable representation of the object.

Currently simply returns a stringified object.

Return type
str

map(*mapping*)

Updates list items right values in-place, using a mapping function provided.

Parameters
mapping (Callable[[Union[int, float, complex, Expr]], Union[int, float, complex, Expr]]) – A callable object which takes each original value and returns the corresponding new value.

Return type
LinearListCombiner

property n_symbols: int

Returns the number of free symbols in the object.

Return type
int

property num_spin_orbs: int

Return the number of spin-orbitals that this operator explicitly acts on.

Return type
int

operator_class

alias of *FermionOperator*

print_table()

Print internal list formatted as a table.

Return type
None

qubit_encode(*mapping=None*, *qubits=None*)

Performs qubit encoding (mapping), using provided mapping function, of the current *FermionOperatorList*.

Terms are treated and mapped independently.

Parameters

- **mapping** (QubitMapping, default: `None`) – Mapping class. Default mapping procedure is the Jordan-Wigner transformation.
- **qubits** (`Optional[List[Qubit]]`, default: `None`) – The qubit register. If left as None, a default register will be assumed if possible. See the mapping class documentation for further details.

Returns

QubitOperatorList – Mapped *QubitOperatorList*.

retrotterize (*new_trotter_number*, *initial_trotter_number=1*, *new_trotter_order=1*, *initial_trotter_order=1*, *constant=1.0*, *inner_coefficients=False*)

Retrotterize an expression given a OperatorList representing a product of exponentials.

This method assumes that the OperatorList represents a product of exponentials, with each Operator in the list corresponding to an exponent of a single exponential in the product. Scalar factors within the OperatorList are treated as scalar multipliers within each exponent.

The OperatorList is first untrotterized using the provided `initial_trotter_number` and `initial_trotter_order`, then subsequently Trotterized using the provided `new_trotter_number` and `new_trotter_order`. The returned OperatorList corresponds to the generated product of exponentials, in a similar manner to the original OperatorList.

Parameters

- **new_trotter_number** (`int`) – The desired number of Trotter steps in the final Trotter-Suzuki expansion.
- **initial_trotter_number** (`int`, default: `1`) – The number of Trotter steps in the original Trotter-Suzuki expansion.
- **new_trotter_order** (`int`, default: `1`) – The order of the original Trotter-Suzuki expansion used. Currently, only a first order (ABABAB…) expansion is supported.
- **initial_trotter_order** (`int`, default: `1`) – The desired order of the final Trotter-Suzuki expansion. Currently, only a first order (ABABAB…) expansion is supported.
- **constant** (`Union[float, complex]`, default: `1.0`) – An additional constant multiplier in the exponent.
- **inner_coefficients** (`bool`, default: `False`) – By default, generated scalar factors in each exponent are stored in the coefficients of the generated OperatorList, with the coefficient of each inner Operator unchanged. Set this to True to instead store all scalar factors as coefficients in each Operator, with the outer coefficients of the OperatorList left unchanged. See examples for a comparison.

Returns

`TypeVar(OperatorListT, bound= OperatorList)` – The exponential product retrotterized with the provided new Trotter number and order. Each element is an individual exponent.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> retrotterised = qol.retrotterize(new_trotter_number=4,initial_trotter_
↪number=2)
```

(continues on next page)

(continued from previous page)

```
>>> print(retrotterised)
0.25       [(1.0, X0 X1)],
0.25       [(1.0, Z0)],
0.25       [(1.0, X0 X1)],
0.25       [(1.0, Z0)],
0.25       [(1.0, X0 X1)],
0.25       [(1.0, Z0)],
0.25       [(1.0, X0 X1)],
0.25       [(1.0, Z0)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> retrotterised = qol.retrotterize(new_trotter_number=4,initial_trotter_
↪number=2,inner_coefficients=True)
>>> print(retrotterised)
0.5        [(0.5, X0 X1)],
0.5        [(0.5, Z0)],
0.5        [(0.5, X0 X1)],
0.5        [(0.5, Z0)],
0.5        [(0.5, X0 X1)],
0.5        [(0.5, Z0)],
0.5        [(0.5, X0 X1)],
0.5        [(0.5, Z0)]
```

reversed_order()

Reverses internal list order and returns it as a new object.

Return type
LinearListCombiner

simplify(**args*, ***kwargs*)

Simplifies expressions stored in left and right values of list items.

Parameters

- **args** (Any) – Args to be passed to *sympy.simplify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.simplify()*.

Returns
LinearListCombiner – Updated instance of LinearListCombiner.

split()

Generates pair objects from list items.

Return type
Iterator[LinearListCombiner]

sublist(*sublist_indices*)

Returns a new instance containing a subset of the terms of the original object.

Parameters
sublist_indices (list[int]) – List of indices of the operator list elements to constitute a new object.

Returns
TypeVar(OperatorListT, bound= OperatorList) – A new instance of OperatorList, being

a sublist of the original operator.

Raises

ValueError – If sublist_indices contains indices not contained in self, or self is empty.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6)
>>> op2 = QubitOperator("Z0", -1.6j)
>>> op3 = QubitOperator("Z1 Z2 Z3 Z5", -5.6j)
>>> long_operator = QubitOperatorList([(1, op1), (1, op2), (1, op3)])
>>> short_operator = long_operator.sublist([0, 2])
>>> print(short_operator)
1          [(4.6, X0 Y1 Z3)],
1          [(-5.6j, Z1 Z2 Z3 Z5)]
```

subs (*symbol_map*)

Returns a new objects with symbols substituted.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type

TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution (*symbol_map=None*)

Substitutes free symbols for numerical values according to a map.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) –

Return type

LinearListCombiner

sympify (**args*, ***kwargs*)

Sympifies left and right values of list items.

Replaces left and right values with their corresponding symbolic expressions.

Parameters

- **args** (Any) – Args to be passed to *sympy.sympify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.sympify()*.

Returns

LinearListCombiner – Updated instance of LinearListCombiner.

Raises

RuntimeError – when sympification fails.

trotterize_as_linear_combination (*trotter_number*, *trotter_order=1*, *constant=1.0*, *inner_coefficients=False*)

Trotterize an exponent linear combination of Operators.

This method assumes that the OperatorList represents the exponential of a linear combination of Operators, with each Operator within the OperatorList corresponding to a term in this linear combination. Trotterization is performed at the level of these Operators. The Operators contained within the returned OperatorList correspond to exponents within the Trotter sequence.

Parameters

- **trotter_number** (int) – The number of Trotter steps in the Trotter-Suzuki expansion.
- **trotter_order** (int, default: 1) – The order of the Trotter-Suzuki approximation to be used. Presently, only first-order (ABABAB…) is supported.
- **constant** (Union[float, complex], default: 1.0) – An additional constant multiplier in the exponent.
- **inner_coefficients** (bool, default: False) – By default, generated scalar factors in each exponent are stored in the coefficients of the generated OperatorList, with the coefficient of each inner Operator unchanged. Set this to True to instead store all scalar factors as coefficients in each Operator, with the outer coefficients of the OperatorList left unchanged. See examples for a comparison.

Returns

TypeVar(OperatorListT, bound= OperatorList) – A Trotterized form of the exponential product, where each element is an individual exponent.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1.,op1),(1.,op2)])
>>> result = qol.trotterize_as_linear_combination(2)
>>> print(result)
0.5       [(1.0, X0 X1)],
0.5       [(1.0, Z0)],
0.5       [(1.0, X0 X1)],
0.5       [(1.0, Z0)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1.,op1),(1.,op2)])
>>> result = qol.trotterize_as_linear_combination(2,inner_coefficients=True)
>>> print(result)
1.0       [(0.5, X0 X1)],
1.0       [(0.5, Z0)],
1.0       [(0.5, X0 X1)],
1.0       [(0.5, Z0)]
```

unsympify()

Unsympifies dicionary values.

Replaces symbolic expressions with their corresponding numeric values.

Returns

LinearListCombiner – Updated instance of LinearListCombiner.

Raises

TypeError – When unsympification fails.

untrotterize(*trotter_number*, *trotter_order=1*)

Reverse a Trotter-Suzuki expansion given an OperatorList representing a product of exponentials.

This method assumes that the OperatorList represents a product of exponentials, with each Operator in the list corresponding to an exponent of a single exponential in the product. Scalar factors within the OperatorList are treated as scalar multipliers within each exponent. An Operator corresponding to the exponent of a single, untrotterized exponential is returned.

Parameters

- **trotter_number** (int) – The number of Trotter steps within the Trotter expansion to be reversed.
- **trotter_order** (int, default: 1) – The order of the Trotter-Suzuki expansion used. Currently, only a first order (ABABAB…) expansion is supported.

Returns

TypeVar(OperatorT, bound= Operator) – The exponent of the untrotterised operator.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> untrotterised = qol.untrotterize(2)
>>> print(untrotterised)
(1.0, X0 X1), (1.0, Z0)
```

---

untrotterize_partitioned(*trotter_number*, *trotter_order=1*, *inner_coefficients=False*)

Reverse a Trotter-Suzuki expansion given a OperatorList representing a product of exponentials, maintaining separation of exponents.

This method assumes that the OperatorList represents a product of exponentials, with each Operator in the list corresponding to an exponent of a single exponential in the product. Scalar factors within the OperatorList are treated as scalar multipliers within each exponent. A OperatorList is returned wherein each term represents a single term in the exponent of the single, untrotterized exponential.

Parameters

- **trotter_number** (int) – The number of Trotter steps within the Trotter expansion to be reversed.
- **trotter_order** (int, default: 1) – The order of the Trotter-Suzuki expansion used. Currently, only a first order (ABABAB…) expansion is supported.
- **inner_coefficients** (bool, default: False) – By default, generated scalar factors in each exponent are stored in the coefficients of the generated OperatorList, with the coefficient of each inner Operator unchanged. Set this to True to instead store all scalar factors as coefficients in each Operator, with the outer coefficients of the OperatorList left unchanged. See examples for a comparison.

Returns

TypeVar(OperatorListT, bound= OperatorList) – The terms in the exponent of the untrotterised operator as a OperatorList.

Raises

ValueError – If the provided Trotter number is not compatible with the OperatorList.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> untrotterised = qol.untrotterize_partitioned(2)
>>> print(untrotterised)
1.0       [(1.0, X0 X1)],
1.0       [(1.0, Z0)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> untrotterised = qol.untrotterize_partitioned(2,inner_coefficients=True)
>>> print(untrotterised)
0.5       [(2.0, X0 X1)],
0.5       [(2.0, Z0)]
```

class FermionOperatorString (*initializer: tuple | List[Tuple[int, int]] | Tuple[Tuple[int, int]] | None = None*)

Bases: `tuple`

Handles a single fermionic string of creation and annihilation operators.

It is a tuple of tuples, each of which contains two integers, with the first indicating a spin orbital number, and the second being either 1 or 0, corresponding to creation and annihilation operation correspondingly. Defined to constitute a single term in FermionOperator.

Examples

```
>>> FermionOperatorString(((3, 1), (2, 0)))
((3, 1), (2, 0))
>>> FermionOperatorString((1, 1))
((1, 1),)
>>> print(FermionOperatorString(((3, 1), (2, 0))))
F3^ F2
```

FERMION_ANNIHILATION = 0

Integer used to indicate a fermion operator is an annihilation operator.

FERMION_CREATION = 1

Integer used to indicate a fermion operator is a creation operator.

apply_bra (*fock_state*, *power=1*)

Performs an operation on a bra `FermionState` state.

Parameters

- **fock_state** (*FermionState*) – Input fermion state (bra state).
- **power** (`int`, default: `1`) – Power of operation (how many times operator acts on a bra state).

Returns

FermionState – New bra state.

Examples

```
>>> f_op_string = FermionOperatorString.from_string("F1^ F0")
>>> bra = FermionState([0, 1])
>>> print(bra)
(1.0, [0, 1])
>>> print(f_op_string.apply_bra(bra))
(1.0, [1, 0])
>>> bra = FermionState([1, 0])
>>> print(f_op_string.apply_bra(bra))
(0)
```

apply_ket (*fock_state*, *power=1*)

Performs an operation on a ket `FermionState` state.

Parameters

- **fock_state** (*FermionState*) – Input fermion state (ket state).
- **power** (`int`, default: `1`) – Power of operation (how many times operator acts on a ket state).

Returns

FermionState – New ket state.

Examples

```
>>> f_op_string = FermionOperatorString.from_string("F1^ F0")
>>> ket = FermionState([1, 0])
>>> print(ket)
(1.0, [1, 0])
>>> print(f_op_string.apply_ket(ket))
(1.0, [0, 1])
>>> ket = FermionState([0, 1])
>>> print(f_op_string.apply_ket(ket))
(0)
```

apply_state (*fock_state*, *state_type*, *power*)

Implements a general operation on a `FermionState` (bra or ket) object.

Parameters

- **fock_state** (*FermionState*) – Input fermion state.
- **state_type** (`StateType`) – Enum defining whether the state is bra or ket.
- **power** (`int`) – Power of operation.

Returns

FermionState – New state object.

count (*value*, */*)

Return number of occurrences of value.

dagger ()

Performs a conjugation operation on a fermion creation-annihilation operator string.

Reverses order and swaps creation and annihilation labels.

Examples

```
>>> f_op_string = FermionOperatorString.from_string("F3^ F1^ F2 F0")
>>> print(f_op_string.dagger())
F0^ F2^ F1 F3
```

Return type
FermionOperatorString

classmethod from_string(*string*)

Generates a class instance from a string.

Parameters
string (str) – Formatted string input.

Returns
FermionOperatorString – String of fermion creation/annihilation operators corresponding to a valid input python string.

Examples

```
>>> FermionOperatorString.from_string("F3^ F2")
((3, 1), (2, 0))
>>> FermionOperatorString.from_string("3^ 2")
((3, 1), (2, 0))
```

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

is_empty()

Checks if object is empty.

Return type
bool

is_particle_conserving()

Checks if operator string is particle-conserving.

Examples

```
>>> FermionOperatorString.from_string("F3^ F2").is_particle_conserving()
True
>>> FermionOperatorString.from_string("F3^ F2 F0").is_particle_conserving()
False
```

Return type
bool

items()

Returns current class instance.

Return type
Sequence[Tuple[int, int]]

to_latex(*operator_symbol='a'*, *index_to_latex=None*)

Generate a LaTeX representation of the operator string.

Parameters

- **operator_symbol** (str, default: "a") – Symbol to use for the creation/annihilation operator.
- **index_to_latex** (Optional[Callable[[int], str]], default: None) – Function mapping a spin-orbital index to a latex string.

Returns
str – LaTeX compilable equation string.

Examples

```
>>> fos = FermionOperatorString(((0, 1), (0, 0)))
>>> print(fos.to_latex())
a_{0}^{\dagger} a_{0}
>>> fos = FermionOperatorString(((0,1), (1, 1), (4, 5)))
>>> print(fos.to_latex(operator_symbol="c"))
c_{0}^{\dagger} c_{1}^{\dagger} c_{4}
>>> fos = FermionOperatorString.from_string("F5^ F6^ F3 F4")
>>> print(fos.to_latex(operator_symbol="f"))
f_{5}^{\dagger} f_{6}^{\dagger} f_{3} f_{4}
```

class OrbitalOptimizer(*v_init=None*, *occ=None*, *split_rotation=False*, *functional=None*, *minimizer=None*, *point_group=None*, *orbital_irreps=None*, *reduce_free_parameters=True*)

Bases: *OrbitalTransformer*

Handles minimization of a functional of molecular orbital coefficients.

Parameters

- **v_init** (Optional[array], default: None) – Initial orbital coefficients.
- **occ** (Optional[array], default: None) – Molecular orbital occupations (if split_rotation=True).
- **split_rotation** (bool, default: False) – If True, do not allow mixing between occupied and virtual orbitals.
- **functional** (Optional[Callable], default: None) – The objective function to be minimized.
- **minimizer** (Optional[GeneralMinimizer], default: None) – An InQuanto minimizer.
- **point_group** (Union[*PointGroup*, str, None], default: None) – If passed, symmetry information will be used to reduce free parameters.
- **orbital_irreps** (Optional[list], default: None) – Orbital irreducible representations, needed if point_group is passed.

- **reduce_free_parameters** (bool, default: True) – If True, the objective function will be simplified to depend on fewer parameters.

static compute_unitary(*v_init=None*, *v_final=None*)

Computes the unitary relating column vectors v_init and v_final.

Computes the matrix $U = v_{init}^{-1} v_{final}$

Parameters

- **v_init** (Optional[ndarray], default: None) – Initial orbitals.
- **v_final** (Optional[ndarray], default: None) – Final orbitals.

Returns

ndarray – Unitary matrix relating initial and final orbitals.

construct_random_variables(*v=None*, *low=-0.1*, *high=0.1*, *seed=None*)

Constructs dim * (dim - 1)/2 variables sampled from uniform distribution.

Uniform distribution is specified by given high, low and seed.

Parameters

- **v** (Optional[array], default: None) – Orbitals to be rotated.
- **low** (float, default: -0.1) – Lower bound of uniform distribution domain.
- **high** (float, default: 0.1) – Upper bound of uniform distribution domain.
- **seed** (Optional[int], default: None) – Random number generator seed.

Returns

ndarray – Array of random initial variables.

generate_report()

Generates a summarising report.

Returns

Dict – Results of the optimization.

static gram_schmidt(*v*, *overlap=None*)

Orthogonalizes column vectors in v using Gram-Schmidt algorithm with respect to an overlap matrix.

Parameters

- **v** (ndarray) – Orbitals/vectors to be orthonormalized.
- **overlap** (Optional[ndarray], default: None) – Overlap matrix.

Returns

ndarray – Orthogonalized vectors.

map_variables_to_rotation_matrix(*variables=None*)

Maps N * (N - 1)/2 variables to a unitary matrix.

Parameters

variables (Optional[array], default: None) – Variables to be mapped to a unitary matrix.

Returns

ndarray – Unitary rotation matrix.

map_variables_to_skew_matrix(*variables=None*)

Constructs a skew-symmetric matrix from N * (N-1)/2 variables where N is the dimension of the skew matrix.

Parameters

variables (Optional[array], default: None) – Variables to be mapped to a skew-symmetric matrix.

Returns

ndarray – Skew symmetric matrix.

optimize(*orb_init=None*, *initial_variables=None*, *functional=None*, *random_initial_variables=False*)

Minimizes a functional which depends on orbital coefficients.

Parameters

- **orb_init** (Optional[List[float]], default: None) – Initial orbitals.
- **initial_variables** (Optional[List[int]], default: None) – Initial guess variables.
- **functional** (Optional[Callable], default: None) – The functional to minimize, must be callable and a function of only MO coefficients.
- **random_initial_variables** (bool, default: False) – Should starting variables be randomised.

Returns

Tuple[List, ndarray, float] – New MO coefficients, unitary for orb_init -> new orbitals, final value of objective function.

static orthonormalize(*v*, *overlap=None*)

Finds the closest orthonormal set of vectors with respect to overlap matrix.

Parameters

- **v** (ndarray) – Column vectors/molecular orbitals.
- **overlap** (Optional[ndarray], default: None) – Overlap matrix.

Returns

ndarray – Orthonormalised array of column vectors.

transform(*v*, *tu*)

Apply unitary, tu, to an array of column vectors.

Parameters

- **v** (ndarray) – Array of column vectors.
- **tu** (ndarray) – Unitary matrix.

Returns

ndarray – Transformed set of column vectors in numpy array.

class OrbitalTransformer(*v_init=None*, *v_final=None*)

Bases: object

Class holding convenience functions for manipulating molecular orbitals.

Initialised with initial and final orbital arrays.

Parameters

- **v_init** (Optional[ndarray], default: None) – Initial orbitals.

- **v_final** (Optional[ndarray], default: None) – Final orbitals.

static compute_unitary(*v_init=None*, *v_final=None*)

Computes the unitary relating column vectors v_init and v_final.

Computes the matrix $U = v_{init}^{-1} v_{final}$

Parameters

- **v_init** (Optional[ndarray], default: None) – Initial orbitals.
- **v_final** (Optional[ndarray], default: None) – Final orbitals.

Returns
ndarray – Unitary matrix relating initial and final orbitals.

static gram_schmidt(*v*, *overlap=None*)

Orthogonalizes column vectors in v using Gram-Schmidt algorithm with respect to an overlap matrix.

Parameters

- **v** (ndarray) – Orbitals/vectors to be orthonormalized.
- **overlap** (Optional[ndarray], default: None) – Overlap matrix.

Returns
ndarray – Orthogonalized vectors.

static orthonormalize(*v*, *overlap=None*)

Finds the closest orthonormal set of vectors with respect to overlap matrix.

Parameters

- **v** (ndarray) – Column vectors/molecular orbitals.
- **overlap** (Optional[ndarray], default: None) – Overlap matrix.

Returns
ndarray – Orthonormalised array of column vectors.

transform(*v*, *tu*)

Apply unitary, tu, to an array of column vectors.

Parameters

- **v** (ndarray) – Array of column vectors.
- **tu** (ndarray) – Unitary matrix.

Returns
ndarray – Transformed set of column vectors in numpy array.

class QubitOperator(*data=None*, *coeff=1.0*)

Bases: QubitPauliOperator, Operator

InQuanto's representation of a linear operator acting on a 2^N dimensional Hilbert space with Pauli operators.

Can be constructed from a string, a list or a tuple of tuples (containing a qubit index (integer) and a string with a Pauli gate symbol), a *QubitOperatorString* together with a single coefficient, or a dictionary with each item containing a *QubitOperatorString* and a coefficient.

Parameters

- **data** (Union[str, Iterable[Tuple[int, str]], Dict[*QubitOperatorString*, Union[int, float, complex, Expr]], *QubitOperatorString*, None], default: None) – Data defined as a string "X0 Y1", iterable of tuples ((0, 'Y'), (1, 'X')), *QubitOperatorString*, or as a dictionary of *QubitOperatorString* and CoeffType objects.
- **coeff** (Union[int, float, complex, Expr], default: 1.0) – Coefficient attached to data.

Example

```
>>> op0 = QubitOperator("X0 Y1 Z3", 4.6)
>>> print(op0)
(4.6, X0 Y1 Z3)
```

```
>>> op1 = QubitOperator(((0, "X"), (1, "Y"), (3, "Z")), 4.6)
>>> print(op1)
(4.6, X0 Y1 Z3)
```

```
>>> op2 = QubitOperator([(0, "X"), (1, "Y"), (3, "Z")], 4.6)
>>> print(op2)
(4.6, X0 Y1 Z3)
```

```
>>> qs = QubitOperatorString.from_string("X0 Y1 Z3")
>>> op3 = QubitOperator(qs, 4.6)
>>> print(op3)
(4.6, X0 Y1 Z3)
```

```
>>> qs0 = QubitOperatorString.from_string("X0 Y1 Z3")
>>> qs1 = QubitOperatorString.from_tuple([(0, Pauli.Y), (1, Pauli.X)])
>>> dictionary = {qs0: 4.6, qs1: -1.7j}
>>> op4 = QubitOperator(dictionary)
>>> print(op4)
(4.6, X0 Y1 Z3), (-1.7j, Y0 X1)
```

class TrotterizeCoefficientsLocation(*value*)

Bases: str, Enum

Determines where coefficients will be stored upon performing Trotterization.

INNER = 'inner'

All coefficients will be stored in the "inner" coefficients, within the component QubitOperators in the Trotterized result.

MIXED = 'mixed'

The Trotter step factor will be stored in the "outer" coefficients, whereas the original coefficients of the original Operator will remain in the component Operators.

OUTER = 'outer'

All coefficients will be stored in the "outer" coefficients, the coefficients stored directly in the generated OperatorList.

property all_qubits: Set[Qubit]

The set of all qubits the operator ranges over (including qubits that were provided explicitly as identities)

Return type
Set[Qubit]

Type
return

anticommutator(*other_operator*, *abs_tol=None*)

Calculates the anticommutator with another *QubitOperator*, within a tolerance.

Parameters

- **other_operator** (*QubitOperator*) – The other *QubitOperator*.
- **abs_tol** (Optional[float], default: None) – Threshold below which terms are deemed negligible.

Returns
QubitOperator – The anticommutator of the two operators.

anticommutes_with(*other_operator*, *abs_tol=1e-10*)

Calculates whether operator anticommutes with another *QubitOperator*, within a tolerance.

If both operators are single Pauli strings, we use tket's .commutes_with() method and flip the result. Otherwise, it calculates the whole anticommutator and checks if it is zero.

Parameters

- **other_operator** (*QubitOperator*) – The other *QubitOperator*.
- **abs_tol** (float, default: 1e-10) – Threshold below which terms are deemed negligible.

Returns
bool – True if operators anticommute, within tolerance, otherwise False.

antihermitian_part()

Return the anti-Hermitian (all imaginary-coefficient terms) part of the original *QubitOperator*.

In case the original *QubitOperator* object contains symbolic coefficients that do not have an associated type, those will be cast to the imaginary Expr type.

Examples

```
>>> qo = QubitOperator.from_string("(1.0, X0 Y1), (0.1j, Y0 X1), (0.5 + 0.2j,␣
↪Z0 Z1)")
>>> print(qo.antihermitian_part())
(0.1j, Y0 X1), (0.2j, Z0 Z1)
>>> a = Symbol('a', real=True)
>>> b = Symbol('b', imaginary=True)
>>> c = Symbol('c')
>>> p_str_a = QubitOperatorString.from_string("X0 Y1")
>>> p_str_b = QubitOperatorString.from_string("Y0 X1")
>>> p_str_c = QubitOperatorString.from_string("Z0 Z1")
>>> qo = QubitOperator({p_str_a: a, p_str_b: b, p_str_c: c})
>>> print(qo.antihermitian_part())
(b, Y0 X1), (1.0*I*im(c), Z0 Z1)
```

Return type
QubitOperator

approx_equal_to(*other*, *abs_tol=1e-10*)

Checks if object's dictionary values are numerically identical to the other object values.

Parameters

- **other** (LinearDictCombiner) – Object to compare to.
- **abs_tol** (float, default: 1e-10) – Threshold of comparing numeric values.

Raises

TypeError – When comparison of two values can't be done due to types mismatch.

Return type

bool

approx_equal_to_by_random_subs(*other*, *order=1*, *abs_tol=1e-10*)

Checks if object's dictionary values are numerically identical to the other object values.

Symbols contained in the difference of the two objects, if any, are substituted by random numeric values prior to norm check.

Parameters

- **other** (LinearDictCombiner) – Object to compare to.
- **order** (int, default: 1) – Parameter specifying the norm formula (see numpy.linalg.norm documentation).
- **abs_tol** (float, default: 1e-10) – Threshold vs which the norm of the difference is checked.

Return type

bool

as_scalar(*abs_tol=None*)

If the operator is a sum of identity terms or zero, return the sum of the coefficients, otherwise return None.

Note that this does not perform combination on terms and will return zero only if all coefficients are zero.

Parameters

abs_tol (Optional[float], default: None) – Tolerance for checking if coefficients are zero. Set to None to test using a standard python == comparison.

Returns

Union[float, complex, None] – The operator as a scalar if it can be represented as such, otherwise None.

clone()

Performs shallow copy of the object.

Return type

TypeVar(SYMBOLICTYPE, bound= Symbolic)

property coefficients: List[int | float | complex | Expr]

Returns dictionary values.

Return type

List[Union[int, float, complex, Expr]]

commutator(*other_operator*, *abs_tol=None*)

Calculate the commutator with another operator.

Computes commutator. Small terms in the result may be discarded.

Parameters

- **other_operator** (*QubitOperator*) – The other *QubitOperator*.
- **abs_tol** (Optional[float], default: None) – Threshold below which terms are discarded. Set to a negative value to skip.

Returns

QubitOperator – The commutator.

commutes_with(*other_operator*, *abs_tol=1e-10*)

Calculates whether operator commutes with another *QubitOperator*, within a tolerance.

If both operators are single Pauli strings, we use tket. Otherwise, it calculates the whole commutator and checks if it is zero.

Parameters

- **other_operator** (*QubitOperator*) – The other *QubitOperator*.
- **abs_tol** (float, default: 1e−10) – Threshold below which terms are deemed negligible.

Returns

bool – True if operators commute, within tolerance, otherwise False.

compress(*abs_tol=1e-10*, *symbol_sub_type=CompressSymbolSubType.NONE*)

Adapted from pytket.QubitPauliOperator to account for non-sympy coefficients.

Parameters

- **abs_tol** (float, default: 1e−10) – The threshold below which to remove values.
- **symbol_sub_type** (CompressSymbolSubType, default: CompressSymbolSubType.NONE) – Defines the behaviour for dealing with symbolic expressions in coefficients. If “none”, symbolic expressions are left intact. If “unity”, substitutes all free symbols with 1, and removes all imaginary and real components below tolerance. If “random”, substitutes all free symbols with a random number between 0 and 1, and removes imaginary and real components below tolerance.

Warning: When :code: *symbol_sub_type != “none”*, this method assumes significant expression structure is known a priori, and is best suited to operators which have simple product expressions, such as excitation operators for VQE ansätze and digital quantum simulation. Otherwise, it may remove terms relevant to computation. Each expression is of the form $f(a_1, a_2,$
$ldots, a_n)$ for some symbols a_i. $|f(a_1, a_2,$
$ldots, a_n)|$ is assumed to monotonically increase in both real and imaginary components for all a_i
$in[0, 1]$.

Return type

None

copy()

Performs deep copy of the object.

Return type

TypeVar(SYMBOLICTYPE, bound= Symbolic)

dagger()

Return the Hermitian conjugate of *QubitOperator*.

Return type
QubitOperator

df()

Returns a Pandas DataFrame object of the dictionary.

Return type
DataFrame

dot_state(*state*, *qubits=None*)

Calculate the result of operating on a given qubit state.

Can accept right-hand state as a QubitState, QubitStateString or a numpy.ndarray. In the former two cases, we maintain a symbolic representation of both operator and state, and each Pauli is implemented in sequence. Resultant states are returned as QubitState. This should support sympy parametrised states & operators, but the use of parametrised states & operators is untested. In this case, the optional qubits parameter is ignored.

For a numpy.ndarray, we hand off to pytket's QubitPauliOperator.dot_state() method - this should be faster for dense states, but slower for sparse ones. Here, a register of qubits may be specified in the qubits parameter to determine the meaning of the indices of the provided state vector. From the pytket documentation:

- When qubits is an explicit list, the qubits are ordered with qubits[0] as the most significant qubit for indexing into state.
- If None, qubits sequentially indexed from 0 in the default register and ordered by ILO-BE so Qubit(0) is the most significant.

Parameters

- **state** (Union[*QubitState*, *QubitStateString*, ndarray]) – Input qubit state to operated on.
- **qubits** (Optional[List[Qubit]], default: None) – For ndarray input, determines sequencing of qubits in the state, if not mapped to the default register. Ignored for other input types.

Returns
Union[*QubitState*, ndarray] – Output state.

empty()

Checks if dictionary is empty.

Return type
bool

evalf(**args*, ***kwargs*)

Evaluates symbolic expressions stored in dict values and replaces them with the results.

Parameters

- **args** (Any) – Args to be passed to *sympy.evalf()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.evalf()*.

Returns
LinearDictCombiner – Updated instance of LinearDictCombiner.

exponentiate_commuting_operator(*additional_exponent=1.0*, *check_commuting=True*)

Exponentiate a QubitOperator where all terms commute, returning as a product of operators.

As all terms are mutually commuting, exponentiation reduces to a product of exponentials of individual terms (i.e. $e^{\sum_i P_i} = \prod_i e^{P_i}$). Each individual exponential can further be expanded trigonometrically. While storing these as a product is efficient, expanding the product will result in an exponential number of terms, and thus this method returns the result in factorised form, as a QubitOperatorList.

Parameters

- **additional_exponent** (complex, default: 1.0) – Optional additional factor in exponent.
- **check_commuting** (bool, default: True) – Set to False to skip checking whether all terms commute.

Returns

QubitOperatorList – The exponentiated operator in factorised form.

Raises

ValueError – If commutativity checking is performed and the operator is not a commuting set of terms.

exponentiate_single_term(*additional_exponent=1.0*, *coeff_cutoff=1e-14*)

Exponentiates a single weighted Pauli string through trigonometric expansion.

This will except if the operator contains more than one term. It will attempt to maintain single term if rotation is sufficiently close to an integer multiple of pi/2. Set coeff_cutoff to None to disable this behaviour.

Parameters

- **additional_exponent** (complex, default: 1.0) – Optional additional factor in exponent.
- **coeff_cutoff** (Optional[float], default: 1e-14) – If a Pauli string is weighted by an integer multiple of pi/2 and exponentiated, the resulting expansion will have a single Pauli term (as opposed to two). If this parameter is not None, it will be used to determine a threshold for cutting off negligible terms in the trigonometric expansion to avoid floating point errors resulting in illusory growth in the number of terms. Set to None to disable this behaviour.

Returns

QubitOperator – The exponentiated operator.

Raises

ValueError – If the operator is not a single term.

free_symbols()

Returns the free symbols in the coefficient values.

free_symbols_ordered()

Returns the free symbols in the dict, ordered alphabetically.

Returns

SymbolSet – Ordered set of symbols.

classmethod from_list(*pauli_list*)

Construct a QubitPauliOperator from a serializable JSON list format, as returned by QubitPauliOperator.to_list()

Returns

New QubitPauliOperator instance.

Return type
QubitPauliOperator

Parameters
pauli_list (List[Dict[str, Any]]) –

classmethod from_string (*input_string*)

Constructs a child class instance from a string.

Parameters
input_string (str) – String in the format `coeff1 [(coeff1_1, term1_1), ..., (coeff1_n, term1_n)], ..., coeffn [(coeffn_1, termn_1), ...]`.

Returns
Child class object.

get (*key*, *default*)

Parameters

- **key** (QubitPauliString) –
- **default** (Union[int, float, complex, Expr]) –

Return type
Union[int, float, complex, Expr]

hermitian_factorisation ()

Returns a tuple of the real and imaginary parts of the original *QubitOperator*.

For example, both P and Q from $O = P + iQ$.

In case the original *QubitOperator* object contains symbolic coefficients that do not have an associated type, those will be cast into both the real and imaginary Expr types and assigned to both objects (imaginary component will be multiplied by $-I$ in order to return its real part).

Returns
Tuple[*QubitOperator*, *QubitOperator*] – Real and imaginary parts of the qubit operator.

Examples

```
>>> qo = QubitOperator.from_string("(1.0, X0 Y1), (0.1j, Y0 X1), (0.5 + 0.2j,␣
↪Z0 Z1)")
>>> re_qo, im_qo = qo.hermitian_factorisation()
>>> print(re_qo)
(1.0, X0 Y1), (0.5, Z0 Z1)
>>> print(im_qo)
(0.1, Y0 X1), (0.2, Z0 Z1)
>>> a = Symbol('a', real=True)
>>> b = Symbol('b', imaginary=True)
>>> c = Symbol('c')
>>> p_str_a = QubitOperatorString.from_string("X0 Y1")
>>> p_str_b = QubitOperatorString.from_string("Y0 X1")
>>> p_str_c = QubitOperatorString.from_string("Z0 Z1")
>>> qo = QubitOperator({p_str_a: a, p_str_b: b, p_str_c: c})
>>> re_qo, im_qo = qo.hermitian_factorisation()
>>> print(re_qo)
(a, X0 Y1), (re(c), Z0 Z1)
```

(continues on next page)

(continued from previous page)

```
>>> print(im_qo)
(-I*b, Y0 X1), (im(c), Z0 Z1)
```

hermitian_part()

Return the Hermitian (all real-coefficient terms) part of the original *QubitOperator*.

In case the original *QubitOperator* object contains symbolic coefficients that do not have an associated type, those will be cast to the real Expr type.

Examples

```
>>> qo = QubitOperator.from_string("(1.0, X0 Y1), (0.1j, Y0 X1), (0.5 + 0.2j,␣
↪Z0 Z1)")
>>> print(qo.hermitian_part())
(1.0, X0 Y1), (0.5, Z0 Z1)
>>> a = Symbol('a', real=True)
>>> b = Symbol('b', imaginary=True)
>>> c = Symbol('c')
>>> p_str_a = QubitOperatorString.from_string("X0 Y1")
>>> p_str_b = QubitOperatorString.from_string("Y0 X1")
>>> p_str_c = QubitOperatorString.from_string("Z0 Z1")
>>> qo = QubitOperator({p_str_a: a, p_str_b: b, p_str_c: c})
>>> print(qo.hermitian_part())
(a, X0 Y1), (re(c), Z0 Z1)
```

Return type

QubitOperator

classmethod identity()

Return an identity operator.

Examples

```
>>> print(QubitOperator.identity())
(1.0, )
```

Return type

QubitOperator

is_all_coeff_complex()

Check if all coefficients are complex values.

Warning: Returns None if there is a free symbol in a coefficient.

Return type

bool

is_all_coeff_imag()

Check if all coefficients are complex values.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type
bool

is_all_coeff_real()

Check if all coefficients are real values.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type
bool

is_all_coeff_symbolic()

Check if all coefficients contain free symbols.

Return type
bool

is_antihermitian()

Check if operator is antihermitian.

Check is performed by taking the Hermitian conjugate of operator and testing for opposite equality.

Returns
bool – True if antihermitian, False otherwise.

is_any_coeff_complex()

Check if any coefficient is a complex value.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type
bool

is_any_coeff_imag()

Check if any coefficient is an imaginary value.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type
bool

is_any_coeff_real()

Check if any coefficient is a real value.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type
bool

is_any_coeff_symbolic()

Check if any coefficient contains a free symbol.

Return type
bool

is_commuting_operator()

Return True if every term in operator commutes with every other term, otherwise False.

Return type
bool

is_hermitian()

Check if operator is Hermitian.

Check is performed by taking the Hermitian conjugate of operator and testing for equality.

Returns
bool – True if Hermitian, False otherwise.

is_parallel_with(*other*, *abs_tol=1e-10*)

Returns True if other is parallel with this (i.e. a scalar multiple of this), otherwise False.

Parameters

- **other** (LinearDictCombiner) – The other object to compare against
- **abs_tol** (Optional[float], default: 1e-10) – Tolerance threshold for comparison. Set to None to test for exact equivalence.

Returns
bool – True if other is parallel with this, otherwise False.

is_self_inverse(*abs_tol=1e-10*)

Check if operator is its own inverse.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold for comparison with identity.

Returns
bool – True if self-inverse, False otherwise.

is_unit_1norm(*abs_tol=1e-10*)

Returns True if operator has unit 1-norm, else False.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold for comparison with unity.

Return type
bool

is_unit_2norm(*abs_tol=1e-10*)

Returns True if operator has unit 1-norm, else False.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold for comparison with unity.

Return type
bool

is_unit_norm(*order=2*, *abs_tol=1e-10*)

Returns True if operator has unit p-norm, else False.

Parameters

- **order** (int, default: 2) – Norm order.
- **abs_tol** (float, default: 1e-10) – Tolerance threshold for comparison with unity.

Raises
ValueError – Coefficients contain free symbols.

Return type
bool

is_unitary(*abs_tol=1e-10*)

Check if operator is unitary.

Checking is performed by multiplying the operator by its Hermitian conjugate and comparing against the identity.

Parameters
abs_tol – Tolerance threshold for comparison with identity.

Returns
bool – True if unitary, False otherwise.

items()

Returns dictionary items.

Return type
ItemsView[Any, Union[int, float, complex, Expr]]

static key_from_str(*key_str*)

Returns a *QubitOperatorString* instance initialised from the input string.

Parameters
key_str (str) – Input python string.

Returns
QubitOperatorString – Operator string initialised from input.

list_class

alias of *QubitOperatorList*

make_hashable()

Return a hashable representation of the object.

Currently simply returns a stringified object.

Return type
str

map(*mapping*)

Updates dictionary values, using a mapping function provided.

Parameters
mapping (Callable[[Union[int, float, complex, Expr]], Union[int, float, complex, Expr]]) – Mapping function to update the dict.

Returns
LinearDictCombiner – None.

property n_symbols: int

Returns the number of free symbols in the object.

Return type
int

norm_coefficients(*order=2*)

Returns the p-norm of the coefficients.

Parameters
order (int, default: 2) – Norm order.

Return type
Union[complex, float]

normalized(*norm_value=1.0*, *norm_order=2*)

Returns a copy of this object with normalised coefficients.

Parameters

- **norm_value** (float, default: 1.0) – The desired norm of the returned operator.
- **norm_order** (int, default: 2) – The order of the norm to be used.

Returns
LinearDictCombiner – A copy of the object with coefficients normalised to the desired value.

pad(*register_qubits=None*, *zero_to_max=False*)

Modify *QubitOperator* in-place by replacing implicit identities with explicit identities.

By default, this will assume a minimal register - i.e. the register contains only qubits acted on by any term in the *QubitOperator*. A specific register of qubits may be provided by setting the register_qubits parameter. This must contain all qubits acted on by the *QubitOperator*. Alternatively, zero_to_max may be set to True in order to assume that the qubit register is indexed 0-N, where N is the highest integer indexed qubit in the original *QubitOperator*. These modes of operation are incompatible and this method will except if zero_to_max is set to True and register_qubits is provided. See *padded()* for a non-in-place version of this method.

Parameters

- **register_qubits** (Optional[Iterable[Qubit]], default: None) – A qubit register used to determine which padding identities will be added.
- **zero_to_max** (bool, default: False) – Set to True to assume a 0-N indexed qubit register as described above.

Raises

- **PaddingIncompatibleArgumentsError** – If register_qubits has been provided while zero_to_max is set to True.

- **PaddingInferenceError** – If `zero_to_max` is set to True and maximum qubit index cannot be inferred, for instance by non-integer labelled qubits.
- **PaddingInvalidRegisterError** – If *QubitOperator* acts on qubits not in provided register.

Examples

```
>>> qs = QubitOperator("X0") + QubitOperator("X1")
>>> qs.pad()
>>> print(qs)
(1.0, X0 I1), (1.0, I0 X1)
```

```
>>> qs = QubitOperator("X0")
>>> qs.pad([Qubit(0),Qubit(1)])
>>> print(qs)
(1.0, X0 I1)
```

```
>>> qs = QubitOperator("X1")
>>> print(qs.padded(zero_to_max=True))
(1.0, I0 X1)
```

Return type
None

padded (*register_qubits=None*, *zero_to_max=False*)

Return a copy of the *QubitOperator* with implicit identities replaced by explicit identities.

By default, this will assume a minimal register - i.e. the register contains only qubits acted on by any term in the *QubitOperator*. A specific register of qubits may be provided by setting the `register_qubits` parameter. This must contain all qubits acted on by the *QubitOperator*. Alternatively, `zero_to_max` may be set to True in order to assume that the qubit register is indexed 0-N, where N is the highest integer indexed qubit in the original *QubitOperator*. These modes of operation are incompatible and this method will except if `zero_to_max` is set to True and `register_qubits` is provided. See *pad()* for an in-place version of this method.

Parameters

- **register_qubits** (Optional[Iterable[Qubit]], default: None) – A qubit register used to determine which padding identities will be added.
- **zero_to_max** (bool, default: False) – Set to True to assume a 0-N indexed qubit register as described above.

Raises

- **PaddingIncompatibleArgumentsError** – If register_qubits has been provided while `zero_to_max` is set to True.
- **PaddingInferenceError** – If `zero_to_max` is set to True and maximum qubit index cannot be inferred, for instance by non-integer labelled qubits.
- **PaddingInvalidRegisterError** – If *QubitOperator* acts on qubits not in provided register.

Returns
QubitOperator – Modified *QubitOperator*.

Examples

```
>>> qs = QubitOperator("X0") + QubitOperator("X1")
>>> qs_padded = qs.padded()
>>> print(qs_padded)
(1.0, X0 I1), (1.0, I0 X1)
```

```
>>> qs = QubitOperator("X0")
>>> qs_padded = qs.padded([Qubit(0),Qubit(1)])
>>> print(qs_padded)
(1.0, X0 I1)
```

```
>>> qs = QubitOperator("X1")
>>> qs_padded = qs.padded(zero_to_max=True)
>>> print(qs_padded)
(1.0, I0 X1)
```

property pauli_strings: List[*QubitOperatorString*]

Return the Pauli strings within the operator sum as a list.

Return type
List[*QubitOperatorString*]

print_table()

Print dictionary formatted as a table.

Return type
NoReturn

qubitwise_anticommutes_with(*other_operator*)

Calculates whether two single-term QubitOperators qubit-wise anticommute.

Two Pauli strings qubit-wise anticommute if every Pauli acting on a given qubit anticommutes with the Pauli acting on the same qubit in the other string. This necessitates that the Paulis are different, and both non-identity, for every qubit.

This method is currently not defined for QubitOperators consisting of multiple terms.

Parameters
other_operator (*QubitOperator*) – The other single-term *QubitOperator*.

Returns
bool – True if the operators qubit-wise anticommute, otherwise False.

Raises
ValueError – If either operator consists of more than a single term.

qubitwise_commutes_with(*other_operator*)

Calculates whether two single-term QubitOperators qubit-wise commute.

Two Pauli strings qubit-wise commute if every Pauli acting on a given qubit commutes with the Pauli acting on the same qubit in the other string. This necessitates that either the Paulis are the same, or that at least one is an identity.

This method is currently not defined for QubitOperators consisting of multiple terms.

Parameters
other_operator (*QubitOperator*) – The other single-term *QubitOperator*.

Returns
bool – True if the operators qubit-wise commute, otherwise False.

Raises
ValueError – If either operator consists of more than a single term.

remove_global_phase(*phase=0.0*)

Returns a copy with a global phase applied such that the first element has the desired phase.

Parameters
phase (float, default: 0.0) – The phase to yield on the first element, in half-turns (i.e. multiples of pi).

Returns
LinearDictCombiner – A copy of the object with the desired global phase applied.

reversed_order()

Reverses the order of terms and returns it as a new object.

Return type
LinearDictCombiner

simplify(**args*, ***kwargs*)

Simplifies expressions stored in dictionary values.

Parameters

- **args** (Any) – Args to be passed to *sympy.simplify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.simplify()*.

Returns
LinearDictCombiner – Updated instance of LinearDictCombiner.

split()

Generates pair objects from dictionary items.

Return type
Iterator[LinearDictCombiner]

state_expectation(*state*, **args*, ***kwargs*)

Calculate expectation value of operator with input state.

Can accept right-hand state as a QubitState or a numpy.ndarray. In the first case, we maintain a symbolic representation of both operator and state, and each Pauli is implemented in sequence. This should support sympy parametrised states & operators, but the use of parametrised states & operators is untested. For a numpy.ndarray, we hand off to pytket's QubitPauliOperator.dot_state() method and return a numpy.ndarray - this should be faster for dense states, but slower for sparse ones.

Parameters

- **state** (Union[*QubitState*, ndarray]) – The state to be acted upon.
- ***args** – Additional arguments passed to pytket.utils.operators.QubitPauliOperator.state_expectation().
- ****kwargs** – Additional keyword arguments passed to pytket.utils.operators.QubitPauliOperator.state_expectation().

Returns
complex – Expectation value of *QubitOperator*.

subs (*symbol_map*)

In-place substitution for symbolic expressions.

Iterates through each Symbol and performs the substitution.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Union[str, Symbol], Union[float, complex, Expr, None]]]) – A map from SymPy symbols to SymPy expressions, floating-point or complex values.

Returns

QubitOperator – The operator after substitution.

symbol_substitution (*symbol_map=None*)

Substitutes free symbols for numerical values according to a map.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) –

Return type

LinearDictCombiner

sympify (**args*, ***kwargs*)

Sympifies dictionary values.

Replaces values with their corresponding symbolic expressions.

Parameters

- **args** (Any) – Args to be passed to *sympy.sympify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.sympify()*.

Returns

LinearDictCombiner – Updated instance of LinearDictCombiner.

Raises

RuntimeError – When sympification fails.

symplectic_representation (*qubits=None*)

Generate the symplectic representation of the operator.

This is a binary M*2N matrix where M is the number of terms and N is the number of qubits. In the leftmost half of the matrix, element (i,j) is True where qubit j is acted on by an X or a Y in term i, and otherwise is False. In the rightmost half, element (i,j) is True where qubit (j-num_qubits) is acted on by a Y or a Z in term i, and otherwise is False.

Notes

If qubits is not specified, a minimal register will be assumed and columns will be assigned to qubits in ascending numerical order by index.

Parameters

qubits (Optional[List[Qubit]], default: None) – A register of qubits. If not provided, a minimal register is assumed.

Returns

ndarray – The symplectic form of this operator

property terms: List[Any]

Returns dictionary keys.

Return type
List[Any]

to_latex(*imaginary_unit='\\\\text{i}'*, ***kwargs*)

Generate a LaTeX representation of the operator.

Parameters

- **imaginary_unit** (str, default: r"\\text{i}") – Symbol to use for the imaginary unit.
- ****kwargs** – Keyword arguments passed to the *to_latex()* method of component operator strings (*FermionOperatorString* or *QubitOperatorString*).

Returns
str – LaTeX compilable equation string.

Examples

```
>>> from inquanto.operators import FermionOperator
>>> from sympy import sympify
>>> c = sympify("c")
>>> fo = FermionOperator([(-c, "F1 F2^"), (c**2, "F1^"), (c, "F0 F5 F3^")])
>>> print(fo.to_latex())
- c a_{1} a_{2}^{\dagger} + c^{2} a_{1}^{\dagger} + c a_{0} a_{5} a_{3}^{\
↪dagger}
>>> fo = FermionOperator([(1.0, "F0^ F1^ F3^"), (-1.0j, "F3 F1 F0")])
>>> print(fo.to_latex())
a_{0}^{\dagger} a_{1}^{\dagger} a_{3}^{\dagger} -\text{i} a_{3} a_{1} a_{0}
>>> from sympy import sqrt
>>> fo = FermionOperator([(sqrt(2), "F0^ F1"), (0.5-8j, "F5 F5^"), (2j, "F1^␣
↪F1^")])
>>> print(fo.to_latex(imaginary_unit=r"\text{j}", operator_symbol="f"))
\sqrt{2} f_{0}^{\dagger} f_{1} + (0.5-8.0\text{j}) f_{5} f_{5}^{\dagger} + 2.
↪0\text{j} f_{1}^{\dagger} f_{1}^{\dagger}
```

```
>>> from inquanto.operators import QubitOperator, QubitOperatorString
>>> qos1 = QubitOperatorString.from_string("X1 Y2 Z3")
>>> qos2 = QubitOperatorString.from_string("Z0 Y1 X4")
>>> qos3 = QubitOperatorString.from_list([(("a", [0]), "I"), (("b", [1]), "Z
↪"), (("c", [2]), "Z")])
>>> qo = QubitOperator({qos1: -1.0, qos2: 3+4j, qos3: c})
>>> print(qo.to_latex())
- X_{1} Y_{2} Z_{3} + (3.0+ 4.0\text{i}) Z_{0} Y_{1} X_{4} + c I_{0} Z_{1} Z_
↪{2}
>>> qo = QubitOperator({qos3: 3j})
>>> print(qo.to_latex(imaginary_unit="j", show_labels=True))
3.0j I^{\text{a}}_{0} Z^{\text{b}}_{1} Z^{\text{c}}_{2}
```

to_list()

Generate a list serialized representation of QubitPauliOperator,
suitable for writing to JSON.

Returns
JSON serializable list of dictionaries.

Return type
List[Dict[str, Any]]

to_sparse_matrix(*qubits=None*)

Represents the sparse operator as a dense operator under the ordering scheme specified by `qubits`, and generates the corresponding matrix.

- When `qubits` is an explicit list, the qubits are ordered with `qubits[0]` as the most significant qubit for indexing into the matrix.
- If `None`, then no padding qubits are introduced and we use the ILO-BE convention, e.g. `Qubit("a", 0)` is more significant than `Qubit("a", 1)` or `Qubit("b")`.
- Giving a number specifies the number of qubits to use in the final operator, treated as sequentially indexed from 0 in the default register (padding with identities as necessary) and ordered by ILO-BE so `Qubit(0)` is the most significant.

Parameters
qubits (*Union[List[Qubit], int, None], optional*) – Sequencing of qubits in the matrix, either as an explicit list, number of qubits to pad to, or infer from the operator. Defaults to None

Returns
A sparse matrix representation of the operator.

Return type
csc_matrix

toeplitz_decomposition()

Returns a tuple of the Hermitian and anti-Hermitian parts of the original *QubitOperator*.

In case the original *QubitOperator* object contains symbolic coefficients that do not have an associated type, those will be cast into both the real and imaginary Expr types and assigned to both objects.

Returns
Tuple[*QubitOperator*] – Hermitian and anti-Hermitian parts of operator.

Examples

```
>>> qo = QubitOperator.from_string("(1.0, X0 Y1), (0.1j, Y0 X1), (0.5 + 0.2j,␣
↪Z0 Z1)")
>>> herm_qo, antiherm_qo = qo.toeplitz_decomposition()
>>> print(herm_qo)
(1.0, X0 Y1), (0.5, Z0 Z1)
>>> print(antiherm_qo)
(0.1j, Y0 X1), (0.2j, Z0 Z1)
>>> a = Symbol('a', real=True)
>>> b = Symbol('b', imaginary=True)
>>> c = Symbol('c')
>>> p_str_a = QubitOperatorString.from_string("X0 Y1")
>>> p_str_b = QubitOperatorString.from_string("Y0 X1")
>>> p_str_c = QubitOperatorString.from_string("Z0 Z1")
>>> qo = QubitOperator({p_str_a: a, p_str_b: b, p_str_c: c})
>>> herm_qo, antiherm_qo = qo.toeplitz_decomposition()
>>> print(herm_qo)
```

(continues on next page)

(continued from previous page)

```
(a, X0 Y1), (re(c), Z0 Z1)
>>> print(antiherm_qo)
(b, Y0 X1), (1.0*I*im(c), Z0 Z1)
```

---

totally_commuting_decomposition(*abs_tol=1e-10*)

Decomposes into two separate operators, one including the totally commuting terms and the other including all other terms.

This will return two QubitOperators. The first is comprised of all terms which commute with all other terms, the second comprised of terms which do not. An empty QubitOperator will be returned if either of these sets is empty.

Parameters

abs_tol (`float`, default: `1e-10`) – Tolerance threshold used for determining commutativity - see .commutator() for details.

Returns

The totally commuting operator, and the remainder operator.

trotterize(*trotter_number=1*, *trotter_order=1*, *constant=1.0*, *coefficients_location=TrotterizeCoefficientsLocation.OUTER*)

Trotterizes the operator, treating the operator as an exponent.

Assuming that this operator is an exponent, this will generate an OperatorList with each element in the OperatorList corresponding to a single exponent in the Trotter product of exponentials.

Parameters

- **trotter_number** (`int`, default: `1`) – The number of time-slices in the Trotter approximation.
- **trotter_order** (`int`, default: `1`) – The order of the Trotter-Suzuki approximation being used. Currently, this only supports 1, i.e. ABABAB.
- **constant** (`Union[float, complex]`, default: `1.0`) – An additional constant factor to multiply each exponent by, which may be useful when, for example, constructing a time evolution operator.
- **coefficients_location** (*TrotterizeCoefficientsLocation*, default: `TrotterizeCoefficientsLocation.OUTER`) – By default, the coefficient of each term in the input operator is multiplied by the Trotter factor and stored in the outer coefficient of the generated OperatorList, with the coefficient of each inner operator set to 1. This behaviour can be controlled with this argument - set to "outer" for default behaviour. Setting this parameter to "inner" will store all scalars in the inner operator coefficient, with the outer coefficients of the generated OperatorList set to 1. Setting this parameter to "mixed" will store the Trotter factor in the outer coefficients of the generated OperatorList, with the inner coefficients of each term left untouched. See examples for a comparison.

Returns

`TypeVar`(`OperatorListT`, bound= OperatorList) – A Trotterized form of the exponential product, where each element is an individual exponent.

Examples

```
>>> from inquanto.operators import QubitOperator
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6) + QubitOperator("Z1 Z2 Z3 Z5", -5.6j)
```

(continues on next page)

(continued from previous page)

```
>>> trotter_operator = op1.trotterize(trotter_number=2)
>>> print(trotter_operator)
2.3       [(1.0, X0 Y1 Z3)],
-2.8j     [(1.0, Z1 Z2 Z3 Z5)],
2.3       [(1.0, X0 Y1 Z3)],
-2.8j     [(1.0, Z1 Z2 Z3 Z5)]
>>> from inquanto.operators import QubitOperator
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6) + QubitOperator("Z1 Z2 Z3 Z5", -5.6j)
>>> trotter_operator = op1.trotterize(trotter_number=2,coefficients_location=
↪"inner")
>>> print(trotter_operator)
1.0       [(2.3, X0 Y1 Z3)],
1.0       [(-2.8j, Z1 Z2 Z3 Z5)],
1.0       [(2.3, X0 Y1 Z3)],
1.0       [(-2.8j, Z1 Z2 Z3 Z5)]
>>> from inquanto.operators import QubitOperator
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6) + QubitOperator("Z1 Z2 Z3 Z5", -5.6j)
>>> trotter_operator = op1.trotterize(trotter_number=2,coefficients_location=
↪"mixed")
>>> print(trotter_operator)
0.5       [(4.6, X0 Y1 Z3)],
0.5       [(-5.6j, Z1 Z2 Z3 Z5)],
0.5       [(4.6, X0 Y1 Z3)],
0.5       [(-5.6j, Z1 Z2 Z3 Z5)]
```

unsympify()

Unsympifies dicionary values.

Replaces symbolic expressions with their corresponding numeric values.

Returns

LinearDictCombiner – Updated instance of LinearDictCombiner.

Raises

TypeError – When unsympification fails.

classmethod zero()

Return object with a zero dict entry.

Examples

```
>>> print(LinearDictCombiner.zero())
(0)
```

class QubitOperatorList(*data=None*, *coeff=1.0*)

Bases: OperatorList

Ordered list of *QubitOperator* objects associated with parameters.

Stores tuples of *QubitOperator* objects and corresponding coefficient values in an ordered list. In contrast to *QubitOperator*, this class is not assumed to comprise a linear combination. Typically, this will be of use when considering sequences of operators upon which some non-linear operation is performed. For instance, this may be used to store a Trotterised combination of exponentiated *QubitOperator* s.

This class may be instantiated from a *QubitOperator* or a *QubitOperatorString* object and a scalar

or symbolic coefficient. It may also be instantiated with a list of tuples of scalar or symbolic coefficients and *QubitOperator* objects.

Parameters

- **data** (Union[*QubitOperator*, *QubitOperatorString*, List[Tuple[Union[int, float, complex, Expr], *QubitOperator*]], None], default: None) – Input data from which the list of qubit operators is built. See *QubitOperator* for methods of constructing terms.
- **coeff** (Union[int, float, complex, Expr], default: 1.0) – Multiplicative scalar coefficient. Used only if data is not of type list.

Examples

```
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6)
>>> op2 = QubitOperator([(0, "Z")], -1.6j)
>>> op3 = QubitOperator("Z1 Z2 Z3 Z5", -5.6j)
>>> trotter_operator = QubitOperatorList([(1, op1), (1, op2), (1, op3)])
>>> print(trotter_operator)
1          [(4.6, X0 Y1 Z3)],
1          [(-1.6j, Z0)],
1          [(-5.6j, Z1 Z2 Z3 Z5)]
```

```
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6)
>>> op2 = QubitOperator([(0, "Z")], -1.6j)
>>> trotter_operator = QubitOperatorList([(sympify("a"), op1), (sympify("b"),␣
↪op2)])
>>> print(trotter_operator)
a          [(4.6, X0 Y1 Z3)],
b          [(-1.6j, Z0)]
```

class CompressScalarsBehavior(*value*)

Bases: str, Enum

Governs compression of scalars method behaviour.

ALL = 'all'

Combine all coefficients possible, simplifying inner terms.

ONLY_IDENTITIES_AND_ZERO = 'simple'

Only compress based on terms which are a scalar multiple of the identity operator, or zero.

OUTER = 'outer'

Combine all "outer" coefficients (coefficients stored directly in the top-level OperatorList) into one.

class ExpandExponentialProductCoefficientsBehavior(*value*)

Bases: str, Enum

Governs behaviour for expansion of exponential products of commuting operators.

BRING_INTO_OPERATOR = 'compact'

Treat top-level coefficients of the QubitOperatorList as being outside the exponential; multiply generated component QubitOperators by them and return QubitOperatorList with unit top-level coefficients.

IGNORE = 'ignore'

Drop top-level coefficients entirely.

IN_EXPONENT = 'inside'

Treat top-level coefficients of the QubitOperatorList as being within the exponent.

OUTSIDE_EXPONENT = 'outside'

Treat top-level coefficients of the QubitOperatorList as being outside the exponential; return them as top-level coefficients of the generated QubitOperatorList.

class FactoryCoefficientsLocation(*value*)

Bases: `str`, `Enum`

Determines where the from_Operator method places coefficients.

INNER = 'inner'

Coefficients are left within the component operators.

OUTER = 'outer'

Coefficients are moved to be directly stored at the top-level of the OperatorList.

property all_qubits: Set[Qubit]

Returns a set of all qubits included in any operator in the QubitOperatorList.

Return type
`Set`[`Qubit`]

clone()

Performs shallow copy of the object.

Return type
`TypeVar`(`SYMBOLICTYPE`, bound= Symbolic)

collapse_as_linear_combination(*ignore_outer_coefficients=False*)

Treating the OperatorList as a linear combination, return it in the form of a Operator.

By default, each term is multiplied by its corresponding scalar coefficient, then all such multiplied terms are summed to yield a single Operator. The first step may be skipped (i.e. the scalar coefficients associated with each component Operator may be ignored) by setting `ignore_outer_coefficients` to True.

Parameters
ignore_outer_coefficients (`bool`, default: `False`) – Set to True to skip multiplication by the "outer" coefficients in the OperatorList.

Returns
`TypeVar`(`OperatorT`, bound= Operator) – The sum of all terms in the OperatorList, multiplied by their associated coefficients if requested.

collapse_as_product(*reverse=False*, *ignore_outer_coefficients=False*)

Treating the OperatorList as a product of separate terms, return the full product as an Operator.

By default, each Operator in the OperatorList is multiplied by its corresponding coefficient, and the product is taken sequentially with the leftmost term given by the first element of the OperatorList. This behaviour can be reversed with the `reverse` parameter - if set to True, the leftmost term will be given by the last element of OperatorList.

If ignore_outer_coefficients is set to True, the first step (the multiplication of Operator terms by their corresponding coefficients) is skipped - i.e. the "outer" coefficients stored in the OperatorList are ignored.

> **Danger:** In the general case, the number of terms in the expansion will blow up exponentially (and thus the runtime of this method will also blow up exponentially).

Parameters

- **reverse** (bool, default: False) – Set to True to reverse the order of the product.
- **ignore_outer_coefficients** (bool, default: False) – Set to True to skip multiplication by the "outer" coefficients in the OperatorList.

Returns

TypeVar(OperatorT, bound= Operator) – The product of each component operator.

compatibility_matrix(*abs_tol=1e-10*)

Returns the compatibility matrix of the operator list.

This matrix is the adjacency matrix of the compatibility graph of the OperatorList. It is a boolean N*N matrix where N is the number of operators in the OperatorList. Element (n,m) is True if operators n and m commute, otherwise False.

Parameters

abs_tol (float, default: 1e−10) – Tolerance threshold used for determining commutativity - see QubitOperator.commutator() for details.

Returns

ndarray – The compatibility matrix of the operator list.

compress_scalars_as_product(*abs_tol=1e-12*, *inner_coefficient=False*, *coefficients_to_compress=CompressScalarsBehavior.ONLY_IDENTITIES_AND_ZERO*)

Treating the OperatorList as a product, compress identity terms or resolve to zero if possible.

To do this, we iterate through the (coefficient, operator) pairs in the OperatorList. If any coefficient or operator is zero, then we return an empty OperatorList, as the product will be zero. If the iteration operator is an identity, it will be treated as a scalar multiplier, itself multiplied by its associated coefficient, with the operator itself removed from the OperatorList. These multipliers are multiplied together and – if they do not equate to 1 – are prepended to the OperatorList as a separate identity term.

By default, (coefficient,operator) pairs which are not identity or zero will be ignored. This behaviour may be controlled with the coefficients_to_compress parameter. This can be set to "outer" to include all "outer" coefficients (of the (coefficient,operator) pairs) in the prepended identity term. It can also be set to "all" to additionally bring coefficients within the non-identity operators into the prepended identity term – i.e. for operators where all the "inner" coefficients are equal. See examples for a comparison.

The prepended identity term will, by default, store the multiplied scalar factor in the "outer" coefficient of the (coefficient, operator) pair. The inner_coefficient parameter may be set to True to instead store it within the operator.

Note: Term simplification is not performed on component operators. It is possible that a component operator will resolve on simplification to the identity or zero. This method will not catch these occurrences.

Parameters

- **abs_tol** (float, default: 1e−12) – Numerical threshold for comparison of numbers to 0 and 1. Set to None to use exact identity.
- **inner_coefficient** (bool, default: False) – Set to True to store generated scalar factors within the identity term, as described above.
- **coefficients_to_compress** (*CompressScalarsBehavior*, default: CompressScalarsBehavior.ONLY_IDENTITIES_AND_ZERO) – Controls which scalar factors will be combined, as described above.

Returns

TypeVar(OperatorListT, bound= OperatorList) – The OperatorList with identity and zero terms combined, as described above.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0", 2) + QubitOperator("Z1", 2)
>>> op2 = 3 * QubitOperator.identity()
>>> qol = QubitOperatorList([(5, op1), (7, op2)])
>>> result = qol.compress_scalars_as_product()
>>> print(result)
21.0      [(1.0, )],
5         [(2, X0), (2.0, Z1)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0", 2) + QubitOperator("Z1", 2)
>>> op2 = 3 * QubitOperator.identity()
>>> qol = QubitOperatorList([(5, op1), (7, op2)])
>>> result = qol.compress_scalars_as_product(inner_coefficient=True)
>>> print(result)
1         [(21.0, )],
5         [(2, X0), (2.0, Z1)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0", 2) + QubitOperator("Z1", 2)
>>> op2 = 3 * QubitOperator.identity()
>>> qol = QubitOperatorList([(5, op1), (7, op2)])
>>> result = qol.compress_scalars_as_product(coefficients_to_compress="outer")
>>> print(result)
105.0     [(1.0, )],
1.0       [(2, X0), (2.0, Z1)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0", 2) + QubitOperator("Z1", 2)
>>> op2 = 3 * QubitOperator.identity()
>>> qol = QubitOperatorList([(5, op1), (7, op2)])
>>> result = qol.compress_scalars_as_product(coefficients_to_compress="all")
>>> print(result)
210.0     [(1.0, )],
1.0       [(1.0, X0), (1.0, Z1)]
```

compute_jacobian (*symbols*, *as_sympy_sparse=False*)

Generate symbolic sparse Jacobian Matrix.

If the number of terms in the operator list is N and the number of symbols in the symbols list is M then the resulted matrix has a shape (N, M) and $J_{i,j} = \frac{\partial c_i}{\partial \theta_j}$

Parameters

- **symbols** (List[Symbol]) – List of symbols w.r.t. which we differentiate each coefficients.
- **as_sympy_sparse** (bool, default: False) – If this is true then it converts the output to an ImmutableSparseMatrix.

Returns

Union[List, ImmutableSparseMatrix] – Jacobian Matrix in list of tuples of (i,j, J_ij) (this format is referred as a row-sorted list of non-zero elements of the matrix.)

Examples

```
>>> op0 = QubitOperator("X0 Y2 Z3", 4.6)
>>> op1 = QubitOperator(((0, "X"), (1, "Y"), (3, "Z")), 0.1 + 2.0j)
>>> op2 = QubitOperator([(0, "X"), (1, "Z"), (3, "Z")], -1.3)
>>> qs = QubitOperatorString.from_string("X0 Y1 Y3")
>>> op3 = QubitOperator(qs, 0.8)
>>> qs0 = QubitOperatorString.from_string("X0 Y1 Z3 X4")
>>> qs1 = QubitOperatorString.from_tuple([(0, Pauli.Y), (1, Pauli.X)])
>>> dictionary = {qs0: 2.1, qs1: -1.7j}
>>> op4 = QubitOperator(dictionary)
```

```
>>> a, b, c = sympify("a,b,c")
>>> trotter_operator = QubitOperatorList([(a**3, op1), (2, op2), (b, op3), (c,
↪ op4)])
>>> print(trotter_operator)
a**3      [(0.1+2j, X0 Y1 Z3)],
2         [(-1.3, X0 Z1 Z3)],
b         [(0.8, X0 Y1 Y3)],
c         [(2.1, X0 Y1 Z3 X4), (-1.7j, Y0 X1)]
```

```
>>> jacobian_matrix = trotter_operator.compute_jacobian([a, b])
>>> print(jacobian_matrix)
[(0, 0, 3*a**2), (2, 1, 1)]
```

```
>>> jacobian_matrix = trotter_operator.compute_jacobian([a, b], as_sympy_
↪sparse=True)
>>> print(jacobian_matrix)
Matrix([[3*a**2, 0], [0, 0], [0, 1], [0, 0]])
```

copy()

Returns a deep copy of self.

Return type

LinearListCombiner

df()

Returns a Pandas DataFrame object of the dictionary.

dot_state_as_linear_combination(*state*, *qubits=None*)

Operate upon a state acting as a linear combination of the component operators.

Associated scalar constants are treated as scalar multiplier of their respective QubitOperator terms. This method can accept right-hand state as a `QubitState`, `QubitStateString` or a `numpy.ndarray`. In the former two cases, we maintain a symbolic representation of both operator and state, and each Pauli is implemented in sequence. Resultant states are returned as `QubitState`. In this case, the optional qubits parameter is ignored.

For a `numpy.ndarray`, we hand off to pytket's `QubitPauliOperator.dot_state()` method - this should be faster for dense states, but slower for sparse ones. Here, a register of qubits may be specified in the `qubits` parameter to determine the meaning of the indices of the provided state vector. From the pytket documentation:

- When `qubits` is an explicit list, the qubits are ordered with `qubits[0]` as the most significant qubit for indexing into state.

- If `None`, qubits sequentially indexed from 0 in the default register and ordered by ILO-BE so `Qubit(0)` is the most significant.

Parameters

- **state** (`Union[QubitState, QubitStateString, ndarray]`) – Input qubit state to operated on.
- **qubits** (`Optional[List[Qubit]]`, default: `None`) – For ndarray input, determines sequencing of qubits in the state, if not mapped to the default register. Ignored for other input types.

Returns

`Union[QubitState, ndarray]` – Output state.

dot_state_as_product (*state*, *reverse=False*, *qubits=None*)

Operate upon a state with each component operator in sequence, starting from operator indexed 0.

This will apply each operator in sequence, starting from the top of the list - i.e. the top of the list is on the right hand side when acting on a ket. Ordering can be controlled with the `reverse` parameter. Associated scalar constants are treated as scalar multipliers.

The right-hand state ket can be accepted as a QubitState,QubitStateString or a ndarray. In the former two cases, we maintain a symbolic representation of both operator and state, and each Pauli is implemented in sequence. Resultant states are returned as QubitState. In this case, the `qubits` parameter is ignored.

For ndarrays, we hand off to pytket's QubitPauliOperatorString.dot_state() method - this should be faster for dense states, but slower for sparse ones. Here, a register of qubits may be specified in the `qubits` parameter to determine the meaning of the indices of the provided state vector. From the pytket documentation:

- When `qubits` is an explicit list, the qubits are ordered with `qubits[0]` as the most significant qubit for indexing into state.
- If `None`, qubits sequentially indexed from 0 in the default register and ordered by ILO-BE so `Qubit(0)` is the most significant.

> **Danger:** In the general case, this will blow up exponentially.

Parameters

- **state** (`Union[QubitState, QubitStateString, ndarray]`) – The state to be acted upon.
- **reverse** (`bool`, default: `False`) – Set to True to reverse order of operations applied (i.e. treat the 0th indexed operator as the leftmost operator).
- **qubits** (`Optional[List[Qubit]]`, default: `None`) –

Returns

`Union[QubitState, ndarray]` – The state yielded from acting on input state in the type as described above.

empty()

Checks if internal list is empty.

Return type

`bool`

equality_matrix(*abs_tol=1e-10*)

Returns the equality matrix of the operator list.

The equality matrix is a boolean N*N matrix where N is the number of operators in the OperatorList. Element (n,m) is True if operators n and m are equal, otherwise False.

Parameters

abs_tol (Optional[float], default: 1e-10) – Threshold of comparing numeric values. Set to None to test for exact equivalence.

Returns

ndarray – The equality matrix of the QubitOperatorList.

evalf(**args*, ***kwargs*)

Numerically evaluates symbolic expressions stored in the left and right values of list items and replaces them with the results.

Parameters

- **args** (Any) – Args to be passed to *sympy.evalf()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.evalf()*.

Returns

LinearListCombiner – Updated instance of LinearListCombiner.

expand_exponential_product_commuting_operators(*expansion_coefficients_behavior=ExpandExponentialProduct*
additional_exponent=1.0,
check_commuting=True,
combine_scalars=True)

Trigonometrically expand a QubitOperatorList which represents a product of exponentials of operators with mutually commuting terms.

Given a QubitOperatorList, this method interprets it as a product of exponentials of the component operators. This means that for a QubitOperatorList consisting of terms (c_i,:math:*hat{O}_i*), it is interpreted as $\prod_i e^{c_i \hat{O}_i}$. Component operators must consist of terms which all mutually commute.

Each component operator is first expanded as its own exponential product (as they consist of terms which all mutually commute, $e^{A+B} = e^A e^B$). Each of the individual exponentiated terms are trigonometrically expanded. These are then concatenated to return a QubitOperatorList with the desired form.

Notes

Typically this may be used upon an operator generated by Trotterization methods to yield a computable form of the exponential product. While this method scales polynomially, expansion of the resulting operator or calculating its action on a state will typically be exponentially costly.

By default, this method will combine constant scalar multiplicative factors (obtained by, for instance, pi rotations) into one identity term prepended to the generated QubitOperatorList. This behaviour can be disabled by setting combine_scalars to False. For more detail, and for control over the process of combining constant factors, see the .compress_scalars_as_product method.

Parameters

- **expansion_coefficients_behavior** (*ExpandExponentialProductCoefficientsBehavior*, default: ExpandExponentialProductCoefficientsBehavior.IN_EXPONENT) – By default, it will be assumed that the 'outer' coefficients associated with each component

operator are within the exponent, as per the above formula. This may be set to "outside" to instead assume that they are scalar multiples of the exponential itself, rather than the exponent (i.e. $\prod_i c_i e^{\hat{O}_i}$). In this case, the coefficients will be returned as the outer coefficients of the returned QubitOperatorList, as input. Set to "compact" to assume similar structure to "outside", but returning coefficients within the component QubitOperators. Set to "ignore" to drop the 'outer' coefficients entirely. See examples for a comparison.

- **additional_exponent** (complex, default: 1.0) – Optional additional factor in each exponent.
- **check_commuting** – Set to False to skip checking whether all terms in each component operator commute.
- **combine_scalars** – Set to False to disable combination of identity and zero terms.

Returns

QubitOperatorList – The trigonometrically expanded form of the input QubitOperatorList.

Examples

```
>>> import sympy
>>> op= QubitOperator("X0", -1.j * sympy.Symbol('x'))
>>> qol = QubitOperatorList([(2, op)])
>>> result = qol.expand_exponential_product_commuting_operators()
>>> print(result)
1         [(1.0*cos(2.0*x), ), (-1.0*I*sin(2.0*x), X0)]
>>> import sympy
>>> op = QubitOperator("X0", -1.j * sympy.Symbol('x'))
>>> qol = QubitOperatorList([(2, op)])
>>> result = qol.expand_exponential_product_commuting_operators(expansion_
↪coefficients_behavior="outside")
>>> print(result)
2         [(1.0*cos(1.0*x), ), (-1.0*I*sin(1.0*x), X0)]
>>> import sympy
>>> op = QubitOperator("X0", -1.j * sympy.Symbol('x'))
>>> qol = QubitOperatorList([(2, op)])
>>> result = qol.expand_exponential_product_commuting_operators(expansion_
↪coefficients_behavior="ignore")
>>> print(result)
1         [(1.0*cos(1.0*x), ), (-1.0*I*sin(1.0*x), X0)]
>>> import sympy
>>> op = QubitOperator("X0", -1.j * sympy.Symbol('x'))
>>> qol = QubitOperatorList([(2, op)])
>>> result = qol.expand_exponential_product_commuting_operators(expansion_
↪coefficients_behavior="compact")
>>> print(result)
1.0       [(2.0*cos(1.0*x), ), (-2.0*I*sin(1.0*x), X0)]
```

free_symbols()

Returns the free symbols in the coefficient values.

Return type

set

free_symbols_ordered()

Returns the free symbols in the list, ordered alphabetically.

Returns
SymbolSet – Ordered set of symbols.

classmethod from_Operator(*input*, *additional_coefficient=1.0*, *coefficients_location=FactoryCoefficientsLocation.INNER*)

Converts an Operator to an OperatorList with terms in arbitrary order.

Each term in the Operator is split into a separate component Operator in the OperatorList. The resulting location of each scalar coefficient in the input Operator can be controlled with the coefficients_location parameter. Setting this to 'inner' will leave coefficients stored as part of the component operators, 'outer' will move the coefficients to the 'outer' level.

Parameters

- **input** (Operator) – The input operator to split into an OperatorList.
- **additional_coefficient** (Union[int, float, complex, Expr], default: 1.0) – An additional factor to include in the "outer" coefficients of the generated OperatorList.
- **coefficients_location** (*FactoryCoefficientsLocation*, default: FactoryCoefficientsLocation.INNER) – The destination of the coefficients of the input operator, as described above.

Returns
TypeVar(OperatorListT, bound= OperatorList) – An OperatorList as described above.

Raises
ValueError – On invalid input to the coefficients_location parameter.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op = QubitOperator("X0", 2.) + QubitOperator("Z1", 2.)
>>> qol = QubitOperatorList.from_Operator(op)
>>> print(qol)
1.0       [(2.0, X0)],
1.0       [(2.0, Z1)]
>>> op = QubitOperator("X0", 2.) + QubitOperator("Z1", 2.)
>>> qol = QubitOperatorList.from_Operator(op,coefficients_location='outer')
>>> print(qol)
2.0       [(1.0, X0)],
2.0       [(1.0, Z1)]
```

classmethod from_list(*ops*, *symbol_format='term{}'*)

Converts a list of QubitOperators to a QubitOperatorList.

Each QubitOperator in the list will be a separate entry in the generated QubitOperatorList. Fresh symbols will be generated to represent the "outer" coefficients of the generated QubitOperatorList.

Parameters

- **ops** (List[*QubitOperator*]) – A list of QubitOperators which will comprise the terms of the generated QubitOperatorList.
- **symbol_format** – A raw string containing one positional substitution (in which a numerical index will be placed), used to generate each symbolic coefficient.

Returns
QubitOperatorList – A QubitOperatorList with terms corresponding to the input QubitOperators, and coefficients as newly generated symbols.

classmethod from_string(*input_string*)

Constructs a child class instance from a string.

Parameters

input_string (str) – String in the format `coeff1 [(coeff1_1, term1_1), ..., (coeff1_n, term1_n)], ..., coeffn [(coeffn_1, termn_1), ...]`.

Returns

Child class object.

incompatibility_matrix(*abs_tol=1e-10*)

Returns the incompatibility matrix of the operator list.

This matrix is the adjacency matrix of the incompatibility graph of the OperatorList. It is a boolean N*N matrix where N is the number of operators in the OperatorList. Element (n,m) is False if operators n and m commute, otherwise True.

Parameters

abs_tol (float, default: `1e-10`) – Tolerance threshold used for determining commutativity - see QubitOperator.commutator() for details.

Returns

ndarray – The incompatibility matrix of the operator list.

items()

Returns internal list.

Return type

List[Tuple[Any, Union[int, float, complex, Expr]]]

make_hashable()

Return a hashable representation of the object.

Currently simply returns a stringified object.

Return type

str

map(*mapping*)

Updates list items right values in-place, using a mapping function provided.

Parameters

mapping (Callable[[Union[int, float, complex, Expr]], Union[int, float, complex, Expr]]) – A callable object which takes each original value and returns the corresponding new value.

Return type

LinearListCombiner

property n_symbols: int

Returns the number of free symbols in the object.

Return type

int

operator_class

alias of *QubitOperator*

parallelity_matrix(*abs_tol=1e-10*)

Returns the "parallelity matrix" of the operator list.

This matrix is a boolean N*N matrix where N is the number of operators in the OperatorList. Element (n,m) is True if operators n and m are parallel - i.e. they are a scalar multiple of each other. Otherwise, the element in the returned matrix is False.

Parameters
abs_tol (Optional[float], default: 1e-10) – Threshold of comparing numeric values. Set to None to test for exact equivalence.

Returns
ndarray – The "parallelity matrix" of the operator list.

print_table()

Print internal list formatted as a table.

Return type
None

qubitwise_compatibility_matrix()

Returns the qubit-wise compatibility matrix of the operator list.

This matrix is the adjacency matrix of the qubit-wise compatibility graph of the OperatorList. It is a boolean N*N matrix where N is the number of operators in the OperatorList. Element (n,m) is True if operators n and m qubit-wise commute, otherwise False. See QubitOperator.qubitwise_commutes_with() for further details on qubit-wise commutativity.

Note that individual QubitOperators within the OperatorList must be comprised of single terms for this to be well-defined.

Returns
ndarray – The qubit-wise compatibility matrix of the operator list.

qubitwise_incompatibility_matrix()

Returns the qubit-wise incompatibility matrix of the operator list.

This matrix is the adjacency matrix of the qubit-wise incompatibility graph of the OperatorList. It is a boolean N*N matrix where N is the number of operators in the OperatorList. Element (n,m) is False if operators n and m qubit-wise commute, otherwise True. See QubitOperator.qubitwise_commutes_with() for further details on qubit-wise commutativity.

Note that individual QubitOperators within the OperatorList must be comprised of single terms for this to be well-defined.

Returns: The qubit-wise incompatibility matrix of the operator list.

Return type
ndarray

reduce_exponents_by_commutation(*abs_tol=1e-10*)

Given a QubitOperatorList representing a product of exponentials, combine terms by commutation.

Given a QubitOperatorList, this method interprets it as a product of exponentials of the component operators. This means that for a QubitOperatorList consisting of terms (c_i,:math:*hat{O}_i*), it is interpreted as $\prod_i e^{c_i \hat{O}_i}$. This method attempts to combine terms which are scalable multiples of one another. Each term is commuted backwards through the QubitOperatorList until it reaches an operator that it does not commute with. If it encounters an operator which is a scalable multiple of the term, then the terms are combined. Otherwise, the term is left unchanged.

Parameters

abs_tol (float, default: 1e-10) – Tolerance threshold used for determining commutativity - see QubitOperator.commutator() for details.

Returns

QubitOperatorList – A reduced form of the QubitOperatorList as described above.

retrotterize(*new_trotter_number*, *initial_trotter_number=1*, *new_trotter_order=1*, *initial_trotter_order=1*, *constant=1.0*, *inner_coefficients=False*)

Retrotterize an expression given a OperatorList representing a product of exponentials.

This method assumes that the OperatorList represents a product of exponentials, with each Operator in the list corresponding to an exponent of a single exponential in the product. Scalar factors within the OperatorList are treated as scalar multipliers within each exponent.

The OperatorList is first untrotterized using the provided initial_trotter_number and initial_trotter_order, then subsequently Trotterized using the provided new_trotter_number and new_trotter_order. The returned OperatorList corresponds to the generated product of exponentials, in a similar manner to the original OperatorList.

Parameters

- **new_trotter_number** (int) – The desired number of Trotter steps in the final Trotter-Suzuki expansion.
- **initial_trotter_number** (int, default: 1) – The number of Trotter steps in the original Trotter-Suzuki expansion.
- **new_trotter_order** (int, default: 1) – The order of the original Trotter-Suzuki expansion used. Currently, only a first order (ABABAB…) expansion is supported.
- **initial_trotter_order** (int, default: 1) – The desired order of the final Trotter-Suzuki expansion. Currently, only a first order (ABABAB…) expansion is supported.
- **constant** (Union[float, complex], default: 1.0) – An additional constant multiplier in the exponent.
- **inner_coefficients** (bool, default: False) – By default, generated scalar factors in each exponent are stored in the coefficients of the generated OperatorList, with the coefficient of each inner Operator unchanged. Set this to True to instead store all scalar factors as coefficients in each Operator, with the outer coefficients of the OperatorList left unchanged. See examples for a comparison.

Returns

TypeVar(OperatorListT, bound= OperatorList) – The exponential product retrotterized with the provided new Trotter number and order. Each element is an individual exponent.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> retrotterised = qol.retrotterize(new_trotter_number=4,initial_trotter_
↪number=2)
>>> print(retrotterised)
0.25      [(1.0, X0 X1)],
0.25      [(1.0, Z0)],
0.25      [(1.0, X0 X1)],
```

(continues on next page)

(continued from previous page)

```
0.25       [(1.0, Z0)],
0.25       [(1.0, X0 X1)],
0.25       [(1.0, Z0)],
0.25       [(1.0, X0 X1)],
0.25       [(1.0, Z0)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> retrotterised = qol.retrotterize(new_trotter_number=4,initial_trotter_
↪number=2,inner_coefficients=True)
>>> print(retrotterised)
0.5        [(0.5, X0 X1)],
0.5        [(0.5, Z0)],
0.5        [(0.5, X0 X1)],
0.5        [(0.5, Z0)],
0.5        [(0.5, X0 X1)],
0.5        [(0.5, Z0)],
0.5        [(0.5, X0 X1)],
0.5        [(0.5, Z0)]
```

reversed_order()

Reverses internal list order and returns it as a new object.

Return type
LinearListCombiner

simplify(**args*, ***kwargs*)

Simplifies expressions stored in left and right values of list items.

Parameters

- **args** (Any) – Args to be passed to *sympy.simplify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.simplify()*.

Returns
LinearListCombiner – Updated instance of LinearListCombiner.

split()

Generates pair objects from list items.

Return type
Iterator[LinearListCombiner]

split_totally_commuting_set(*abs_tol=1e-10*)

For a QubitOperatorList, separate it into a totally commuting part and a remainder.

This will return two QubitOperatorList - the first comprised of all the component QubitOperators which commute with all other terms, the second comprised of all other terms. An empty QubitOperatorList will be returned if either of these sets is empty.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold used for determining commutativity - see QubitOperator.commutator() for details.

Returns
Two QubitOperatorLists - the first representing the totally commuting set, the second representing the rest of the operator.

sublist (*sublist_indices*)

Returns a new instance containing a subset of the terms of the original object.

Parameters

sublist_indices (list[int]) – List of indices of the operator list elements to constitute a new object.

Returns

TypeVar(OperatorListT, bound= OperatorList) – A new instance of OperatorList, being a sublist of the original operator.

Raises

ValueError – If sublist_indices contains indices not contained in self, or self is empty.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6)
>>> op2 = QubitOperator("Z0", -1.6j)
>>> op3 = QubitOperator("Z1 Z2 Z3 Z5", -5.6j)
>>> long_operator = QubitOperatorList([(1, op1), (1, op2), (1, op3)])
>>> short_operator = long_operator.sublist([0, 2])
>>> print(short_operator)
1          [(4.6, X0 Y1 Z3)],
1          [(-5.6j, Z1 Z2 Z3 Z5)]
```

---

subs (*symbol_map*)

Returns a new objects with symbols substituted.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type

TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution (*symbol_map=None*)

Substitutes free symbols for numerical values according to a map.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) –

Return type

LinearListCombiner

sympify (**args*, ***kwargs*)

Sympifies left and right values of list items.

Replaces left and right values with their corresponding symbolic expressions.

Parameters

- **args** (Any) – Args to be passed to *sympy.sympify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.sympify()*.

Returns
`LinearListCombiner` – Updated instance of LinearListCombiner.

Raises
RuntimeError – when sympification fails.

to_sparse_matrices(*qubits=None*)

Returns a list of sparse matrices representing each element of the QubitOperatorList.

Outer coefficients are treated by multiplying their corresponding QubitOperators. Otherwise, this method acts largely as a wrapper for QubitOperator.to_sparse_matrix(), which derives from the base pytket QubitPauliOperator.to_sparse_matrix() method. The qubits parameter specifies the ordering scheme for qubits. Note that if no explicit qubits are provided, we use the set of all qubits included in any operator in the QubitOperatorList (i.e. `self.all_qubits`), ordered ILO-BE as per pytket. From the pytket docs:

- When `qubits` is an explicit list, the qubits are ordered with `qubits[0]` as the most significant qubit for indexing into the matrix.
- If `None`, then no padding qubits are introduced and we use the ILO-BE convention, e.g. `Qubit("a",0)` is more significant than `Qubit("a",1)` or `Qubit("b")`.
- Giving a number specifies the number of qubits to use in the final operator, treated as sequentially indexed from 0 in the default register (padding with identities as necessary) and ordered by ILO-BE so `Qubit(0)` is the most significant.

Parameters
qubits (`Union[List[Qubit], int, None]`, default: `None`) – Sequencing of qubits in the matrix, either as an explicit list, number of qubits to pad to, or infer from the operator list.

Returns
`csc_matrix` – A sparse matrix representation of the operator.

trotterize_as_linear_combination(*trotter_number*, *trotter_order=1*, *constant=1.0*, *inner_coefficients=False*)

Trotterize an exponent linear combination of Operators.

This method assumes that the OperatorList represents the exponential of a linear combination of Operators, with each Operator within the OperatorList corresponding to a term in this linear combination. Trotterization is performed at the level of these Operators. The Operators contained within the returned OperatorList correspond to exponents within the Trotter sequence.

Parameters

- **trotter_number** (`int`) – The number of Trotter steps in the Trotter-Suzuki expansion.
- **trotter_order** (`int`, default: `1`) – The order of the Trotter-Suzuki approximation to be used. Presently, only first-order (ABABAB…) is supported.
- **constant** (`Union[float, complex]`, default: `1.0`) – An additional constant multiplier in the exponent.
- **inner_coefficients** (`bool`, default: `False`) – By default, generated scalar factors in each exponent are stored in the coefficients of the generated OperatorList, with the coefficient of each inner Operator unchanged. Set this to True to instead store all scalar factors as coefficients in each Operator, with the outer coefficients of the OperatorList left unchanged. See examples for a comparison.

Returns
`TypeVar(OperatorListT, bound= OperatorList)` – A Trotterized form of the exponential product, where each element is an individual exponent.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1.,op1),(1.,op2)])
>>> result = qol.trotterize_as_linear_combination(2)
>>> print(result)
0.5       [(1.0, X0 X1)],
0.5       [(1.0, Z0)],
0.5       [(1.0, X0 X1)],
0.5       [(1.0, Z0)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1.,op1),(1.,op2)])
>>> result = qol.trotterize_as_linear_combination(2,inner_coefficients=True)
>>> print(result)
1.0       [(0.5, X0 X1)],
1.0       [(0.5, Z0)],
1.0       [(0.5, X0 X1)],
1.0       [(0.5, Z0)]
```

unsympify()

Unsympifies dicionary values.

Replaces symbolic expressions with their corresponding numeric values.

Returns

LinearListCombiner – Updated instance of LinearListCombiner.

Raises

TypeError – When unsympification fails.

untrotterize(*trotter_number*, *trotter_order=1*)

Reverse a Trotter-Suzuki expansion given an OperatorList representing a product of exponentials.

This method assumes that the OperatorList represents a product of exponentials, with each Operator in the list corresponding to an exponent of a single exponential in the product. Scalar factors within the OperatorList are treated as scalar multipliers within each exponent. An Operator corresponding to the exponent of a single, untrotterized exponential is returned.

Parameters

- **trotter_number** (int) – The number of Trotter steps within the Trotter expansion to be reversed.
- **trotter_order** (int, default: 1) – The order of the Trotter-Suzuki expansion used. Currently, only a first order (ABABAB…) expansion is supported.

Returns

TypeVar(OperatorT, bound= Operator) – The exponent of the untrotterised operator.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
```

(continues on next page)

(continued from previous page)

```
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> untrotterised = qol.untrotterize(2)
>>> print(untrotterised)
(1.0, X0 X1), (1.0, Z0)
```

untrotterize_partitioned(*trotter_number*, *trotter_order=1*, *inner_coefficients=False*)

Reverse a Trotter-Suzuki expansion given a OperatorList representing a product of exponentials, maintaining separation of exponents.

This method assumes that the OperatorList represents a product of exponentials, with each Operator in the list corresponding to an exponent of a single exponential in the product. Scalar factors within the OperatorList are treated as scalar multipliers within each exponent. A OperatorList is returned wherein each term represents a single term in the exponent of the single, untrotterized exponential.

Parameters

- **trotter_number** (int) – The number of Trotter steps within the Trotter expansion to be reversed.
- **trotter_order** (int, default: 1) – The order of the Trotter-Suzuki expansion used. Currently, only a first order (ABABAB…) expansion is supported.
- **inner_coefficients** (bool, default: False) – By default, generated scalar factors in each exponent are stored in the coefficients of the generated OperatorList, with the coefficient of each inner Operator unchanged. Set this to True to instead store all scalar factors as coefficients in each Operator, with the outer coefficients of the OperatorList left unchanged. See examples for a comparison.

Returns

TypeVar(OperatorListT, bound= OperatorList) – The terms in the exponent of the untrotterised operator as a OperatorList.

Raises

ValueError – If the provided Trotter number is not compatible with the OperatorList.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> untrotterised = qol.untrotterize_partitioned(2)
>>> print(untrotterised)
1.0       [(1.0, X0 X1)],
1.0       [(1.0, Z0)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> untrotterised = qol.untrotterize_partitioned(2,inner_coefficients=True)
>>> print(untrotterised)
0.5       [(2.0, X0 X1)],
0.5       [(2.0, Z0)]
```

class QubitOperatorString

Bases: QubitPauliString

Handles a Pauli string: a term in *QubitOperator*.

anticommutator(*other*)

Calculates the commutator of this with another QubitOperatorString.

A quick check is first performed to determine if the operators anticommute, returning a zero QubitOperator if so. If not, the anticommutator is determined by explicit multiplication.

Notes

The anticommutator is returned as a QubitOperator, not a QubitOperatorString. This is due to the fact that the anticommutator will include a scalar factor in {1,i,-1,-i}, rather than being an unweighted Pauli string.

Parameters

other (*QubitOperatorString*) – The other Pauli string.

Returns

QubitOperator – The anticommutator of the two Pauli strings.

anticommutes_with(*other*)

Returns True if this QubitOperatorString anticommutes with the other, otherwise False.

This counts whether an even or odd number of qubits have differing non-identity Paulis acting on them, returning False in the case of even and True in the case of odd.

Parameters

other (*QubitOperatorString*) – The other Pauli string.

Returns

bool – True if the two Pauli strings anticommute, else False.

commutator(*other*)

Calculates the commutator of this with another QubitOperatorString.

This operator is treated as being on the left (i.e. A in AB-BA). A quick check is first performed to determine if the operators commute, returning a zero QubitOperator if so. If not, the commutator is determined by explicit multiplication.

Notes

The commutator is returned as a QubitOperator, not a QubitOperatorString. This is due to the fact that the commutator will include a scalar factor in {1,i,-1,-i}, rather than being an unweighted Pauli string.

Parameters

other (*QubitOperatorString*) – The other Pauli string.

Returns

QubitOperator – The commutator of the two Pauli strings.

commutes_with(*self: pytket._tket.pauli.QubitPauliString*, *other: pytket._tket.pauli.QubitPauliString*) → bool

Returns

True if the two strings commute, else False

compress (*self: pytket._tket.pauli.QubitPauliString*) → None

Removes I terms to compress the sparse representation.

dot_state (*state*, *qubits=None*)

Calculate the result of operating on a given qubit state.

Can accept right-hand state as a `QubitState`, `QubitStateString` or a `numpy.ndarray`. In the former two cases, we maintain a symbolic representation of both operator and state, and each Pauli is implemented in sequence. Resultant states are returned as `QubitState`. This should support sympy parametrised states & operators, but the use of parametrised states & operators is untested.

For a `numpy.ndarray`, we hand off to pytket's `QubitPauliOperatorString.dot_state()` method - this should be faster for dense states, but slower for sparse ones. Here, a register of qubits may be specified in the `qubits` parameter to determine the meaning of the indices of the provided state vector. From the pytket documentation:

- When `qubits` is an explicit list, the qubits are ordered with `qubits[0]` as the most significant qubit for indexing into state.
- If `None`, qubits sequentially indexed from 0 in the default register and ordered by ILO-BE so `Qubit(0)` is the most significant.

Parameters

- **state** (Union[*QubitState*, *QubitStateString*, ndarray]) – Input qubit state to operated on.
- **qubits** (Optional[List[Qubit]], default: None) – For ndarray input, determines sequencing of qubits in the state, if not mapped to the default register. Ignored for other input types.

Returns

Union[*QubitState*, ndarray] – Output state.

classmethod from_list (*s*)

Construct *QubitOperatorString* from list of tuples.

Parameters

s (List[Tuple[Tuple[str, List[int]], str]]) – A list containing entries of the form `((qubit_label, [qubit_index]), pauli)`.

Returns

QubitOperatorString – Output operator string.

Raises

ValueError – if multiple Paulis are acting on a single qubit.

Examples

```
>>> qo = QubitOperatorString.from_list([(("q", [0]), "X"), (("q", [1]), "Y")])
>>> print(qo)
X0 Y1
```

classmethod from_string (*s*)

Constructs *QubitOperatorString* from a string.

String should contain Pauli gate symbols (X, Y or Z) followed by index of a qubit on which it acts, with different pairs separated by space.

Parameters
s (str) – A python string representing a Pauli word.

Returns
QubitOperatorString – Output operator string.

Raises
ValueError – If multiple Paulis are acting on a single qubit.

Examples

```
>>> qs = QubitOperatorString.from_string("X1 X3 X5")
>>> print(qs)
X1 X3 X5
```

classmethod from_symplectic_row(*symplectic_row*, *qubits=None*)

Generate a *QubitOperatorString* from symplectic row vector form.

This is a length 2N vector where N = the number of qubits. The first N entries indicate if an X is performed on each qubit, and the next N entries indicate if a Z is performed on the qubit. Qubits are assigned based on provided qubit register from left to right, if qubits is not provided then a default register indexed 0-num_qubits is assumed.

Parameters

- **symplectic_row** (ndarray) – The row vector corresponding to the Pauli string in symplectic form.
- **qubits** (Optional[List[Qubit]], default: None) – A register of qubits.

Returns
QubitOperatorString – The operator described by symplectic_row.

classmethod from_tuple(*t*)

Constructs *QubitOperatorString* from a tuple.

Accepts a list of tuples, with the left value being an index of a qubit and the right value being a Pauli object (Pauli.X, Pauli.Y or Pauli.Z) designating a gate acting on this qubit.

Parameters
t (List[Tuple[int, Pauli]]) – A list of tuples representing a Pauli word.

Raises
ValueError – if multiple Paulis are acting on a single qubit.

Returns
QubitOperatorString – Output operator string.

Examples

```
>>> qs = QubitOperatorString.from_tuple([(1, Pauli.X), (3, Pauli.X), (5,␣
↪Pauli.X)])
>>> print(qs)
X1 X3 X5
```

property map

the QubitPauliString's underlying dict mapping Qubit to Pauli

Type
return

padded(*register_qubits=None*, *zero_to_max=False*)

Return a copy of the *QubitOperatorString* with implicit identities replaced with explicit identities.

A qubit register should be provided in order to determine which qubits the operator acts on. Alternatively, `zero_to_max` may be set to True in order to assume that the qubit register is indexed 0-N, where N is the highest integer indexed qubit in the original *QubitOperatorString*. These modes of operation are incompatible and this method will except if `zero_to_max` is set to True and qubits_register is provided.

Parameters

- **register_qubits** (`Optional[Iterable[Qubit]]`, default: `None`) – A qubit register used to determine which padding identities will be added.
- **zero_to_max** (`bool`, default: `False`) – Set to True to assume a 0-N indexed qubit register as described above.

Returns
QubitOperatorString – A copy of the *QubitOperatorString* with implicit identities over the specified register replaced with explicit identities.

Raises

- **PaddingIncompatibleArgumentsError** – If `register_qubits` has been provided while `zero_to_max` is set to True.
- **PaddingInferenceError** – If `zero_to_max` is set to True and maximum qubit index cannot be inferred, for instance by non-integer labelled qubits.
- **PaddingInvalidRegisterError** – If `zero_to_max` is not set and no qubit register is provided, or if string acts on qubits not in register.

Examples

```
>>> qs = QubitOperatorString.from_string("X0")
>>> print(qs.padded([Qubit(0),Qubit(1)]))
X0 I1
```

```
>>> qs = QubitOperatorString.from_string("X1")
>>> print(qs.padded(zero_to_max=True))
I0 X1
```

---

property pauli_list: List[Pauli]

Return list of Pauli objects contained in a *QubitOperatorString*.

Return type
`List[Pauli]`

property qubit_id_list: List[Qubit]

Return list of Qubits contained in a *QubitOperatorString*.

Return type
`List[Qubit]`

property qubit_list: List[Qubit]

Return list of Qubits contained in a *QubitOperatorString*.

Return type
List[Qubit]

qubitwise_anticommutes_with(*other*)

Returns True if this Pauli string qubit-wise anticommutes with another Pauli string, otherwise False.

Two Pauli strings qubit-wise anticommute if every Pauli acting on a given qubit anticommutes with the Pauli acting on the same qubit in the other string. This necessitates that the Paulis are different, and both non-identity, for every qubit.

Warning: By necessity, this method assumes that both Pauli strings act on a register comprised of the union of the qubits acted on by the two Pauli strings. This includes explicit identity operations, but not implicit ones. Caution should be used where QubitOperatorStrings involve implicit identities. Consider padding with explicit identities with the .padded() method.

Parameters
other (*QubitOperatorString*) – The other QubitOperatorString.

Returns
bool – True if the two QubitOperatorStrings qubit-wise anticommute, else False.

qubitwise_commutes_with(*other*)

Returns True if this Pauli string qubit-wise commutes with another Pauli string, otherwise False.

Two Pauli strings qubit-wise commute if every Pauli acting on a given qubit commutes with the Pauli acting on the same qubit in the other string. This necessitates that either the Paulis are the same, or that at least one is an identity.

Parameters
other (*QubitOperatorString*) – The other Pauli string.

Returns
bool – True if the Pauli strings are qubit-wise commuting, otherwise False.

state_expectation(**args*, ***kwargs*)

Overloaded function.

1. state_expectation(self: pytket._tket.pauli.QubitPauliString, state: numpy.ndarray[numpy.complex128[m, 1]]) -> complex

Calculates the expectation value of the state with the pauli string. Maps the qubits of the statevector with sequentially-indexed qubits in the default register, with `Qubit(0)` being the most significant qubit.

Parameters
state – statevector for qubits `Qubit(0)` to `Qubit(n-1)`

Returns
expectation value with respect to state

2. state_expectation(self: pytket._tket.pauli.QubitPauliString, state: numpy.ndarray[numpy.complex128[m, 1]], qubits: List[pytket._tket.circuit.Qubit]) -> complex

Calculates the expectation value of the state with the pauli string. Maps the qubits of the statevector according to the ordered list *qubits*, with `qubits[0]` being the most significant qubit.

Parameters

- **state** – statevector

- **qubits** – order of qubits in *state* from most to least significant

Returns
expectation value with respect to state

to_dict()

Returns the underlying dictionary of *QubitOperatorString*.

Return type
Dict[Qubit, Pauli]

to_latex(*show_indices=True*, *show_labels=False*, *swap_scripts=False*, *text_labels=True*)

Generate a LaTeX representation of the operator string.

Parameters

- **show_indices** (bool, default: True) – Toggle whether the index on each Pauli operator should be printed.
- **show_labels** (bool, default: False) – Toggle whether the label on each Pauli operator should be printed.
- **swap_scripts** (bool, default: False) – By default, indices are printed as subscripts and labels as superscripts. Set to True to swap them.
- **text_labels** (bool, default: True) – If True, labels are wrapped in text{}.

Returns
str – LaTeX compilable equation string.

Examples

```
>>> qos = QubitOperatorString.from_string("X0 X1 X2")
>>> print(qos.to_latex())
X_{0} X_{1} X_{2}
>>> print(qos.to_latex(show_labels=True, text_labels=False))
X^{q}_{0} X^{q}_{1} X^{q}_{2}
>>> qos = QubitOperatorString.from_list([(("Alice", [0, 1]), "X"), (("Bob",␣
↪[1]), "Y")])
>>> print(qos.to_latex(show_labels=True))
X^{\text{Alice}}_{0, 1} Y^{\text{Bob}}_{1}
>>> print(qos.to_latex(show_labels=True, show_indices=False, swap_
↪scripts=True))
X_{\text{Alice}} Y_{\text{Bob}}
```

to_list(*self: pytket._tket.pauli.QubitPauliString*) → json

A JSON-serializable representation of the QubitPauliString. :return: a list of Qubit-to-Pauli entries, represented as dicts.

to_sparse_matrix(**args*, ***kwargs*)

Overloaded function.

1. to_sparse_matrix(self: pytket._tket.pauli.QubitPauliString) -> scipy.sparse.csc_matrix[numpy.complex128]

Represents the sparse string as a dense string (without padding for extra qubits) and generates the matrix for the tensor. Uses the ILO-BE convention, so Qubit("a", 0) is more significant that Qubit("a", 1) and Qubit("b") for indexing into the matrix.

Returns
a sparse matrix corresponding to the operator

2. to_sparse_matrix(self: pytket._tket.pauli.QubitPauliString, n_qubits: int) -> scipy.sparse.csc_matrix[numpy.complex128]

Represents the sparse string as a dense string over *n_qubits* qubits (sequentially indexed from 0 in the default register) and generates the matrix for the tensor. Uses the ILO-BE convention, so `Qubit(0)` is the most significant bit for indexing into the matrix.

Parameters
n_qubits – the number of qubits in the full operator

Returns
a sparse matrix corresponding to the operator

3. to_sparse_matrix(self: pytket._tket.pauli.QubitPauliString, qubits: List[pytket._tket.circuit.Qubit]) -> scipy.sparse.csc_matrix[numpy.complex128]

Represents the sparse string as a dense string and generates the matrix for the tensor. Orders qubits according to *qubits* (padding with identities if they are not in the sparse string), so `qubits[0]` is the most significant bit for indexing into the matrix.

Parameters
qubits – the ordered list of qubits in the full operator

Returns
a sparse matrix corresponding to the operator

class RestrictedOneBodyRDM(*rdm1*, *rdms=None*)

Bases: `OneBodyRDM`

One-body reduced density matrix in a spin restricted representation.

Parameters

- **rdm1** (`ndarray`) – Reduced one-body density matrix as a 2D array. $\text{1RDM}_{ij} = \langle c^\dagger_{i,\alpha} c_{j,\alpha} \rangle + \langle c^\dagger_{i,\beta} c_{j,\beta} \rangle$
- **rdms** (`Optional`[`ndarray`], default: `None`) – Spin density matrix. If unspecified, spin symmetry is assumed (rdm1a = rdm1b). $\text{1RDM}^s_{ij} = \langle c^\dagger_{i,\alpha} c_{j,\alpha} \rangle - \langle c^\dagger_{i,\beta} c_{j,\beta} \rangle$

copy()

Performs a deep copy of object.

Return type
`RDM`

get_block(*mask*)

Return a new RDM spanning a subset of the original's orbitals.

All orbitals not specified in *mask* are ignored.

Parameters
mask (`ndarray`) – List of indices of orbitals to retain.

Returns
RestrictedOneBodyRDM – New, smaller RDM with only the target orbitals.

get_occupations(*as_int=True*)

Returns the diagonal elements of the 1-RDM.

Parameters
as_int (`bool`, default: `True`) – Should the results be rounded to the nearest integer.

Returns
List[Union[float, int]] – Number of electrons in each orbital.

classmethod load_h5(*name*)

Loads RDM object from .h5 file.

Parameters
name (Union[str, Group]) – Name of .h5 file to be loaded.

Returns
RDM – Loaded RDM object.

mean_field_rdm2()

Calculate the mean-field two-body RDM object.

Returns
RestrictedTwoBodyRDM – Two body RDM object.

n_orb()

Returns number of spatial orbitals.

Return type
int

n_spin_orb()

Returns number of spin-orbitals.

Return type
int

rotate(*rotation*)

Rotate the density matrix to a new basis.

Parameters
rotation (ndarray) – Rotation matrix as a 2D array.

Returns
RestrictedOneBodyRDM – RDM after rotation.

save_h5(*name*)

Dumps RDM object to .h5 file.

Parameters
name (Union[str, Group]) – Destination filename of .h5 file.

Return type
None

set_block(*mask*, *rdm*)

Set the RDM entries for a specified set of orbitals.

Parameters

- **mask** (ndarray) – List of indices of orbitals to be edited.
- **rdm** (*RestrictedOneBodyRDM*) – RDM object to replace target orbitals.

Returns
RestrictedOneBodyRDM – Updated RDM object with target orbitals overwritten.

trace()

Return the trace of the 1-RDM.

Return type
float

class RestrictedTwoBodyRDM(*rdm2*)

Bases: RDM

Two-body reduced density matrix in a spin restricted representation.

Parameters
rdm2 (ndarray) – Reduced two-body density matrix as a 4D array. $2\text{RDM}_{ijkl} = \sum_{\sigma,\sigma' \in \{\alpha,\beta\}} \langle c^\dagger_{i,\sigma} c^\dagger_{k,\sigma'} c_{l,\sigma'} c_{j,\sigma} \rangle$

copy()

Performs a deep copy of object.

Return type
RDM

classmethod load_h5(*name*)

Loads RDM object from .h5 file.

Parameters
name (Union[str, Group]) – Name of .h5 file to be loaded.

Returns
RDM – Loaded RDM object.

n_orb()

Returns number of spatial orbitals.

Return type
int

n_spin_orb()

Returns number of spin-orbitals.

Return type
int

rotate(*rotation*)

Rotate the density matrix to a new basis.

Parameters
rotation (ndarray) – Rotation matrix as a 2D array.

Returns
RestrictedTwoBodyRDM – RDM after rotation.

save_h5(*name*)

Dumps RDM object to .h5 file.

Parameters
name (Union[str, Group]) – Destination filename of .h5 file.

Return type
None

class SymmetryOperatorFermionic(**args*, ***kwargs*)

Bases: *FermionOperator*, SymmetryOperator

A class for representing symmetry in a fermionic space.

This is largely an extension of *FermionOperator*, providing functionality relating to validating symmetries and finding symmetry sectors.

Parameters

- **data** – Input data from which the Fermion operator is built. Multiple input formats are supported.
- **coeff** – Multiplicative scalar coefficient. Used only when data is of type tuple or *FermionOperatorString*.

class TrotterizeCoefficientsLocation(*value*)

Bases: str, Enum

Determines where coefficients will be stored upon performing Trotterization.

INNER = 'inner'

All coefficients will be stored in the "inner" coefficients, within the component QubitOperators in the Trotterized result.

MIXED = 'mixed'

The Trotter step factor will be stored in the "outer" coefficients, whereas the original coefficients of the original Operator will remain in the component Operators.

OUTER = 'outer'

All coefficients will be stored in the "outer" coefficients, the coefficients stored directly in the generated OperatorList.

apply_bra(*fock_state*)

Performs an operation on a bra FermionState state.

Parameters

fock_state (*FermionState*) – FermionState object (bra state).

Returns

FermionState – New bra state.

apply_ket(*fock_state*)

Performs an operation on a ket FermionState state.

Parameters

fock_state (*FermionState*) – FermionState object (ket state).

Returns

FermionState – New ket state.

approx_equal_to(*other*, *abs_tol=1e-10*)

Checks for equality between two FermionOperators.

First, dictionary equivalence is tested for. If false, operators are compared in normal-ordered form, and difference is compared to *abs_tol*.

Parameters

- **other** (*FermionOperator*) – An operator to test for equality with.
- **abs_tol** (float, default: 1e-10) – Tolerance threshold for negligible terms in comparison.

Returns
bool – True if this operator is equal to other, otherwise False.

> **Danger:** This method may use the *normal_ordered()* method internally, which may be exponentially costly.

approx_equal_to_by_random_subs (*other*, *order=1*, *abs_tol=1e-10*)

Checks if object's dictionary values are numerically identical to the other object values.

Symbols contained in the difference of the two objects, if any, are substituted by random numeric values prior to norm check.

Parameters

- **other** (LinearDictCombiner) – Object to compare to.
- **order** (int, default: 1) – Parameter specifying the norm formula (see numpy.linalg.norm documentation).
- **abs_tol** (float, default: 1e-10) – Threshold vs which the norm of the difference is checked.

Return type
bool

as_scalar (*abs_tol=None*)

If the operator is a sum of identity terms or zero, return the sum of the coefficients, otherwise return None.

Note that this does not perform combination on terms and will return zero only if all coefficients are zero.

Parameters
abs_tol (Optional[float], default: None) – Tolerance for checking if coefficients are zero. Set to None to test using a standard python == comparison.

Returns
Union[float, complex, None] – The operator as a scalar if it can be represented as such, otherwise None.

classmethod ca (*a*, *b*)

Return object with a single one-body creation-annihilation pair with a unit coefficient in a dict.

Parameters

- **a** (int) – Index of fermionic mode to which a creation operator is applied.
- **b** (int) – Index of fermionic mode to which an annihilation operator is applied.

Returns
The creation-annihilation operator pair.

Examples

```
>>> print(FermionOperator.ca(2, 0))
(1.0, F2^ F0 )
```

classmethod caca (*a*, *b*, *c*, *d*)

Return object with a single two-body creation-annihilation pair with a unit coefficient in a dict.

Ordered as creation-annihilation-creation-annihilation.

Parameters

- **a** (int) – Index of fermionic mode to which the first creation operator is applied.
- **b** (int) – Index of fermionic mode to which the first annihilation operator is applied.
- **c** (int) – Index of fermionic mode to which the second creation operator is applied.
- **d** (int) – Index of fermionic mode to which the second annihilation operator is applied.

Returns
The composed two-body operator.

Examples

```
>>> print(FermionOperator.caca(2, 0, 3, 1))
(1.0, F2^ F0  F3^ F1 )
```

classmethod ccaa(*a*, *b*, *c*, *d*)

Return object with a single two-body creation-annihilation pair with a unit coefficient in a dict.

Ordered as creation-creation-annihilation-annihilation.

Parameters

- **a** (int) – Index of fermionic mode to which the first creation operator is applied.
- **b** (int) – Index of fermionic mode to which the second creation operator is applied.
- **c** (int) – Index of fermionic mode to which the first annihilation operator is applied.
- **d** (int) – Index of fermionic mode to which the second annihilation operator is applied.

Returns
The composed two-body operator.

Examples

```
>>> print(FermionOperator.ccaa(3, 2, 1, 0))
(1.0, F3^ F2^ F1  F0 )
```

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

property coefficients: List[int | float | complex | Expr]

Returns dictionary values.

Return type
List[Union[int, float, complex, Expr]]

commutator(*other_operator*)

Return the commutator of self with other_operator.

This calculation is performed by explicit calculation through multiplication.

Parameters
other_operator (*FermionOperator*) – The other fermionic operator.

Returns
FermionOperator – The commutator.

commutes_with(*other_operator*, *abs_tol=1e-10*)

Returns True if self commutes with other_operator (within tolerance), otherwise False.

Parameters

- **other_operator** (*FermionOperator*) – The other fermionic operator.
- **abs_tol** (float, default: 1e-10) – Tolerance threshold for negligible terms in the commutator.

Returns
bool – True if operators commute, within tolerance, otherwise False.

compress(*abs_tol=1e-10*, *symbol_sub_type=CompressSymbolSubType.NONE*)

Combines duplicate terms, removing those with negligible coefficient.

Parameters

- **abs_tol** (float, default: 1e-10) – Tolerance for comparing values to zero.
- **symbol_sub_type** (Union[CompressSymbolSubType, str], default: CompressSymbolSubType.NONE) – Defines the behaviour for dealing with symbolic expressions in coefficients. If "none", symbolic expressions are left intact. If "unity", substitutes all free symbols with 1, and removes all imaginary and real components below tolerance. If "random", substitutes all free symbols with a random number between 0 and 1, and removes imaginary and real components below tolerance.

Warning: When :code: *symbol_sub_type != "none"*, this method assumes significant expression structure is known a priori, and is best suited to operators which have simple product expressions, such as excitation operators for VQE ansätze and digital quantum simulation. Otherwise, it may remove terms relevant to computation. Each expression is of the form $f(a_1, a_2, \ldots, a_n)$ for some symbols a_i. $|f(a_1, a_2, \ldots, a_n)|$ is assumed to monotonically increase in both real and imaginary components for all $a_i \in [0, 1]$.

Returns
LinearDictCombiner – Updated instance of LinearDictCombiner.

classmethod constant(*value*)

Return object with a provided constant entry in a dict.

Returns
Operator representation of provided constant scalar.

Examples

```
>>> print(FermionOperator.constant(0.5))
(0.5, )
```

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

dagger()

Returns the Hermitian conjugate of an operator.

Return type
FermionOperator

df()

Returns a Pandas DataFrame object of the dictionary.

Return type
DataFrame

empty()

Checks if dictionary is empty.

Return type
bool

evalf(**args*, ***kwargs*)

Evaluates symbolic expressions stored in dict values and replaces them with the results.

Parameters

- **args** (Any) – Args to be passed to *sympy.evalf()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.evalf()*.

Returns
LinearDictCombiner – Updated instance of LinearDictCombiner.

free_symbols()

Returns the free symbols in the coefficient values.

free_symbols_ordered()

Returns the free symbols in the dict, ordered alphabetically.

Returns
SymbolSet – Ordered set of symbols.

freeze(*index_map*, *occupation*)

Adaptation of OpenFermion's freeze_orbitals method with mask and consistent index pruning.

Parameters

- **index_map** (List[int]) – A list of integers or None entries, whose size is equal to the number of spin-orbitals, where None indicates orbitals to be frozen and the remaining sequence of integers is expected to be continuous.
- **occupation** (List[int]) – A list of 1s and 0s of the same length as index_map, indicating occupied and unoccupied orbitals.

Returns
FermionOperator – New operator with frozen orbitals removed.

classmethod from_list(*data*)

Construct *FermionOperator* from a list of tuples.

Parameters
data (List[Tuple[Union[int, float, complex, Expr], *FermionOperatorString*]]) – List data representing a fermion operator term or sum of fermion operator terms. Each term in the list must be a tuple containing a scalar multiplicative coefficient, followed by a *FermionOperatorString* or tuple (see *FermionOperator.from_tuple()*).

Returns

FermionOperator – Fermion operator object corresponding to a valid input.

Examples

```
>>> fos0 = FermionOperatorString(((1, 0), (2, 1)))
>>> fos1 = FermionOperatorString(((1, 0), (3, 1)))
>>> op = FermionOperator.from_list([(0.9, fos0), (0.1, fos1)])
>>> print(op)
(0.9, F1  F2^), (0.1, F1  F3^)
>>> op = FermionOperator.from_list([(0.9, ((1, 0), (2, 1))), (0.1, fos1)])
>>> print(op)
(0.9, F1  F2^), (0.1, F1  F3^)
```

classmethod from_string(*input_string*)

Constructs a child class instance from a string.

Parameters

input_string (str) – String in the format `coeff1 [(coeff1_1, term1_1), ..., (coeff1_n, term1_n)], ..., coeffn [(coeffn_1, termn_1), ...]`.

Returns

Child class object.

classmethod from_tuple(*data*, *coeff=1.0*)

Construct *FermionOperator* from a tuple of terms.

Parameters

- **data** (Tuple) – Data representing a fermion operator term, which may be a product of single fermion creation or anihilation operators. Creation and annihilation operators acting on orbital index `q` are given by tuples `(q, 0)` and `(q, 1)` respectively. A product of single operators is given by a tuple of tuples; for example, the number operator: `((q, 1), (q, 0))`.
- **coeff** (Union[int, float, complex, Expr], default: `1.0`) – Multiplicative scalar coefficient.

Returns

FermionOperator – Fermion operator object corresponding to a valid input.

Examples

```
>>> op = FermionOperator.from_tuple(((1, 0), (2, 1)), 1.0)
>>> print(op)
(1.0, F1  F2^)
```

classmethod identity()

Return object with an identity entry in a dict.

Examples

```
>>> print(FermionOperator.identity())
(1.0, )
```

infer_num_spin_orbs()

Returns the number of modes that this operator acts upon, inferring the existence of modes with index from 0 to the maximum index.

Returns

int – The minimum number of spin orbitals in the Fock space to which this *FermionOperator* operates on.

Examples

```
>>> op = FermionOperator.from_string("(1.0, F1 F2^)")
>>> print(op.infer_num_spin_orbs())
3
```

is_all_coeff_complex()

Check if all coefficients are complex values.

Warning: Returns None if there is a free symbol in a coefficient.

Return type

bool

is_all_coeff_imag()

Check if all coefficients are complex values.

Warning: Returns None if there is a free symbol in a coefficient.

Return type

bool

is_all_coeff_real()

Check if all coefficients are real values.

Warning: Returns None if there is a free symbol in a coefficient.

Return type

bool

is_all_coeff_symbolic()

Check if all coefficients contain free symbols.

Return type

bool

is_antihermitian(*abs_tol=1e-10*)

Returns True if operator is anti-Hermitian (within a tolerance), else False.

This explicitly calculates the Hermitian conjugate, multiplies by -1 and tests for equality. Normal-ordering is performed before comparison.

Parameters

abs_tol (`float`, default: `1e-10`) – Tolerance threshold for negligible terms in comparison.

Returns

`bool` – True if operator is antihermitian, otherwise False.

> **Danger:** This method uses the *normal_ordered()* method internally, which may be exponentially costly.

is_any_coeff_complex()

Check if any coefficient is a complex value.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type

`bool`

is_any_coeff_imag()

Check if any coefficient is an imaginary value.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type

`bool`

is_any_coeff_real()

Check if any coefficient is a real value.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type

`bool`

is_any_coeff_symbolic()

Check if any coefficient contains a free symbol.

Return type

`bool`

is_commuting_operator()

Return True if every term in operator commutes with every other term, otherwise False.

Return type

`bool`

is_hermitian(*abs_tol=1e-10*)

Returns True if operator is Hermitian (within a tolerance), else False.

This explicitly calculates the Hermitian conjugate and tests for equality. Normal-ordering is performed before comparison.

Parameters

abs_tol (float, default: 1e-10) – Tolerance threshold for negligible terms in comparison.

Returns

bool – True if operator is Hermitian, otherwise False.

Danger: This method uses the *normal_ordered()* method internally, which may be exponentially costly.

is_normal_ordered()

Return whether or not term is in normal-ordered form.

This method is a (modified) version of the OpenFermion *is_normal_ordered()* method. In our convention, a normal-ordered operator operator has all creation operators to the left of annihilation operators, and each "block" of creation/annihilation operators are ordered with orbital indices in descending order (from left to right).

Returns

bool – True if operator is normal-ordered, false otherwise.

Examples

```
>>> op = FermionOperator.from_string("(3.5, F1 F2^)")
>>> print(op.is_normal_ordered())
False
>>> op = FermionOperator.from_string("(3.5, F1^ F2^)")
>>> print(op.is_normal_ordered())
False
```

is_parallel_with(*other*, *abs_tol=1e-10*)

Returns True if other is parallel with this (i.e. a scalar multiple of this), otherwise False.

Parameters

- **other** (LinearDictCombiner) – The other object to compare against
- **abs_tol** (Optional[float], default: 1e-10) – Tolerance threshold for comparison. Set to None to test for exact equivalence.

Returns

bool – True if other is parallel with this, otherwise False.

is_self_inverse(*abs_tol=1e-10*)

Returns True if operator is its own inverse (within a tolerance), False otherwise.

This explicitly calculates the square of the operator and compares to the identity. Normal-ordering is performed before comparison.

Parameters

abs_tol (float, default: 1e-10) – Tolerance threshold for negligible terms in comparison.

Returns

bool – True if operator is self-inverse, otherwise False.

> **Danger:** This method uses the *normal_ordered()* method internally, which may be exponentially costly.

is_symmetry_of(*operator*)

Check if operator is symmetry of given operator.

Checks by determining if operator commutes with all terms of other operator. True if it commutes, False otherwise.

Parameters

operator (*FermionOperator*) – Operator to compare to.

Returns

bool – True if this is a symmetry of operator, otherwise False.

is_two_body_number_conserving(*check_spin_symmetry=False*)

Query whether operator has correct form to be from a molecule.

This method is a copy of the OpenFermion *is_two_body_number_conserving()* method.

Require that term is particle-number conserving (same number of raising and lowering operators). Require that term has 0, 2 or 4 ladder operators. Require that term conserves spin (parity of raising operators equals parity of lowering operators).

Parameters

check_spin_symmetry (*bool*) – Whether to check if operator conserves spin.

Returns

bool – True if operator conserves electron number (and, optionally, spin), False otherwise.

Examples

```
>>> op = FermionOperator.from_string("(1.0, F1^ F0)")
>>> print(op.is_two_body_number_conserving())
True
>>> op = FermionOperator.from_string("(1.0, F0^)")
>>> print(op.is_two_body_number_conserving())
False
>>> op = FermionOperator.from_string("(1.0, F1^ F0^)")
>>> print(op.is_two_body_number_conserving())
False
>>> op = FermionOperator.from_string("(1.0, F1^ F0)")
>>> print(op.is_two_body_number_conserving(check_spin_symmetry=True))
False
```

---

is_unit_1norm(*abs_tol=1e-10*)

Returns True if operator has unit 1-norm, else False.

Parameters

abs_tol (float, default: 1e−10) – Tolerance threshold for comparison with unity.

Return type

bool

is_unit_2norm(*abs_tol=1e-10*)

Returns True if operator has unit 1-norm, else False.

Parameters

abs_tol (float, default: 1e−10) – Tolerance threshold for comparison with unity.

Return type
bool

is_unit_norm(*order=2*, *abs_tol=1e-10*)

Returns True if operator has unit p-norm, else False.

Parameters

- **order** (int, default: 2) – Norm order.
- **abs_tol** (float, default: 1e-10) – Tolerance threshold for comparison with unity.

Raises
ValueError – Coefficients contain free symbols.

Return type
bool

is_unitary(*abs_tol=1e-10*)

Returns True if operator is unitary (within a tolerance), False otherwise.

This explicitly calculates the Hermitian conjugate, right-multiplies by the initial operator and tests for equality to the identity. Normal-ordering is performed before comparison.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold for negligible terms in comparison.

Returns
bool – True if operator is unitary, otherwise False.

Danger: This method uses the *normal_ordered()* method internally, which may be exponentially costly.

items()

Returns dictionary items.

Return type
ItemsView[Any, Union[int, float, complex, Expr]]

static key_from_str(*key_str*)

Returns a FermionOperatorString instance initiated from the input string.

Parameters
key_str (str) – An input string describing the FermionOperatorString to be created - see FermionOperatorString.from_string() for more detail.

Returns
FermionOperatorString – A generated FermionOperatorString.

list_class

alias of *FermionOperatorList*

make_hashable()

Return a hashable representation of the object.

Currently simply returns a stringified object.

Return type
str

map(*mapping*)

Updates dictionary values, using a mapping function provided.

Parameters

mapping (Callable[[Union[int, float, complex, Expr]], Union[int, float, complex, Expr]]) – Mapping function to update the dict.

Returns

LinearDictCombiner – None.

property n_symbols: int

Returns the number of free symbols in the object.

Return type

int

norm_coefficients(*order=2*)

Returns the p-norm of the coefficients.

Parameters

order (int, default: 2) – Norm order.

Return type

Union[complex, float]

normal_ordered()

Returns a normal-ordered version of *FermionOperator*.

Normal-ordering the operator moves all creation operators to the left of annihilation operators, and orders orbital indices in descending order (from left to right).

Notes

Makes use of OpenFermion _normal_ordered_ladder_term() function.

Examples

```
>>> op = FermionOperator.from_string("(0.5, F1 F2^), (0.2, F1 F2 F3^ F4^)")
>>> op_no = op.normal_ordered()
>>> print(op_no)
(-0.5, F2^ F1 ), (0.2, F4^ F3^ F2  F1 )
```

Return type

FermionOperator

normalized(*norm_value=1.0*, *norm_order=2*)

Returns a copy of this object with normalised coefficients.

Parameters

- **norm_value** (float, default: 1.0) – The desired norm of the returned operator.
- **norm_order** (int, default: 2) – The order of the norm to be used.

Returns

LinearDictCombiner – A copy of the object with coefficients normalised to the desired value.

property num_spin_orbs: int

Return the number of spin-orbitals that this operator explicitly acts on.

Return type
int

permuted_operator(*permutation*)

Permutes the indices in a FermionOperator.

Permutation is according to a list or a dict of indices, mapping the old to a new operator order.

Parameters
permutation (Union[Dict, List[int]]) – List of integers or a dict, mapping the old operator terms indices to the new ones. In case if a list is given, list index acts as a key (old index) and list value corresponds to the new index of an operator. If a dict is given, it can only contain the indices to be permuted for higher efficiency.

Returns
FermionOperator – Permuted FermionOperator object.

Examples

```
>>> op = FermionOperator.ccaa(1,4,5,6) + FermionOperator.ca(0,4)
>>> print(op)
(1.0, F1^ F4^ F5  F6 ), (1.0, F0^ F4 )
>>> print(op.permuted_operator([0,4,2,3,1,5,6]))
(1.0, F4^ F1^ F5  F6 ), (1.0, F0^ F1 )
>>> print(op.permuted_operator({1:4, 4:1}))
(1.0, F4^ F1^ F5  F6 ), (1.0, F0^ F1 )
```

---

print_table()

Print dictionary formatted as a table.

Return type
NoReturn

qubit_encode(*mapping=None*, *qubits=None*)

Performs qubit encoding (mapping), using provided mapping function, of the current FermionOperator.

Parameters

- **mapping** (QubitMapping, default: None) – Mapping class. Default mapping procedure is Jordan-Wigner.
- **qubits** (Optional[List[Qubit]], default: None) – The qubit register. If left as None, a default register will be assumed if possible. See the mapping class documentation for further details.

Returns
Mapped QubitOperator object.

remove_global_phase(*phase=0.0*)

Returns a copy with a global phase applied such that the first element has the desired phase.

Parameters
phase (float, default: 0.0) – The phase to yield on the first element, in half-turns (i.e. multiples of pi).

Returns
LinearDictCombiner – A copy of the object with the desired global phase applied.

reversed_order()

Reverses the order of terms and returns it as a new object.

Return type
LinearDictCombiner

simplify(**args*, ***kwargs*)

Simplifies expressions stored in dictionary values.

Parameters

- **args** (Any) – Args to be passed to *sympy.simplify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.simplify()*.

Returns
LinearDictCombiner – Updated instance of LinearDictCombiner.

split()

Generates single-term FermionOperator objects.

subs(*symbol_map*)

Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution(*symbol_map=None*)

Substitutes free symbols for numerical values according to a map.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) –

Return type
LinearDictCombiner

symmetry_sector(*state*)

Find the symmetry sector that a fermionic state is in by direct expectation value calculation.

Parameters
state (*FermionState*) – Input fermionic state.

Returns
Union[complex, float, int] – The symmetry sector of the state (i.e. the expectation value).

Notes

Validity is not checked and providing symmetry-broken states may lead to odd results.

> **Danger:** Due to direct expectation value calculation, this may scale exponentially in the general case, although it should be okay for sparse states and operators e.g. Z2 symmetries on number states.

sympify(**args*, ***kwargs*)

Sympifies dictionary values.

Replaces values with their corresponding symbolic expressions.

Parameters

- **args** (Any) – Args to be passed to *sympy.sympify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.sympify()*.

Returns

LinearDictCombiner – Updated instance of LinearDictCombiner.

Raises

RuntimeError – When sympification fails.

property terms: List[Any]

Returns dictionary keys.

Return type

List[Any]

to_ChemistryRestrictedIntegralOperator()

Convert fermion operator into a restricted integral operator.

Uses the *ChemistryRestrictedIntegralOperator.from_FermionOperator()* method internally.

Return type

ChemistryRestrictedIntegralOperator

to_ChemistryUnrestrictedIntegralOperator()

Convert fermion operator into an unrestricted integral operator.

Uses the *ChemistryUnrestrictedIntegralOperator.from_FermionOperator()* method internally.

Return type

ChemistryUnrestrictedIntegralOperator

to_latex(*imaginary_unit='\\\\text{i}'*, ***kwargs*)

Generate a LaTeX representation of the operator.

Parameters

- **imaginary_unit** (str, default: r"\\text{i}") – Symbol to use for the imaginary unit.
- ****kwargs** – Keyword arguments passed to the *to_latex()* method of component operator strings (*FermionOperatorString* or *QubitOperatorString*).

Returns

str – LaTeX compilable equation string.

Examples

```
>>> from inquanto.operators import FermionOperator
>>> from sympy import sympify
>>> c = sympify("c")
>>> fo = FermionOperator([(-c, "F1 F2^"), (c**2, "F1^"), (c, "F0 F5 F3^")])
>>> print(fo.to_latex())
- c a_{1} a_{2}^{\dagger} + c^{2} a_{1}^{\dagger} + c a_{0} a_{5} a_{3}^{\
↪dagger}
>>> fo = FermionOperator([(1.0, "F0^ F1^ F3^"), (-1.0j, "F3 F1 F0")])
>>> print(fo.to_latex())
a_{0}^{\dagger} a_{1}^{\dagger} a_{3}^{\dagger} -\text{i} a_{3} a_{1} a_{0}
>>> from sympy import sqrt
>>> fo = FermionOperator([(sqrt(2), "F0^ F1"), (0.5-8j, "F5 F5^"), (2j, "F1^␣
↪F1^")])
>>> print(fo.to_latex(imaginary_unit=r"\text{j}", operator_symbol="f"))
\sqrt{2} f_{0}^{\dagger} f_{1} + (0.5-8.0\text{j}) f_{5} f_{5}^{\dagger} + 2.
↪0\text{j} f_{1}^{\dagger} f_{1}^{\dagger}
```

```
>>> from inquanto.operators import QubitOperator, QubitOperatorString
>>> qos1 = QubitOperatorString.from_string("X1 Y2 Z3")
>>> qos2 = QubitOperatorString.from_string("Z0 Y1 X4")
>>> qos3 = QubitOperatorString.from_list([(("a", [0]), "I"), (("b", [1]), "Z
↪"), (("c", [2]), "Z")])
>>> qo = QubitOperator({qos1: -1.0, qos2: 3+4j, qos3: c})
>>> print(qo.to_latex())
- X_{1} Y_{2} Z_{3} + (3.0+ 4.0\text{i}) Z_{0} Y_{1} X_{4} + c I_{0} Z_{1} Z_
↪{2}
>>> qo = QubitOperator({qos3: 3j})
>>> print(qo.to_latex(imaginary_unit="j", show_labels=True))
3.0j I^{\text{a}}_{0} Z^{\text{b}}_{1} Z^{\text{c}}_{2}
```

---

trotterize(*trotter_number=1*, *trotter_order=1*, *constant=1.0*, *coefficients_location=TrotterizeCoefficientsLocation.OUTER*)

Trotterizes the operator, treating the operator as an exponent.

Assuming that this operator is an exponent, this will generate an OperatorList with each element in the OperatorList corresponding to a single exponent in the Trotter product of exponentials.

Parameters

- **trotter_number** (int, default: 1) – The number of time-slices in the Trotter approximation.
- **trotter_order** (int, default: 1) – The order of the Trotter-Suzuki approximation being used. Currently, this only supports 1, i.e. ABABAB.
- **constant** (Union[float, complex], default: 1.0) – An additional constant factor to multiply each exponent by, which may be useful when, for example, constructing a time evolution operator.
- **coefficients_location** (*TrotterizeCoefficientsLocation*, default: TrotterizeCoefficientsLocation.OUTER) – By default, the coefficient of each term in the input operator is multiplied by the Trotter factor and stored in the outer coefficient of the generated OperatorList, with the coefficient of each inner operator set to 1. This behaviour can be controlled with this argument - set to "outer" for default behaviour. Setting this parameter to "inner" will store all scalars in the inner operator coefficient, with the outer coefficients of the generated OperatorList set to 1. Setting this parameter to

"mixed" will store the Trotter factor in the outer coefficients of the generated OperatorList, with the inner coefficients of each term left untouched. See examples for a comparison.

Returns

TypeVar(OperatorListT, bound= OperatorList) – A Trotterized form of the exponential product, where each element is an individual exponent.

Examples

```
>>> from inquanto.operators import QubitOperator
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6) + QubitOperator("Z1 Z2 Z3 Z5", -5.6j)
>>> trotter_operator = op1.trotterize(trotter_number=2)
>>> print(trotter_operator)
2.3       [(1.0, X0 Y1 Z3)],
-2.8j     [(1.0, Z1 Z2 Z3 Z5)],
2.3       [(1.0, X0 Y1 Z3)],
-2.8j     [(1.0, Z1 Z2 Z3 Z5)]
>>> from inquanto.operators import QubitOperator
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6) + QubitOperator("Z1 Z2 Z3 Z5", -5.6j)
>>> trotter_operator = op1.trotterize(trotter_number=2,coefficients_location=
↪"inner")
>>> print(trotter_operator)
1.0       [(2.3, X0 Y1 Z3)],
1.0       [(-2.8j, Z1 Z2 Z3 Z5)],
1.0       [(2.3, X0 Y1 Z3)],
1.0       [(-2.8j, Z1 Z2 Z3 Z5)]
>>> from inquanto.operators import QubitOperator
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6) + QubitOperator("Z1 Z2 Z3 Z5", -5.6j)
>>> trotter_operator = op1.trotterize(trotter_number=2,coefficients_location=
↪"mixed")
>>> print(trotter_operator)
0.5       [(4.6, X0 Y1 Z3)],
0.5       [(-5.6j, Z1 Z2 Z3 Z5)],
0.5       [(4.6, X0 Y1 Z3)],
0.5       [(-5.6j, Z1 Z2 Z3 Z5)]
```

truncated (*tolerance=1e-8*, *normal_ordered=True*)

Prunes *FermionOperator* terms with coefficients below provided threshold.

Parameters

- **tolerance** – Threshold below which terms are removed.
- **normal_ordered** – Should the operator be returned in normal-ordered form.

Returns

FermionOperator – New, modified operator.

Examples

```
>>> op = FermionOperator.from_string("(0.999, F0 F1^), (0.001, F2^ F1 )")
>>> print(op.truncated(tolerance=0.005, normal_ordered=False))
(0.999, F0  F1^)
>>> print(op.truncated(tolerance=0.005))
(-0.999, F1^ F0 )
```

unsympify()

Unsympifies dicionary values.

Replaces symbolic expressions with their corresponding numeric values.

Returns

LinearDictCombiner – Updated instance of LinearDictCombiner.

Raises

TypeError – When unsympification fails.

classmethod zero()

Return object with a zero dict entry.

Examples

```
>>> print(LinearDictCombiner.zero())
(0)
```

---

class SymmetryOperatorFermionicFactorised(*data=None*, *coeff=1.0*)

Bases: *FermionOperatorList*, SymmetryOperator

Stores a factorised form of fermionic symmetry operators.

Our symmetry operators may be an exponentiated sum of ladder operator strings. Expanding this out as a single *SymmetryOperatorFermionic* would blow up exponentially. However, we know that many properties can be calculated without this cost - particularly when sets of terms are known to map to single Pauli strings. Here, we store the symmetry operator in factorised form - the overall symmetry operator is the product of self.operators. To make calculation easier, we also enforce that the component operators must all mutually commute.

Parameters

- **data** (Union[*FermionOperator*, *FermionOperatorString*, List[Tuple[Union[int, float, complex, Expr], *FermionOperator*]], None], default: None) –
- **coeff** (Union[int, float, complex, Expr], default: 1.0) –

class CompressScalarsBehavior(*value*)

Bases: str, Enum

Governs compression of scalars method behaviour.

ALL = 'all'

Combine all coefficients possible, simplifying inner terms.

ONLY_IDENTITIES_AND_ZERO = 'simple'

Only compress based on terms which are a scalar multiple of the identity operator, or zero.

OUTER = 'outer'

Combine all “outer” coefficients (coefficients stored directly in the top-level OperatorList) into one.

class FactoryCoefficientsLocation(*value*)

Bases: str, Enum

Determines where the from_Operator method places coefficients.

INNER = 'inner'

Coefficients are left within the component operators.

OUTER = 'outer'

Coefficients are moved to be directly stored at the top-level of the OperatorList.

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

collapse_as_linear_combination(*ignore_outer_coefficients=False*)

Treating the OperatorList as a linear combination, return it in the form of a Operator.

By default, each term is multiplied by its corresponding scalar coefficient, then all such multiplied terms are summed to yield a single Operator. The first step may be skipped (i.e. the scalar coefficients associated with each component Operator may be ignored) by setting `ignore_outer_coefficients` to True.

Parameters
ignore_outer_coefficients (bool, default: False) – Set to True to skip multiplication by the "outer" coefficients in the OperatorList.

Returns
TypeVar(OperatorT, bound= Operator) – The sum of all terms in the OperatorList, multiplied by their associated coefficients if requested.

collapse_as_product(*reverse=False*, *ignore_outer_coefficients=False*)

Treating the OperatorList as a product of separate terms, return the full product as an Operator.

By default, each Operator in the OperatorList is multiplied by its corresponding coefficient, and the product is taken sequentially with the leftmost term given by the first element of the OperatorList. This behaviour can be reversed with the `reverse` parameter - if set to True, the leftmost term will be given by the last element of OperatorList.

If ignore_outer_coefficients is set to True, the first step (the multiplication of Operator terms by their corresponding coefficients) is skipped - i.e. the "outer" coefficients stored in the OperatorList are ignored.

> **Danger:** In the general case, the number of terms in the expansion will blow up exponentially (and thus the runtime of this method will also blow up exponentially).

Parameters

- **reverse** (bool, default: False) – Set to True to reverse the order of the product.
- **ignore_outer_coefficients** (bool, default: False) – Set to True to skip multiplication by the "outer" coefficients in the OperatorList.

Returns
TypeVar(OperatorT, bound= Operator) – The product of each component operator.

compress_scalars_as_product(*abs_tol=1e-12*, *inner_coefficient=False*, *coefficients_to_compress=CompressScalarsBehavior.ONLY_IDENTITIES_AND_ZERO*)

Treating the OperatorList as a product, compress identity terms or resolve to zero if possible.

To do this, we iterate through the (coefficient, operator) pairs in the OperatorList. If any coefficient or operator is zero, then we return an empty OperatorList, as the product will be zero. If the iteration operator is an identity, it will be treated as a scalar multiplier, itself multiplied by its associated coefficient, with the operator itself removed from the OperatorList. These multipliers are multiplied together and – if they do not equate to 1 – are prepended to the OperatorList as a separate identity term.

By default, (coefficient,operator) pairs which are not identity or zero will be ignored. This behaviour may be controlled with the coefficients_to_compress parameter. This can be set to "outer" to include all "outer" coefficients (of the (coefficient,operator) pairs) in the prepended identity term. It can also be set to "all" to additionally bring coefficients within the non-identity operators into the prepended identity term – i.e. for operators where all the "inner" coefficients are equal. See examples for a comparison.

The prepended identity term will, by default, store the multiplied scalar factor in the "outer" coefficient of the (coefficient, operator) pair. The inner_coefficient parameter may be set to True to instead store it within the operator.

Note: Term simplification is not performed on component operators. It is possible that a component operator will resolve on simplification to the identity or zero. This method will not catch these occurrences.

Parameters

- **abs_tol** (`float`, default: `1e-12`) – Numerical threshold for comparison of numbers to 0 and 1. Set to None to use exact identity.
- **inner_coefficient** (`bool`, default: `False`) – Set to True to store generated scalar factors within the identity term, as described above.
- **coefficients_to_compress** (*CompressScalarsBehavior*, default: `CompressScalarsBehavior.ONLY_IDENTITIES_AND_ZERO`) – Controls which scalar factors will be combined, as described above.

Returns

`TypeVar`(`OperatorListT`, bound= OperatorList) – The OperatorList with identity and zero terms combined, as described above.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0", 2) + QubitOperator("Z1", 2)
>>> op2 = 3 * QubitOperator.identity()
>>> qol = QubitOperatorList([(5, op1), (7, op2)])
>>> result = qol.compress_scalars_as_product()
>>> print(result)
21.0      [(1.0, )],
5         [(2, X0), (2.0, Z1)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0", 2) + QubitOperator("Z1", 2)
>>> op2 = 3 * QubitOperator.identity()
>>> qol = QubitOperatorList([(5, op1), (7, op2)])
>>> result = qol.compress_scalars_as_product(inner_coefficient=True)
>>> print(result)
1         [(21.0, )],
5         [(2, X0), (2.0, Z1)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0", 2) + QubitOperator("Z1", 2)
>>> op2 = 3 * QubitOperator.identity()
>>> qol = QubitOperatorList([(5, op1), (7, op2)])
>>> result = qol.compress_scalars_as_product(coefficients_to_compress="outer")
>>> print(result)
105.0     [(1.0, )],
1.0       [(2, X0), (2.0, Z1)]
```

(continues on next page)

(continued from previous page)

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0", 2) + QubitOperator("Z1", 2)
>>> op2 = 3 * QubitOperator.identity()
>>> qol = QubitOperatorList([(5, op1), (7, op2)])
>>> result = qol.compress_scalars_as_product(coefficients_to_compress="all")
>>> print(result)
210.0     [(1.0, )],
1.0       [(1.0, X0), (1.0, Z1)]
```

---

copy()

Returns a deep copy of self.

Return type
LinearListCombiner

df()

Returns a Pandas DataFrame object of the dictionary.

empty()

Checks if internal list is empty.

Return type
bool

evalf(**args*, ***kwargs*)

Numerically evaluates symbolic expressions stored in the left and right values of list items and replaces them with the results.

Parameters

- **args** (Any) – Args to be passed to *sympy.evalf()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.evalf()*.

Returns
LinearListCombiner – Updated instance of LinearListCombiner.

free_symbols()

Returns the free symbols in the coefficient values.

Return type
set

free_symbols_ordered()

Returns the free symbols in the list, ordered alphabetically.

Returns
SymbolSet – Ordered set of symbols.

classmethod from_Operator(*input*, *additional_coefficient=1.0*, *coefficients_location=FactoryCoefficientsLocation.INNER*)

Converts an Operator to an OperatorList with terms in arbitrary order.

Each term in the Operator is split into a separate component Operator in the OperatorList. The resulting location of each scalar coefficient in the input Operator can be controlled with the coefficients_location parameter. Setting this to 'inner' will leave coefficients stored as part of the component operators, 'outer' will move the coefficients to the 'outer' level.

Parameters

- **input** (Operator) – The input operator to split into an OperatorList.
- **additional_coefficient** (Union[int, float, complex, Expr], default: 1.0) – An additional factor to include in the "outer" coefficients of the generated OperatorList.
- **coefficients_location** (*FactoryCoefficientsLocation*, default: FactoryCoefficientsLocation.INNER) – The destination of the coefficients of the input operator, as described above.

Returns

TypeVar(OperatorListT, bound= OperatorList) – An OperatorList as described above.

Raises

ValueError – On invalid input to the coefficients_location parameter.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op = QubitOperator("X0", 2.) + QubitOperator("Z1", 2.)
>>> qol = QubitOperatorList.from_Operator(op)
>>> print(qol)
1.0       [(2.0, X0)],
1.0       [(2.0, Z1)]
>>> op = QubitOperator("X0", 2.) + QubitOperator("Z1", 2.)
>>> qol = QubitOperatorList.from_Operator(op,coefficients_location='outer')
>>> print(qol)
2.0       [(1.0, X0)],
2.0       [(1.0, Z1)]
```

classmethod from_string(*input_string*)

Constructs a child class instance from a string.

Parameters

input_string (str) – String in the format coeff1 [(coeff1_1, term1_1), ..., (coeff1_n, term1_n)], ..., coeffn [(coeffn_1, termn_1), ...].

Returns

Child class object.

infer_num_spin_orbs()

Returns the number of modes that the component operators act upon, inferring the existence of modes with index from 0 to the maximum index.

Returns

int – The minimum number of spin orbitals in the Fock space to which this operator list operates on.

Examples

```
>>> op1 = FermionOperator(FermionOperatorString(((1, 0), (2, 1))), 1.)
>>> op2 = FermionOperator(FermionOperatorString(((0, 0), (6, 1))), 1.)
>>> fto = FermionOperatorList([(1., op1), (1., op2)])
>>> print(fto.infer_num_spin_orbs())
7
```

is_empty()

Return True if operator is 0, else False.

Return type
bool

is_symmetry_of(*operator*)

Check if operator is symmetry of given operator.

Checks by determining if operator commutes with all terms of other operator. True if it commutes, False otherwise.

Parameters
operator (*FermionOperator*) – Operator to compare to.

Returns
bool – True if this is a symmetry of operator, otherwise False.

Danger: This calls *to_symmetry_operator_fermionic()* and thus may scale exponentially!

items()

Returns internal list.

Return type
List[Tuple[Any, Union[int, float, complex, Expr]]]

make_hashable()

Return a hashable representation of the object.

Currently simply returns a stringified object.

Return type
str

map(*mapping*)

Updates list items right values in-place, using a mapping function provided.

Parameters
mapping (Callable[[Union[int, float, complex, Expr]], Union[int, float, complex, Expr]]) – A callable object which takes each original value and returns the corresponding new value.

Return type
LinearListCombiner

property n_symbols: int

Returns the number of free symbols in the object.

Return type
int

property num_spin_orbs: int

Return the number of spin-orbitals that this operator explicitly acts on.

Return type
int

operator_class

alias of *FermionOperator*

print_table()

Print internal list formatted as a table.

Return type
None

qubit_encode(*mapping=None*, *qubits=None*)

Performs qubit encoding (mapping), using provided mapping function, of the current *FermionOperatorList*.

Terms are treated and mapped independently.

Parameters

- **mapping** (QubitMapping, default: None) – Mapping class. Default mapping procedure is the Jordan-Wigner transformation.
- **qubits** (Optional[List[Qubit]], default: None) – The qubit register. If left as None, a default register will be assumed if possible. See the mapping class documentation for further details.

Returns
QubitOperatorList – Mapped *QubitOperatorList*.

retrotterize(*new_trotter_number*, *initial_trotter_number=1*, *new_trotter_order=1*, *initial_trotter_order=1*, *constant=1.0*, *inner_coefficients=False*)

Retrotterize an expression given a OperatorList representing a product of exponentials.

This method assumes that the OperatorList represents a product of exponentials, with each Operator in the list corresponding to an exponent of a single exponential in the product. Scalar factors within the OperatorList are treated as scalar multipliers within each exponent.

The OperatorList is first untrotterized using the provided initial_trotter_number and initial_trotter_order, then subsequently Trotterized using the provided new_trotter_number and new_trotter_order. The returned OperatorList corresponds to the generated product of exponentials, in a similar manner to the original OperatorList.

Parameters

- **new_trotter_number** (int) – The desired number of Trotter steps in the final Trotter-Suzuki expansion.
- **initial_trotter_number** (int, default: 1) – The number of Trotter steps in the original Trotter-Suzuki expansion.
- **new_trotter_order** (int, default: 1) – The order of the original Trotter-Suzuki expansion used. Currently, only a first order (ABABAB…) expansion is supported.
- **initial_trotter_order** (int, default: 1) – The desired order of the final Trotter-Suzuki expansion. Currently, only a first order (ABABAB…) expansion is supported.
- **constant** (Union[float, complex], default: 1.0) – An additional constant multiplier in the exponent.
- **inner_coefficients** (bool, default: False) – By default, generated scalar factors in each exponent are stored in the coefficients of the generated OperatorList, with the coefficient of each inner Operator unchanged. Set this to True to instead store all scalar factors as coefficients in each Operator, with the outer coefficients of the OperatorList left unchanged. See examples for a comparison.

Returns

TypeVar(OperatorListT, bound= OperatorList) – The exponential product retrotterized with the provided new Trotter number and order. Each element is an individual exponent.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> retrotterised = qol.retrotterize(new_trotter_number=4,initial_trotter_
↪number=2)
>>> print(retrotterised)
0.25       [(1.0, X0 X1)],
0.25       [(1.0, Z0)],
0.25       [(1.0, X0 X1)],
0.25       [(1.0, Z0)],
0.25       [(1.0, X0 X1)],
0.25       [(1.0, Z0)],
0.25       [(1.0, X0 X1)],
0.25       [(1.0, Z0)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> retrotterised = qol.retrotterize(new_trotter_number=4,initial_trotter_
↪number=2,inner_coefficients=True)
>>> print(retrotterised)
0.5        [(0.5, X0 X1)],
0.5        [(0.5, Z0)],
0.5        [(0.5, X0 X1)],
0.5        [(0.5, Z0)],
0.5        [(0.5, X0 X1)],
0.5        [(0.5, Z0)],
0.5        [(0.5, X0 X1)],
0.5        [(0.5, Z0)]
```

reversed_order()

Reverses internal list order and returns it as a new object.

Return type

LinearListCombiner

simplify(**args*, ***kwargs*)

Simplifies expressions stored in left and right values of list items.

Parameters

- **args** (Any) – Args to be passed to *sympy.simplify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.simplify()*.

Returns

LinearListCombiner – Updated instance of LinearListCombiner.

split()

Generates pair objects from list items.

Return type
Iterator[LinearListCombiner]

sublist (*sublist_indices*)

Returns a new instance containing a subset of the terms of the original object.

Parameters
sublist_indices (list[int]) – List of indices of the operator list elements to constitute a new object.

Returns
TypeVar(OperatorListT, bound= OperatorList) – A new instance of OperatorList, being a sublist of the original operator.

Raises
ValueError – If sublist_indices contains indices not contained in self, or self is empty.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6)
>>> op2 = QubitOperator("Z0", -1.6j)
>>> op3 = QubitOperator("Z1 Z2 Z3 Z5", -5.6j)
>>> long_operator = QubitOperatorList([(1, op1), (1, op2), (1, op3)])
>>> short_operator = long_operator.sublist([0, 2])
>>> print(short_operator)
1          [(4.6, X0 Y1 Z3)],
1          [(-5.6j, Z1 Z2 Z3 Z5)]
```

subs (*symbol_map*)

Returns a new objects with symbols substituted.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution (*symbol_map=None*)

Substitutes free symbols for numerical values according to a map.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) –

Return type
LinearListCombiner

symmetry_sector (*state*)

Find the symmetry sector that a fermionic state is in by direct expectation value calculation.

As all terms commute, we take the product of their individual expectation values. For certain symmetry operators (e.g. parity operators) this should be polynomially hard.

Parameters
state (*FermionState*) – Input fermionic state.

Returns
Union[complex, float, int] – The symmetry sector of the state (i.e. the expectation value).

Danger: Due to direct expectation value calculation, this may scale exponentially in the general case, although it should be okay for sparse states and operators e.g. Z2 symmetries on number states.

sympify(**args*, ***kwargs*)

Sympifies left and right values of list items.

Replaces left and right values with their corresponding symbolic expressions.

Parameters
- **args** (Any) – Args to be passed to *sympy.sympify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.sympify()*.

Returns
LinearListCombiner – Updated instance of LinearListCombiner.

Raises
RuntimeError – when sympification fails.

to_symmetry_operator_fermionic()

Convert to a *SymmetryOperatorFermionic*.

Danger: This may scale exponentially depending on the operator!

Return type
SymmetryOperatorFermionic

trotterize_as_linear_combination(*trotter_number*, *trotter_order=1*, *constant=1.0*, *inner_coefficients=False*)

Trotterize an exponent linear combination of Operators.

This method assumes that the OperatorList represents the exponential of a linear combination of Operators, with each Operator within the OperatorList corresponding to a term in this linear combination. Trotterization is performed at the level of these Operators. The Operators contained within the returned OperatorList correspond to exponents within the Trotter sequence.

Parameters
- **trotter_number** (int) – The number of Trotter steps in the Trotter-Suzuki expansion.
- **trotter_order** (int, default: 1) – The order of the Trotter-Suzuki approximation to be used. Presently, only first-order (ABABAB…) is supported.
- **constant** (Union[float, complex], default: 1.0) – An additional constant multiplier in the exponent.
- **inner_coefficients** (bool, default: False) – By default, generated scalar factors in each exponent are stored in the coefficients of the generated OperatorList, with the coefficient of each inner Operator unchanged. Set this to True to instead store all scalar factors as coefficients in each Operator, with the outer coefficients of the OperatorList left unchanged. See examples for a comparison.

Returns

TypeVar(OperatorListT, bound= OperatorList) – A Trotterized form of the exponential product, where each element is an individual exponent.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1.,op1),(1.,op2)])
>>> result = qol.trotterize_as_linear_combination(2)
>>> print(result)
0.5       [(1.0, X0 X1)],
0.5       [(1.0, Z0)],
0.5       [(1.0, X0 X1)],
0.5       [(1.0, Z0)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1.,op1),(1.,op2)])
>>> result = qol.trotterize_as_linear_combination(2,inner_coefficients=True)
>>> print(result)
1.0       [(0.5, X0 X1)],
1.0       [(0.5, Z0)],
1.0       [(0.5, X0 X1)],
1.0       [(0.5, Z0)]
```

unsympify()

Unsympifies dicionary values.

Replaces symbolic expressions with their corresponding numeric values.

Returns

LinearListCombiner – Updated instance of LinearListCombiner.

Raises

TypeError – When unsympification fails.

untrotterize(*trotter_number*, *trotter_order=1*)

Reverse a Trotter-Suzuki expansion given an OperatorList representing a product of exponentials.

This method assumes that the OperatorList represents a product of exponentials, with each Operator in the list corresponding to an exponent of a single exponential in the product. Scalar factors within the OperatorList are treated as scalar multipliers within each exponent. An Operator corresponding to the exponent of a single, untrotterized exponential is returned.

Parameters

- **trotter_number** (int) – The number of Trotter steps within the Trotter expansion to be reversed.
- **trotter_order** (int, default: 1) – The order of the Trotter-Suzuki expansion used. Currently, only a first order (ABABAB…) expansion is supported.

Returns

TypeVar(OperatorT, bound= Operator) – The exponent of the untrotterised operator.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> untrotterised = qol.untrotterize(2)
>>> print(untrotterised)
(1.0, X0 X1), (1.0, Z0)
```

untrotterize_partitioned(*trotter_number*, *trotter_order=1*, *inner_coefficients=False*)

Reverse a Trotter-Suzuki expansion given a OperatorList representing a product of exponentials, maintaining separation of exponents.

This method assumes that the OperatorList represents a product of exponentials, with each Operator in the list corresponding to an exponent of a single exponential in the product. Scalar factors within the OperatorList are treated as scalar multipliers within each exponent. A OperatorList is returned wherein each term represents a single term in the exponent of the single, untrotterized exponential.

Parameters

- **trotter_number** (int) – The number of Trotter steps within the Trotter expansion to be reversed.
- **trotter_order** (int, default: 1) – The order of the Trotter-Suzuki expansion used. Currently, only a first order (ABABAB…) expansion is supported.
- **inner_coefficients** (bool, default: False) – By default, generated scalar factors in each exponent are stored in the coefficients of the generated OperatorList, with the coefficient of each inner Operator unchanged. Set this to True to instead store all scalar factors as coefficients in each Operator, with the outer coefficients of the OperatorList left unchanged. See examples for a comparison.

Returns

TypeVar(OperatorListT, bound= OperatorList) – The terms in the exponent of the untrotterised operator as a OperatorList.

Raises

ValueError – If the provided Trotter number is not compatible with the OperatorList.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> untrotterised = qol.untrotterize_partitioned(2)
>>> print(untrotterised)
1.0       [(1.0, X0 X1)],
1.0       [(1.0, Z0)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> untrotterised = qol.untrotterize_partitioned(2,inner_coefficients=True)
>>> print(untrotterised)
0.5       [(2.0, X0 X1)],
0.5       [(2.0, Z0)]
```

class SymmetryOperatorPauli(**args*, ***kwargs*)

Bases: *QubitOperator*, SymmetryOperator

A class for representing symmetry in a qubit space.

This is largely an extension of *QubitOperator*, providing functionality relating to validating symmetries and finding symmetry sectors.

Parameters

- **data** – Data defined as a string "X0 Y1", iterable of tuples ((0, 'Y'), (1, 'X')), *QubitOperatorString*, or as a dictionary of *QubitOperatorString* and CoeffType objects.
- **coeff** – Coefficient attached to data.

class TrotterizeCoefficientsLocation(*value*)

Bases: str, Enum

Determines where coefficients will be stored upon performing Trotterization.

INNER = 'inner'

All coefficients will be stored in the "inner" coefficients, within the component QubitOperators in the Trotterized result.

MIXED = 'mixed'

The Trotter step factor will be stored in the "outer" coefficients, whereas the original coefficients of the original Operator will remain in the component Operators.

OUTER = 'outer'

All coefficients will be stored in the "outer" coefficients, the coefficients stored directly in the generated OperatorList.

property all_qubits: Set[Qubit]

The set of all qubits the operator ranges over (including qubits that were provided explicitly as identities)

Return type
Set[Qubit]

Type
return

anticommutator(*other_operator*, *abs_tol=None*)

Calculates the anticommutator with another *QubitOperator*, within a tolerance.

Parameters

- **other_operator** (*QubitOperator*) – The other *QubitOperator*.
- **abs_tol** (Optional[float], default: None) – Threshold below which terms are deemed negligible.

Returns
QubitOperator – The anticommutator of the two operators.

anticommutes_with(*other_operator*, *abs_tol=1e-10*)

Calculates whether operator anticommutes with another *QubitOperator*, within a tolerance.

If both operators are single Pauli strings, we use tket's .commutes_with() method and flip the result. Otherwise, it calculates the whole anticommutator and checks if it is zero.

Parameters

- **other_operator** (*QubitOperator*) – The other *QubitOperator*.
- **abs_tol** (float, default: 1e-10) – Threshold below which terms are deemed negligible.

Returns

bool – True if operators anticommute, within tolerance, otherwise False.

antihermitian_part()

Return the anti-Hermitian (all imaginary-coefficient terms) part of the original *QubitOperator*.

In case the original *QubitOperator* object contains symbolic coefficients that do not have an associated type, those will be cast to the imaginary Expr type.

Examples

```
>>> qo = QubitOperator.from_string("(1.0, X0 Y1), (0.1j, Y0 X1), (0.5 + 0.2j,␣
↪Z0 Z1)")
>>> print(qo.antihermitian_part())
(0.1j, Y0 X1), (0.2j, Z0 Z1)
>>> a = Symbol('a', real=True)
>>> b = Symbol('b', imaginary=True)
>>> c = Symbol('c')
>>> p_str_a = QubitOperatorString.from_string("X0 Y1")
>>> p_str_b = QubitOperatorString.from_string("Y0 X1")
>>> p_str_c = QubitOperatorString.from_string("Z0 Z1")
>>> qo = QubitOperator({p_str_a: a, p_str_b: b, p_str_c: c})
>>> print(qo.antihermitian_part())
(b, Y0 X1), (1.0*I*im(c), Z0 Z1)
```

Return type

QubitOperator

approx_equal_to(*other*, *abs_tol=1e-10*)

Checks if object's dictionary values are numerically identical to the other object values.

Parameters

- **other** (LinearDictCombiner) – Object to compare to.
- **abs_tol** (float, default: 1e-10) – Threshold of comparing numeric values.

Raises

TypeError – When comparison of two values can't be done due to types mismatch.

Return type

bool

approx_equal_to_by_random_subs(*other*, *order=1*, *abs_tol=1e-10*)

Checks if object's dictionary values are numerically identical to the other object values.

Symbols contained in the difference of the two objects, if any, are substituted by random numeric values prior to norm check.

Parameters

- **other** (LinearDictCombiner) – Object to compare to.

- **order** (int, default: 1) – Parameter specifying the norm formula (see numpy.linalg.norm documentation).
- **abs_tol** (float, default: 1e-10) – Threshold vs which the norm of the difference is checked.

Return type
bool

as_scalar(*abs_tol=None*)

If the operator is a sum of identity terms or zero, return the sum of the coefficients, otherwise return None.

Note that this does not perform combination on terms and will return zero only if all coefficients are zero.

Parameters
abs_tol (Optional[float], default: None) – Tolerance for checking if coefficients are zero. Set to None to test using a standard python == comparison.

Returns
Union[float, complex, None] – The operator as a scalar if it can be represented as such, otherwise None.

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

property coefficients: List[int | float | complex | Expr]

Returns dictionary values.

Return type
List[Union[int, float, complex, Expr]]

commutator(*other_operator*, *abs_tol=None*)

Calculate the commutator with another operator.

Computes commutator. Small terms in the result may be discarded.

Parameters
- **other_operator** (*QubitOperator*) – The other *QubitOperator*.
- **abs_tol** (Optional[float], default: None) – Threshold below which terms are discarded. Set to a negative value to skip.

Returns
QubitOperator – The commutator.

commutes_with(*other_operator*, *abs_tol=1e-10*)

Calculates whether operator commutes with another *QubitOperator*, within a tolerance.

If both operators are single Pauli strings, we use tket. Otherwise, it calculates the whole commutator and checks if it is zero.

Parameters
- **other_operator** (*QubitOperator*) – The other *QubitOperator*.
- **abs_tol** (float, default: 1e-10) – Threshold below which terms are deemed negligible.

Returns
bool – True if operators commute, within tolerance, otherwise False.

compress(*abs_tol=1e-10*, *symbol_sub_type=CompressSymbolSubType.NONE*)

Adapted from `pytket.QubitPauliOperator` to account for non-sympy coefficients.

Parameters

- **abs_tol** (`float`, default: `1e-10`) – The threshold below which to remove values.
- **symbol_sub_type** (`CompressSymbolSubType`, default: `CompressSymbolSubType.NONE`) – Defines the behaviour for dealing with symbolic expressions in coefficients. If "none", symbolic expressions are left intact. If "unity", substitutes all free symbols with 1, and removes all imaginary and real components below tolerance. If "random", substitutes all free symbols with a random number between 0 and 1, and removes imaginary and real components below tolerance.

Warning: When :code: *symbol_sub_type != "none"*, this method assumes significant expression structure is known a priori, and is best suited to operators which have simple product expressions, such as excitation operators for VQE ansätze and digital quantum simulation. Otherwise, it may remove terms relevant to computation. Each expression is of the form $f(a_1, a_2,$
$ldots, a_n)$ for some symbols a_i. $|f(a_1, a_2,$
$ldots, a_n)|$ is assumed to monotonically increase in both real and imaginary components for all a_i
$in[0, 1]$.

Return type
`None`

copy()

Performs deep copy of the object.

Return type
`TypeVar`(SYMBOLICTYPE, bound= Symbolic)

dagger()

Return the Hermitian conjugate of *QubitOperator*.

Return type
QubitOperator

df()

Returns a Pandas DataFrame object of the dictionary.

Return type
`DataFrame`

dot_state(*state*, *qubits=None*)

Calculate the result of operating on a given qubit state.

Can accept right-hand state as a `QubitState`, `QubitStateString` or a `numpy.ndarray`. In the former two cases, we maintain a symbolic representation of both operator and state, and each Pauli is implemented in sequence. Resultant states are returned as `QubitState`. This should support sympy parametrised states & operators, but the use of parametrised states & operators is untested. In this case, the optional qubits parameter is ignored.

For a `numpy.ndarray`, we hand off to pytket's `QubitPauliOperator.dot_state()` method - this should be faster for dense states, but slower for sparse ones. Here, a register of qubits may be specified in the `qubits` parameter to determine the meaning of the indices of the provided state vector. From the pytket documentation:

- When `qubits` is an explicit list, the qubits are ordered with `qubits[0]` as the most significant qubit for indexing into state.
- If `None`, qubits sequentially indexed from 0 in the default register and ordered by ILO-BE so `Qubit(0)` is the most significant.

Parameters

- **state** (Union[*QubitState*, *QubitStateString*, ndarray]) – Input qubit state to operated on.
- **qubits** (Optional[List[Qubit]], default: None) – For ndarray input, determines sequencing of qubits in the state, if not mapped to the default register. Ignored for other input types.

Returns

Union[*QubitState*, ndarray] – Output state.

empty()

Checks if dictionary is empty.

Return type

bool

evalf(**args*, ***kwargs*)

Evaluates symbolic expressions stored in dict values and replaces them with the results.

Parameters

- **args** (Any) – Args to be passed to *sympy.evalf()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.evalf()*.

Returns

LinearDictCombiner – Updated instance of LinearDictCombiner.

exponentiate_commuting_operator(*additional_exponent=1.0*, *check_commuting=True*)

Exponentiate a QubitOperator where all terms commute, returning as a product of operators.

As all terms are mutually commuting, exponentiation reduces to a product of exponentials of individual terms (i.e. $e^{\sum_i P_i} = \prod_i e^{P_i}$). Each individual exponential can further be expanded trigonometrically. While storing these as a product is efficient, expanding the product will result in an exponential number of terms, and thus this method returns the result in factorised form, as a QubitOperatorList.

Parameters

- **additional_exponent** (complex, default: 1.0) – Optional additional factor in exponent.
- **check_commuting** (bool, default: True) – Set to False to skip checking whether all terms commute.

Returns

QubitOperatorList – The exponentiated operator in factorised form.

Raises

ValueError – If commutativity checking is performed and the operator is not a commuting set of terms.

exponentiate_single_term(*additional_exponent=1.0*, *coeff_cutoff=1e-14*)

Exponentiates a single weighted Pauli string through trigonometric expansion.

This will except if the operator contains more than one term. It will attempt to maintain single term if rotation is sufficiently close to an integer multiple of pi/2. Set coeff_cutoff to None to disable this behaviour.

Parameters

- **additional_exponent** (complex, default: 1.0) – Optional additional factor in exponent.
- **coeff_cutoff** (Optional[float], default: 1e-14) – If a Pauli string is weighted by an integer multiple of pi/2 and exponentiated, the resulting expansion will have a single Pauli term (as opposed to two). If this parameter is not None, it will be used to determine a threshold for cutting off negligible terms in the trigonometric expansion to avoid floating point errors resulting in illusory growth in the number of terms. Set to None to disable this behaviour.

Returns

QubitOperator – The exponentiated operator.

Raises

ValueError – If the operator is not a single term.

free_symbols()

Returns the free symbols in the coefficient values.

free_symbols_ordered()

Returns the free symbols in the dict, ordered alphabetically.

Returns

SymbolSet – Ordered set of symbols.

classmethod from_list(*pauli_list*)

Construct a QubitPauliOperator from a serializable JSON list format, as returned by QubitPauliOperator.to_list()

Returns

New QubitPauliOperator instance.

Return type

QubitPauliOperator

Parameters

pauli_list (List[Dict[str, Any]]) –

classmethod from_string(*input_string*)

Constructs a child class instance from a string.

Parameters

input_string (str) – String in the format `coeff1 [(coeff1_1, term1_1), ..., (coeff1_n, term1_n)], ..., coeffn [(coeffn_1, termn_1), ...]`.

Returns

Child class object.

get(*key*, *default*)

Parameters

- **key** (QubitPauliString) –

- **default** (Union[int, float, complex, Expr]) –

Return type
Union[int, float, complex, Expr]

hermitian_factorisation()

Returns a tuple of the real and imaginary parts of the original *QubitOperator*.

For example, both P and Q from $O = P + iQ$.

In case the original *QubitOperator* object contains symbolic coefficients that do not have an associated type, those will be cast into both the real and imaginary Expr types and assigned to both objects (imaginary component will be multiplied by $-I$ in order to return its real part).

Returns
Tuple[*QubitOperator*, *QubitOperator*] – Real and imaginary parts of the qubit operator.

Examples

```
>>> qo = QubitOperator.from_string("(1.0, X0 Y1), (0.1j, Y0 X1), (0.5 + 0.2j,␣
↪Z0 Z1)")
>>> re_qo, im_qo = qo.hermitian_factorisation()
>>> print(re_qo)
(1.0, X0 Y1), (0.5, Z0 Z1)
>>> print(im_qo)
(0.1, Y0 X1), (0.2, Z0 Z1)
>>> a = Symbol('a', real=True)
>>> b = Symbol('b', imaginary=True)
>>> c = Symbol('c')
>>> p_str_a = QubitOperatorString.from_string("X0 Y1")
>>> p_str_b = QubitOperatorString.from_string("Y0 X1")
>>> p_str_c = QubitOperatorString.from_string("Z0 Z1")
>>> qo = QubitOperator({p_str_a: a, p_str_b: b, p_str_c: c})
>>> re_qo, im_qo = qo.hermitian_factorisation()
>>> print(re_qo)
(a, X0 Y1), (re(c), Z0 Z1)
>>> print(im_qo)
(-I*b, Y0 X1), (im(c), Z0 Z1)
```

hermitian_part()

Return the Hermitian (all real-coefficient terms) part of the original *QubitOperator*.

In case the original *QubitOperator* object contains symbolic coefficients that do not have an associated type, those will be cast to the real Expr type.

Examples

```
>>> qo = QubitOperator.from_string("(1.0, X0 Y1), (0.1j, Y0 X1), (0.5 + 0.2j,␣
↪Z0 Z1)")
>>> print(qo.hermitian_part())
(1.0, X0 Y1), (0.5, Z0 Z1)
>>> a = Symbol('a', real=True)
>>> b = Symbol('b', imaginary=True)
>>> c = Symbol('c')
>>> p_str_a = QubitOperatorString.from_string("X0 Y1")
```

(continues on next page)

(continued from previous page)

```
>>> p_str_b = QubitOperatorString.from_string("Y0 X1")
>>> p_str_c = QubitOperatorString.from_string("Z0 Z1")
>>> qo = QubitOperator({p_str_a: a, p_str_b: b, p_str_c: c})
>>> print(qo.hermitian_part())
(a, X0 Y1), (re(c), Z0 Z1)
```

Return type
QubitOperator

classmethod identity()

Return an identity operator.

Examples

```
>>> print(QubitOperator.identity())
(1.0, )
```

Return type
QubitOperator

is_all_coeff_complex()

Check if all coefficients are complex values.

Warning: Returns None if there is a free symbol in a coefficient.

Return type
bool

is_all_coeff_imag()

Check if all coefficients are complex values.

Warning: Returns None if there is a free symbol in a coefficient.

Return type
bool

is_all_coeff_real()

Check if all coefficients are real values.

Warning: Returns None if there is a free symbol in a coefficient.

Return type
bool

is_all_coeff_symbolic()

Check if all coefficients contain free symbols.

Return type
bool

is_antihermitian()

Check if operator is antihermitian.

Check is performed by taking the Hermitian conjugate of operator and testing for opposite equality.

Returns
bool – True if antihermitian, False otherwise.

is_any_coeff_complex()

Check if any coefficient is a complex value.

Warning: Returns None if there is a free symbol in a coefficient.

Return type
bool

is_any_coeff_imag()

Check if any coefficient is an imaginary value.

Warning: Returns None if there is a free symbol in a coefficient.

Return type
bool

is_any_coeff_real()

Check if any coefficient is a real value.

Warning: Returns None if there is a free symbol in a coefficient.

Return type
bool

is_any_coeff_symbolic()

Check if any coefficient contains a free symbol.

Return type
bool

is_commuting_operator()

Return True if every term in operator commutes with every other term, otherwise False.

Return type
bool

is_hermitian()

Check if operator is Hermitian.

Check is performed by taking the Hermitian conjugate of operator and testing for equality.

Returns
bool – True if Hermitian, False otherwise.

is_parallel_with(*other*, *abs_tol=1e-10*)

Returns True if other is parallel with this (i.e. a scalar multiple of this), otherwise False.

Parameters

- **other** (LinearDictCombiner) – The other object to compare against
- **abs_tol** (Optional[float], default: 1e-10) – Tolerance threshold for comparison. Set to None to test for exact equivalence.

Returns
bool – True if other is parallel with this, otherwise False.

is_self_inverse(*abs_tol=1e-10*)

Check if operator is its own inverse.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold for comparison with identity.

Returns
bool – True if self-inverse, False otherwise.

is_symmetry_of(*operator*)

Check if operator is symmetry of given operator.

Checks by determining if operator commutes with all terms of other operator. True if it commutes, False otherwise.

Parameters
operator (*QubitOperator*) – Operator to compare to.

Returns
bool – True if this is a symmetry of operator, otherwise False.

is_unit_1norm(*abs_tol=1e-10*)

Returns True if operator has unit 1-norm, else False.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold for comparison with unity.

Return type
bool

is_unit_2norm(*abs_tol=1e-10*)

Returns True if operator has unit 1-norm, else False.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold for comparison with unity.

Return type
bool

is_unit_norm(*order=2*, *abs_tol=1e-10*)

Returns True if operator has unit p-norm, else False.

Parameters

- **order** (int, default: 2) – Norm order.
- **abs_tol** (float, default: 1e-10) – Tolerance threshold for comparison with unity.

Raises
ValueError – Coefficients contain free symbols.

Return type
bool

is_unitary(*abs_tol=1e-10*)

Check if operator is unitary.

Checking is performed by multiplying the operator by its Hermitian conjugate and comparing against the identity.

Parameters
abs_tol – Tolerance threshold for comparison with identity.

Returns
bool – True if unitary, False otherwise.

items()

Returns dictionary items.

Return type
ItemsView[Any, Union[int, float, complex, Expr]]

static key_from_str(*key_str*)

Returns a *QubitOperatorString* instance initialised from the input string.

Parameters
key_str (str) – Input python string.

Returns
QubitOperatorString – Operator string initialised from input.

list_class

alias of *QubitOperatorList*

make_hashable()

Return a hashable representation of the object.

Currently simply returns a stringified object.

Return type
str

map(*mapping*)

Updates dictionary values, using a mapping function provided.

Parameters
mapping (Callable[[Union[int, float, complex, Expr]], Union[int, float, complex, Expr]]) – Mapping function to update the dict.

Returns
LinearDictCombiner – None.

property n_symbols: int

Returns the number of free symbols in the object.

Return type
int

norm_coefficients(*order=2*)

Returns the p-norm of the coefficients.

Parameters
order (int, default: 2) – Norm order.

Return type
Union[complex, float]

normalized(*norm_value=1.0*, *norm_order=2*)

Returns a copy of this object with normalised coefficients.

Parameters

- **norm_value** (float, default: 1.0) – The desired norm of the returned operator.
- **norm_order** (int, default: 2) – The order of the norm to be used.

Returns
LinearDictCombiner – A copy of the object with coefficients normalised to the desired value.

pad(*register_qubits=None*, *zero_to_max=False*)

Modify *QubitOperator* in-place by replacing implicit identities with explicit identities.

By default, this will assume a minimal register - i.e. the register contains only qubits acted on by any term in the *QubitOperator*. A specific register of qubits may be provided by setting the register_qubits parameter. This must contain all qubits acted on by the *QubitOperator*. Alternatively, zero_to_max may be set to True in order to assume that the qubit register is indexed 0-N, where N is the highest integer indexed qubit in the original *QubitOperator*. These modes of operation are incompatible and this method will except if zero_to_max is set to True and register_qubits is provided. See *padded()* for a non-in-place version of this method.

Parameters

- **register_qubits** (Optional[Iterable[Qubit]], default: None) – A qubit register used to determine which padding identities will be added.
- **zero_to_max** (bool, default: False) – Set to True to assume a 0-N indexed qubit register as described above.

Raises

- **PaddingIncompatibleArgumentsError** – If register_qubits has been provided while zero_to_max is set to True.
- **PaddingInferenceError** – If zero_to_max is set to True and maximum qubit index cannot be inferred, for instance by non-integer labelled qubits.
- **PaddingInvalidRegisterError** – If *QubitOperator* acts on qubits not in provided register.

Examples

```
>>> qs = QubitOperator("X0") + QubitOperator("X1")
>>> qs.pad()
>>> print(qs)
(1.0, X0 I1), (1.0, I0 X1)
```

```
>>> qs = QubitOperator("X0")
>>> qs.pad([Qubit(0),Qubit(1)])
>>> print(qs)
(1.0, X0 I1)
```

```
>>> qs = QubitOperator("X1")
>>> print(qs.padded(zero_to_max=True))
(1.0, I0 X1)
```

Return type
None

padded (*register_qubits=None*, *zero_to_max=False*)

Return a copy of the *QubitOperator* with implicit identities replaced by explicit identities.

By default, this will assume a minimal register - i.e. the register contains only qubits acted on by any term in the *QubitOperator*. A specific register of qubits may be provided by setting the `register_qubits` parameter. This must contain all qubits acted on by the *QubitOperator*. Alternatively, `zero_to_max` may be set to True in order to assume that the qubit register is indexed 0-N, where N is the highest integer indexed qubit in the original *QubitOperator*. These modes of operation are incompatible and this method will except if `zero_to_max` is set to True and `register_qubits` is provided. See *pad()* for an in-place version of this method.

Parameters

- **register_qubits** (Optional[Iterable[Qubit]], default: None) – A qubit register used to determine which padding identities will be added.
- **zero_to_max** (bool, default: False) – Set to True to assume a 0-N indexed qubit register as described above.

Raises

- **PaddingIncompatibleArgumentsError** – If register_qubits has been provided while `zero_to_max` is set to True.
- **PaddingInferenceError** – If `zero_to_max` is set to True and maximum qubit index cannot be inferred, for instance by non-integer labelled qubits.
- **PaddingInvalidRegisterError** – If *QubitOperator* acts on qubits not in provided register.

Returns
QubitOperator – Modified *QubitOperator*.

Examples

```
>>> qs = QubitOperator("X0") + QubitOperator("X1")
>>> qs_padded = qs.padded()
>>> print(qs_padded)
(1.0, X0 I1), (1.0, I0 X1)
```

```
>>> qs = QubitOperator("X0")
>>> qs_padded = qs.padded([Qubit(0),Qubit(1)])
>>> print(qs_padded)
(1.0, X0 I1)
```

```
>>> qs = QubitOperator("X1")
>>> qs_padded = qs.padded(zero_to_max=True)
>>> print(qs_padded)
(1.0, I0 X1)
```

---

property pauli_strings: List[*QubitOperatorString*]

Return the Pauli strings within the operator sum as a list.

Return type
List[*QubitOperatorString*]

print_table()

Print dictionary formatted as a table.

Return type
NoReturn

qubitwise_anticommutes_with(*other_operator*)

Calculates whether two single-term QubitOperators qubit-wise anticommute.

Two Pauli strings qubit-wise anticommute if every Pauli acting on a given qubit anticommutes with the Pauli acting on the same qubit in the other string. This necessitates that the Paulis are different, and both non-identity, for every qubit.

This method is currently not defined for QubitOperators consisting of multiple terms.

Parameters
other_operator (*QubitOperator*) – The other single-term *QubitOperator*.

Returns
bool – True if the operators qubit-wise anticommute, otherwise False.

Raises
ValueError – If either operator consists of more than a single term.

qubitwise_commutes_with(*other_operator*)

Calculates whether two single-term QubitOperators qubit-wise commute.

Two Pauli strings qubit-wise commute if every Pauli acting on a given qubit commutes with the Pauli acting on the same qubit in the other string. This necessitates that either the Paulis are the same, or that at least one is an identity.

This method is currently not defined for QubitOperators consisting of multiple terms.

Parameters
other_operator (*QubitOperator*) – The other single-term *QubitOperator*.

Returns
bool – True if the operators qubit-wise commute, otherwise False.

Raises
ValueError – If either operator consists of more than a single term.

remove_global_phase(*phase=0.0*)

Returns a copy with a global phase applied such that the first element has the desired phase.

Parameters
phase (float, default: 0.0) – The phase to yield on the first element, in half-turns (i.e. multiples of pi).

Returns
LinearDictCombiner – A copy of the object with the desired global phase applied.

reversed_order()

Reverses the order of terms and returns it as a new object.

Return type
LinearDictCombiner

simplify(**args*, ***kwargs*)

Simplifies expressions stored in dictionary values.

Parameters

- **args** (Any) – Args to be passed to *sympy.simplify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.simplify()*.

Returns
LinearDictCombiner – Updated instance of LinearDictCombiner.

split()

Generates pair objects from dictionary items.

Return type
Iterator[LinearDictCombiner]

state_expectation(*state*, **args*, ***kwargs*)

Calculate expectation value of operator with input state.

Can accept right-hand state as a QubitState or a numpy.ndarray. In the first case, we maintain a symbolic representation of both operator and state, and each Pauli is implemented in sequence. This should support sympy parametrised states & operators, but the use of parametrised states & operators is untested. For a numpy.ndarray, we hand off to pytket's QubitPauliOperator.dot_state() method and return a numpy.ndarray - this should be faster for dense states, but slower for sparse ones.

Parameters

- **state** (Union[*QubitState*, ndarray]) – The state to be acted upon.
- ***args** – Additional arguments passed to pytket.utils.operators.QubitPauliOperator.state_expectation().
- ****kwargs** – Additional keyword arguments passed to pytket.utils.operators.QubitPauliOperator.state_expectation().

Returns
complex – Expectation value of *QubitOperator*.

subs(*symbol_map*)

In-place substitution for symbolic expressions.

Iterates through each Symbol and performs the substitution.

Parameters
symbol_map (Union[*SymbolDict*, Dict[Union[str, Symbol], Union[float, complex, Expr, None]]]) – A map from SymPy symbols to SymPy expressions, floating-point or complex values.

Returns
QubitOperator – The operator after substitution.

symbol_substitution(*symbol_map=None*)

Substitutes free symbols for numerical values according to a map.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) –

Return type

LinearDictCombiner

symmetry_sector(*state*)

Find the symmetry sector that a qubit state is in by direct expectation value calculation.

Parameters

state (*QubitState*) – Input qubit state.

Warning: Validity is not checked and providing symmetry-broken states may lead to odd results.

Danger: Due to direct expectation value calculation, this may scale exponentially in the general case, although it should be okay for sparse states and operators e.g. Z2 symmetries on number states.

Returns

Union[float, complex, int] – The symmetry sector of the state (i.e. the expectation value)

sympify(**args*, ***kwargs*)

Sympifies dictionary values.

Replaces values with their corresponding symbolic expressions.

Parameters

- **args** (Any) – Args to be passed to *sympy.sympify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.sympify()*.

Returns

LinearDictCombiner – Updated instance of LinearDictCombiner.

Raises

RuntimeError – When sympification fails.

symplectic_representation(*qubits=None*)

Generate the symplectic representation of the operator.

This is a binary M*2N matrix where M is the number of terms and N is the number of qubits. In the leftmost half of the matrix, element (i,j) is True where qubit j is acted on by an X or a Y in term i, and otherwise is False. In the rightmost half, element (i,j) is True where qubit (j-num_qubits) is acted on by a Y or a Z in term i, and otherwise is False.

Notes

If qubits is not specified, a minimal register will be assumed and columns will be assigned to qubits in ascending numerical order by index.

Parameters

qubits (Optional[List[Qubit]], default: None) – A register of qubits. If not provided, a minimal register is assumed.

Returns

ndarray – The symplectic form of this operator

property terms: List[Any]

Returns dictionary keys.

Return type

List[Any]

to_latex(*imaginary_unit='\\\\text{i}'*, ***kwargs*)

Generate a LaTeX representation of the operator.

Parameters

- **imaginary_unit** (str, default: r"\\text{i}") – Symbol to use for the imaginary unit.
- ****kwargs** – Keyword arguments passed to the *to_latex()* method of component operator strings (*FermionOperatorString* or *QubitOperatorString*).

Returns

str – LaTeX compilable equation string.

Examples

```
>>> from inquanto.operators import FermionOperator
>>> from sympy import sympify
>>> c = sympify("c")
>>> fo = FermionOperator([(-c, "F1 F2^"), (c**2, "F1^"), (c, "F0 F5 F3^")])
>>> print(fo.to_latex())
- c a_{1} a_{2}^{\dagger} + c^{2} a_{1}^{\dagger} + c a_{0} a_{5} a_{3}^{\
↪dagger}
>>> fo = FermionOperator([(1.0, "F0^ F1^ F3^"), (-1.0j, "F3 F1 F0")])
>>> print(fo.to_latex())
a_{0}^{\dagger} a_{1}^{\dagger} a_{3}^{\dagger} -\text{i} a_{3} a_{1} a_{0}
>>> from sympy import sqrt
>>> fo = FermionOperator([(sqrt(2), "F0^ F1"), (0.5-8j, "F5 F5^"), (2j, "F1^␣
↪F1^")])
>>> print(fo.to_latex(imaginary_unit=r"\text{j}", operator_symbol="f"))
\sqrt{2} f_{0}^{\dagger} f_{1} + (0.5-8.0\text{j}) f_{5} f_{5}^{\dagger} + 2.
↪0\text{j} f_{1}^{\dagger} f_{1}^{\dagger}
```

```
>>> from inquanto.operators import QubitOperator, QubitOperatorString
>>> qos1 = QubitOperatorString.from_string("X1 Y2 Z3")
>>> qos2 = QubitOperatorString.from_string("Z0 Y1 X4")
>>> qos3 = QubitOperatorString.from_list([(("a", [0]), "I"), (("b", [1]), "Z
↪"), (("c", [2]), "Z")])
>>> qo = QubitOperator({qos1: -1.0, qos2: 3+4j, qos3: c})
>>> print(qo.to_latex())
- X_{1} Y_{2} Z_{3} + (3.0+ 4.0\text{i}) Z_{0} Y_{1} X_{4} + c I_{0} Z_{1} Z_
↪{2}
>>> qo = QubitOperator({qos3: 3j})
>>> print(qo.to_latex(imaginary_unit="j", show_labels=True))
3.0j I^{\text{a}}_{0} Z^{\text{b}}_{1} Z^{\text{c}}_{2}
```

to_list()

Generate a list serialized representation of QubitPauliOperator,
suitable for writing to JSON.

Returns
JSON serializable list of dictionaries.

Return type
List[Dict[str, Any]]

to_sparse_matrix(*qubits=None*)

Represents the sparse operator as a dense operator under the ordering scheme specified by `qubits`, and generates the corresponding matrix.

- When `qubits` is an explicit list, the qubits are ordered with `qubits[0]` as the most significant qubit for indexing into the matrix.
- If `None`, then no padding qubits are introduced and we use the ILO-BE convention, e.g. `Qubit("a", 0)` is more significant than `Qubit("a", 1)` or `Qubit("b")`.
- Giving a number specifies the number of qubits to use in the final operator, treated as sequentially indexed from 0 in the default register (padding with identities as necessary) and ordered by ILO-BE so `Qubit(0)` is the most significant.

Parameters
qubits (*Union[List[Qubit], int, None], optional*) – Sequencing of qubits in the matrix, either as an explicit list, number of qubits to pad to, or infer from the operator. Defaults to None

Returns
A sparse matrix representation of the operator.

Return type
csc_matrix

toeplitz_decomposition()

Returns a tuple of the Hermitian and anti-Hermitian parts of the original *QubitOperator*.

In case the original *QubitOperator* object contains symbolic coefficients that do not have an associated type, those will be cast into both the real and imaginary Expr types and assigned to both objects.

Returns
Tuple[*QubitOperator*] – Hermitian and anti-Hermitian parts of operator.

Examples

```
>>> qo = QubitOperator.from_string("(1.0, X0 Y1), (0.1j, Y0 X1), (0.5 + 0.2j,␣
↪Z0 Z1)")
>>> herm_qo, antiherm_qo = qo.toeplitz_decomposition()
>>> print(herm_qo)
(1.0, X0 Y1), (0.5, Z0 Z1)
>>> print(antiherm_qo)
(0.1j, Y0 X1), (0.2j, Z0 Z1)
>>> a = Symbol('a', real=True)
>>> b = Symbol('b', imaginary=True)
```

(continues on next page)

(continued from previous page)

```
>>> c = Symbol('c')
>>> p_str_a = QubitOperatorString.from_string("X0 Y1")
>>> p_str_b = QubitOperatorString.from_string("Y0 X1")
>>> p_str_c = QubitOperatorString.from_string("Z0 Z1")
>>> qo = QubitOperator({p_str_a: a, p_str_b: b, p_str_c: c})
>>> herm_qo, antiherm_qo = qo.toeplitz_decomposition()
>>> print(herm_qo)
(a, X0 Y1), (re(c), Z0 Z1)
>>> print(antiherm_qo)
(b, Y0 X1), (1.0*I*im(c), Z0 Z1)
```

totally_commuting_decomposition(*abs_tol=1e-10*)

Decomposes into two separate operators, one including the totally commuting terms and the other including all other terms.

This will return two QubitOperators. The first is comprised of all terms which commute with all other terms, the second comprised of terms which do not. An empty QubitOperator will be returned if either of these sets is empty.

Parameters

abs_tol (float, default: 1e-10) – Tolerance threshold used for determining commutativity - see .commutator() for details.

Returns

The totally commuting operator, and the remainder operator.

trotterize(*trotter_number=1*, *trotter_order=1*, *constant=1.0*, *coefficients_location=TrotterizeCoefficientsLocation.OUTER*)

Trotterizes the operator, treating the operator as an exponent.

Assuming that this operator is an exponent, this will generate an OperatorList with each element in the OperatorList corresponding to a single exponent in the Trotter product of exponentials.

Parameters

- **trotter_number** (int, default: 1) – The number of time-slices in the Trotter approximation.
- **trotter_order** (int, default: 1) – The order of the Trotter-Suzuki approximation being used. Currently, this only supports 1, i.e. ABABAB.
- **constant** (Union[float, complex], default: 1.0) – An additional constant factor to multiply each exponent by, which may be useful when, for example, constructing a time evolution operator.
- **coefficients_location** (*TrotterizeCoefficientsLocation*, default: TrotterizeCoefficientsLocation.OUTER) – By default, the coefficient of each term in the input operator is multiplied by the Trotter factor and stored in the outer coefficient of the generated OperatorList, with the coefficient of each inner operator set to 1. This behaviour can be controlled with this argument - set to "outer" for default behaviour. Setting this parameter to "inner" will store all scalars in the inner operator coefficient, with the outer coefficients of the generated OperatorList set to 1. Setting this parameter to "mixed" will store the Trotter factor in the outer coefficients of the generated OperatorList, with the inner coefficients of each term left untouched. See examples for a comparison.

Returns

TypeVar(OperatorListT, bound= OperatorList) – A Trotterized form of the exponential product, where each element is an individual exponent.

Examples

```
>>> from inquanto.operators import QubitOperator
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6) + QubitOperator("Z1 Z2 Z3 Z5", -5.6j)
>>> trotter_operator = op1.trotterize(trotter_number=2)
>>> print(trotter_operator)
2.3       [(1.0, X0 Y1 Z3)],
-2.8j     [(1.0, Z1 Z2 Z3 Z5)],
2.3       [(1.0, X0 Y1 Z3)],
-2.8j     [(1.0, Z1 Z2 Z3 Z5)]
>>> from inquanto.operators import QubitOperator
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6) + QubitOperator("Z1 Z2 Z3 Z5", -5.6j)
>>> trotter_operator = op1.trotterize(trotter_number=2,coefficients_location=
↪"inner")
>>> print(trotter_operator)
1.0       [(2.3, X0 Y1 Z3)],
1.0       [(-2.8j, Z1 Z2 Z3 Z5)],
1.0       [(2.3, X0 Y1 Z3)],
1.0       [(-2.8j, Z1 Z2 Z3 Z5)]
>>> from inquanto.operators import QubitOperator
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6) + QubitOperator("Z1 Z2 Z3 Z5", -5.6j)
>>> trotter_operator = op1.trotterize(trotter_number=2,coefficients_location=
↪"mixed")
>>> print(trotter_operator)
0.5       [(4.6, X0 Y1 Z3)],
0.5       [(-5.6j, Z1 Z2 Z3 Z5)],
0.5       [(4.6, X0 Y1 Z3)],
0.5       [(-5.6j, Z1 Z2 Z3 Z5)]
```

unsympify()

Unsympifies dicionary values.

Replaces symbolic expressions with their corresponding numeric values.

Returns

LinearDictCombiner – Updated instance of LinearDictCombiner.

Raises

TypeError – When unsympification fails.

classmethod zero()

Return object with a zero dict entry.

Examples

```
>>> print(LinearDictCombiner.zero())
(0)
```

class SymmetryOperatorPauliFactorised(*data=None*, *coeff=1.0*)

Bases: *QubitOperatorList*, SymmetryOperator

Stores a factorised form of qubit symmetry operators.

Our symmetry operators may be an exponentiated sum of Pauli operators. Expanding this out as a single *SymmetryOperatorPauli* may blow up exponentially. However, we know that many properties can be calculated without this cost - particularly when sets of terms are known to map to single Pauli strings. Here, we store the

symmetry operator in factorised form - the overall symmetry operator is the product of self.operators. To make calculation easier, we also enforce that the component operators must all mutually commute.

Parameters

- **data** (Union[*QubitOperator*, *QubitOperatorString*, List[Tuple[Union[int, float, complex, Expr], *QubitOperator*]], None], default: None) – Input data from which the list of qubit operators is built. See *QubitOperator* for methods of constructing terms.
- **coeff** (Union[int, float, complex, Expr], default: 1.0) – Multiplicative scalar coefficient. Used only if data is not of type list.

class CompressScalarsBehavior(*value*)

Bases: str, Enum

Governs compression of scalars method behaviour.

ALL = 'all'

Combine all coefficients possible, simplifying inner terms.

ONLY_IDENTITIES_AND_ZERO = 'simple'

Only compress based on terms which are a scalar multiple of the identity operator, or zero.

OUTER = 'outer'

Combine all "outer" coefficients (coefficients stored directly in the top-level OperatorList) into one.

class ExpandExponentialProductCoefficientsBehavior(*value*)

Bases: str, Enum

Governs behaviour for expansion of exponential products of commuting operators.

BRING_INTO_OPERATOR = 'compact'

Treat top-level coefficients of the QubitOperatorList as being outside the exponential; multiply generated component QubitOperators by them and return QubitOperatorList with unit top-level coefficients.

IGNORE = 'ignore'

Drop top-level coefficients entirely.

IN_EXPONENT = 'inside'

Treat top-level coefficients of the QubitOperatorList as being within the exponent.

OUTSIDE_EXPONENT = 'outside'

Treat top-level coefficients of the QubitOperatorList as being outside the exponential; return them as top-level coefficients of the generated QubitOperatorList.

class FactoryCoefficientsLocation(*value*)

Bases: str, Enum

Determines where the from_Operator method places coefficients.

INNER = 'inner'

Coefficients are left within the component operators.

OUTER = 'outer'

Coefficients are moved to be directly stored at the top-level of the OperatorList.

property all_qubits: Set[Qubit]

Returns a set of all qubits included in any operator in the QubitOperatorList.

Return type
Set[Qubit]

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

collapse_as_linear_combination(*ignore_outer_coefficients=False*)

Treating the OperatorList as a linear combination, return it in the form of a Operator.

By default, each term is multiplied by its corresponding scalar coefficient, then all such multiplied terms are summed to yield a single Operator. The first step may be skipped (i.e. the scalar coefficients associated with each component Operator may be ignored) by setting `ignore_outer_coefficients` to True.

Parameters
ignore_outer_coefficients (bool, default: False) – Set to True to skip multiplication by the "outer" coefficients in the OperatorList.

Returns
TypeVar(OperatorT, bound= Operator) – The sum of all terms in the OperatorList, multiplied by their associated coefficients if requested.

collapse_as_product(*reverse=False*, *ignore_outer_coefficients=False*)

Treating the OperatorList as a product of separate terms, return the full product as an Operator.

By default, each Operator in the OperatorList is multiplied by its corresponding coefficient, and the product is taken sequentially with the leftmost term given by the first element of the OperatorList. This behaviour can be reversed with the `reverse` parameter - if set to True, the leftmost term will be given by the last element of OperatorList.

If ignore_outer_coefficients is set to True, the first step (the multiplication of Operator terms by their corresponding coefficients) is skipped - i.e. the "outer" coefficients stored in the OperatorList are ignored.

> **Danger:** In the general case, the number of terms in the expansion will blow up exponentially (and thus the runtime of this method will also blow up exponentially).

Parameters
- **reverse** (bool, default: False) – Set to True to reverse the order of the product.
- **ignore_outer_coefficients** (bool, default: False) – Set to True to skip multiplication by the "outer" coefficients in the OperatorList.

Returns
TypeVar(OperatorT, bound= Operator) – The product of each component operator.

compatibility_matrix(*abs_tol=1e-10*)

Returns the compatibility matrix of the operator list.

This matrix is the adjacency matrix of the compatibility graph of the OperatorList. It is a boolean N*N matrix where N is the number of operators in the OperatorList. Element (n,m) is True if operators n and m commute, otherwise False.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold used for determining commutativity - see QubitOperator.commutator() for details.

Returns

ndarray – The compatibility matrix of the operator list.

compress_scalars_as_product (*abs_tol=1e-12*, *inner_coefficient=False*, *coefficients_to_compress=CompressScalarsBehavior.ONLY_IDENTITIES_AND_ZERO*)

Treating the OperatorList as a product, compress identity terms or resolve to zero if possible.

To do this, we iterate through the (coefficient, operator) pairs in the OperatorList. If any coefficient or operator is zero, then we return an empty OperatorList, as the product will be zero. If the iteration operator is an identity, it will be treated as a scalar multiplier, itself multiplied by its associated coefficient, with the operator itself removed from the OperatorList. These multipliers are multiplied together and – if they do not equate to 1 – are prepended to the OperatorList as a separate identity term.

By default, (coefficient,operator) pairs which are not identity or zero will be ignored. This behaviour may be controlled with the coefficients_to_compress parameter. This can be set to "outer" to include all "outer" coefficients (of the (coefficient,operator) pairs) in the prepended identity term. It can also be set to "all" to additionally bring coefficients within the non-identity operators into the prepended identity term – i.e. for operators where all the "inner" coefficients are equal. See examples for a comparison.

The prepended identity term will, by default, store the multiplied scalar factor in the "outer" coefficient of the (coefficient, operator) pair. The inner_coefficient parameter may be set to True to instead store it within the operator.

Note: Term simplification is not performed on component operators. It is possible that a component operator will resolve on simplification to the identity or zero. This method will not catch these occurrences.

Parameters

- **abs_tol** (float, default: 1e-12) – Numerical threshold for comparison of numbers to 0 and 1. Set to None to use exact identity.
- **inner_coefficient** (bool, default: False) – Set to True to store generated scalar factors within the identity term, as described above.
- **coefficients_to_compress** (*CompressScalarsBehavior*, default: CompressScalarsBehavior.ONLY_IDENTITIES_AND_ZERO) – Controls which scalar factors will be combined, as described above.

Returns

TypeVar(OperatorListT, bound= OperatorList) – The OperatorList with identity and zero terms combined, as described above.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0", 2) + QubitOperator("Z1", 2)
>>> op2 = 3 * QubitOperator.identity()
>>> qol = QubitOperatorList([(5, op1), (7, op2)])
>>> result = qol.compress_scalars_as_product()
>>> print(result)
21.0       [(1.0, )],
5          [(2, X0), (2.0, Z1)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0", 2) + QubitOperator("Z1", 2)
>>> op2 = 3 * QubitOperator.identity()
```

(continues on next page)

(continued from previous page)

```
>>> qol = QubitOperatorList([(5, op1), (7, op2)])
>>> result = qol.compress_scalars_as_product(inner_coefficient=True)
>>> print(result)
1         [(21.0, )],
5         [(2, X0), (2.0, Z1)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0", 2) + QubitOperator("Z1", 2)
>>> op2 = 3 * QubitOperator.identity()
>>> qol = QubitOperatorList([(5, op1), (7, op2)])
>>> result = qol.compress_scalars_as_product(coefficients_to_compress="outer")
>>> print(result)
105.0     [(1.0, )],
1.0       [(2, X0), (2.0, Z1)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0", 2) + QubitOperator("Z1", 2)
>>> op2 = 3 * QubitOperator.identity()
>>> qol = QubitOperatorList([(5, op1), (7, op2)])
>>> result = qol.compress_scalars_as_product(coefficients_to_compress="all")
>>> print(result)
210.0     [(1.0, )],
1.0       [(1.0, X0), (1.0, Z1)]
```

compute_jacobian (*symbols*, *as_sympy_sparse=False*)

Generate symbolic sparse Jacobian Matrix.

If the number of terms in the operator list is N and the number of symbols in the symbols list is M then the resulted matrix has a shape (N, M) and $J_{i,j} = \frac{\partial c_i}{\partial \theta_j}$

Parameters

- **symbols** (List[Symbol]) – List of symbols w.r.t. which we differentiate each coefficients.
- **as_sympy_sparse** (bool, default: False) – If this is true then it converts the output to an ImmutableSparseMatrix.

Returns

Union[List, ImmutableSparseMatrix] – Jacobian Matrix in list of tuples of (i,j, J_ij) (this format is referred as a row-sorted list of non-zero elements of the matrix.)

Examples

```
>>> op0 = QubitOperator("X0 Y2 Z3", 4.6)
>>> op1 = QubitOperator(((0, "X"), (1, "Y"), (3, "Z")), 0.1 + 2.0j)
>>> op2 = QubitOperator([(0, "X"), (1, "Z"), (3, "Z")], -1.3)
>>> qs = QubitOperatorString.from_string("X0 Y1 Y3")
>>> op3 = QubitOperator(qs, 0.8)
>>> qs0 = QubitOperatorString.from_string("X0 Y1 Z3 X4")
>>> qs1 = QubitOperatorString.from_tuple([(0, Pauli.Y), (1, Pauli.X)])
>>> dictionary = {qs0: 2.1, qs1: -1.7j}
>>> op4 = QubitOperator(dictionary)
```

```
>>> a, b, c = sympify("a,b,c")
>>> trotter_operator = QubitOperatorList([(a**3, op1), (2, op2), (b, op3), (c,
↪ op4)])
```

(continues on next page)

(continued from previous page)

```
>>> print(trotter_operator)
a**3        [(0.1+2j, X0 Y1 Z3)],
2           [(-1.3, X0 Z1 Z3)],
b           [(0.8, X0 Y1 Y3)],
c           [(2.1, X0 Y1 Z3 X4), (-1.7j, Y0 X1)]
```

```
>>> jacobian_matrix = trotter_operator.compute_jacobian([a, b])
>>> print(jacobian_matrix)
[(0, 0, 3*a**2), (2, 1, 1)]
```

```
>>> jacobian_matrix = trotter_operator.compute_jacobian([a, b], as_sympy_
↪sparse=True)
>>> print(jacobian_matrix)
Matrix([[3*a**2, 0], [0, 0], [0, 1], [0, 0]])
```

---

copy()

Returns a deep copy of self.

Return type
LinearListCombiner

df()

Returns a Pandas DataFrame object of the dictionary.

dot_state_as_linear_combination(*state*, *qubits=None*)

Operate upon a state acting as a linear combination of the component operators.

Associated scalar constants are treated as scalar multiplier of their respective QubitOperator terms. This method can accept right-hand state as a `QubitState`, `QubitStateString` or a `numpy.ndarray`. In the former two cases, we maintain a symbolic representation of both operator and state, and each Pauli is implemented in sequence. Resultant states are returned as `QubitState`. In this case, the optional qubits parameter is ignored.

For a `numpy.ndarray`, we hand off to pytket's `QubitPauliOperator.dot_state()` method - this should be faster for dense states, but slower for sparse ones. Here, a register of qubits may be specified in the `qubits` parameter to determine the meaning of the indices of the provided state vector. From the pytket documentation:

- When `qubits` is an explicit list, the qubits are ordered with `qubits[0]` as the most significant qubit for indexing into state.
- If `None`, qubits sequentially indexed from 0 in the default register and ordered by ILO-BE so `Qubit(0)` is the most significant.

Parameters

- **state** (`Union[QubitState, QubitStateString, ndarray]`) – Input qubit state to operated on.
- **qubits** (`Optional[List[Qubit]]`, default: `None`) – For ndarray input, determines sequencing of qubits in the state, if not mapped to the default register. Ignored for other input types.

Returns
`Union[QubitState, ndarray]` – Output state.

dot_state_as_product(*state*, *reverse=False*, *qubits=None*)

Operate upon a state with each component operator in sequence, starting from operator indexed 0.

This will apply each operator in sequence, starting from the top of the list - i.e. the top of the list is on the right hand side when acting on a ket. Ordering can be controlled with the `reverse` parameter. Associated scalar constants are treated as scalar multipliers.

The right-hand state ket can be accepted as a QubitState,QubitStateString or a ndarray. In the former two cases, we maintain a symbolic representation of both operator and state, and each Pauli is implemented in sequence. Resultant states are returned as QubitState. In this case, the `qubits` parameter is ignored.

For ndarrays, we hand off to pytket's QubitPauliOperatorString.dot_state() method - this should be faster for dense states, but slower for sparse ones. Here, a register of qubits may be specified in the `qubits` parameter to determine the meaning of the indices of the provided state vector. From the pytket documentation:

- When `qubits` is an explicit list, the qubits are ordered with `qubits[0]` as the most significant qubit for indexing into state.
- If `None`, qubits sequentially indexed from 0 in the default register and ordered by ILO-BE so `Qubit(0)` is the most significant.

> **Danger:** In the general case, this will blow up exponentially.

Parameters

- **state** (Union[*QubitState*, *QubitStateString*, ndarray]) – The state to be acted upon.
- **reverse** (bool, default: False) – Set to True to reverse order of operations applied (i.e. treat the 0th indexed operator as the leftmost operator).
- **qubits** (Optional[List[Qubit]], default: None) –

Returns

Union[*QubitState*, ndarray] – The state yielded from acting on input state in the type as described above.

empty()

Checks if internal list is empty.

Return type

bool

equality_matrix(*abs_tol=1e-10*)

Returns the equality matrix of the operator list.

The equality matrix is a boolean N*N matrix where N is the number of operators in the OperatorList. Element (n,m) is True if operators n and m are equal, otherwise False.

Parameters

abs_tol (Optional[float], default: 1e-10) – Threshold of comparing numeric values. Set to None to test for exact equivalence.

Returns

ndarray – The equality matrix of the QubitOperatorList.

evalf(**args*, ***kwargs*)

Numerically evaluates symbolic expressions stored in the left and right values of list items and replaces them with the results.

Parameters

- **args** (Any) – Args to be passed to *sympy.evalf()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.evalf()*.

Returns

LinearListCombiner – Updated instance of LinearListCombiner.

expand_exponential_product_commuting_operators(*expansion_coefficients_behavior=ExpandExponentialProduct*
additional_exponent=1.0,
check_commuting=True,
combine_scalars=True)

Trigonometrically expand a QubitOperatorList which represents a product of exponentials of operators with mutually commuting terms.

Given a QubitOperatorList, this method interprets it as a product of exponentials of the component operators. This means that for a QubitOperatorList consisting of terms (c_i,:math:*hat{O}_i*), it is interpreted as $\prod_i e^{c_i \hat{O}_i}$. Component operators must consist of terms which all mutually commute.

Each component operator is first expanded as its own exponential product (as they consist of terms which all mutually commute, $e^{A+B} = e^A e^B$). Each of the individual exponentiated terms are trigonometrically expanded. These are then concatenated to return a QubitOperatorList with the desired form.

Notes

Typically this may be used upon an operator generated by Trotterization methods to yield a computable form of the exponential product. While this method scales polynomially, expansion of the resulting operator or calculating its action on a state will typically be exponentially costly.

By default, this method will combine constant scalar multiplicative factors (obtained by, for instance, pi rotations) into one identity term prepended to the generated QubitOperatorList. This behaviour can be disabled by setting combine_scalars to False. For more detail, and for control over the process of combining constant factors, see the .compress_scalars_as_product method.

Parameters

- **expansion_coefficients_behavior** (*ExpandExponentialProductCoefficientsBehavior*, default: ExpandExponentialProductCoefficientsBehavior.IN_EXPONENT) – By default, it will be assumed that the 'outer' coefficients associated with each component operator are within the exponent, as per the above formula. This may be set to "outside" to instead assume that they are scalar multiples of the exponential itself, rather than the exponent (i.e. $\prod_i c_i e^{\hat{O}_i}$). In this case, the coefficients will be returned as the outer coefficients of the returned QubitOperatorList, as input. Set to "compact" to assume similar structure to "outside", but returning coefficients within the component QubitOperators. Set to "ignore" to drop the 'outer' coefficients entirely. See examples for a comparison.
- **additional_exponent** (complex, default: 1.0) – Optional additional factor in each exponent.
- **check_commuting** – Set to False to skip checking whether all terms in each component operator commute.
- **combine_scalars** – Set to False to disable combination of identity and zero terms.

Returns

QubitOperatorList – The trigonometrically expanded form of the input QubitOperatorList.

Examples

```
>>> import sympy
>>> op= QubitOperator("X0", -1.j * sympy.Symbol('x'))
>>> qol = QubitOperatorList([(2, op)])
>>> result = qol.expand_exponential_product_commuting_operators()
>>> print(result)
1         [(1.0*cos(2.0*x), ), (-1.0*I*sin(2.0*x), X0)]
>>> import sympy
>>> op = QubitOperator("X0", -1.j * sympy.Symbol('x'))
>>> qol = QubitOperatorList([(2, op)])
>>> result = qol.expand_exponential_product_commuting_operators(expansion_
↪coefficients_behavior="outside")
>>> print(result)
2         [(1.0*cos(1.0*x), ), (-1.0*I*sin(1.0*x), X0)]
>>> import sympy
>>> op = QubitOperator("X0", -1.j * sympy.Symbol('x'))
>>> qol = QubitOperatorList([(2, op)])
>>> result = qol.expand_exponential_product_commuting_operators(expansion_
↪coefficients_behavior="ignore")
>>> print(result)
1         [(1.0*cos(1.0*x), ), (-1.0*I*sin(1.0*x), X0)]
>>> import sympy
>>> op = QubitOperator("X0", -1.j * sympy.Symbol('x'))
>>> qol = QubitOperatorList([(2, op)])
>>> result = qol.expand_exponential_product_commuting_operators(expansion_
↪coefficients_behavior="compact")
>>> print(result)
1.0       [(2.0*cos(1.0*x), ), (-2.0*I*sin(1.0*x), X0)]
```

free_symbols()

Returns the free symbols in the coefficient values.

Return type

set

free_symbols_ordered()

Returns the free symbols in the list, ordered alphabetically.

Returns

SymbolSet – Ordered set of symbols.

classmethod from_Operator(*input*, *additional_coefficient=1.0*, *coefficients_location=FactoryCoefficientsLocation.INNER*)

Converts an Operator to an OperatorList with terms in arbitrary order.

Each term in the Operator is split into a separate component Operator in the OperatorList. The resulting location of each scalar coefficient in the input Operator can be controlled with the coefficients_location parameter. Setting this to 'inner' will leave coefficients stored as part of the component operators, 'outer' will move the coefficients to the 'outer' level.

Parameters

- **input** (Operator) – The input operator to split into an OperatorList.
- **additional_coefficient** (Union[int, float, complex, Expr], default: 1.0) – An additional factor to include in the "outer" coefficients of the generated OperatorList.
- **coefficients_location** (*FactoryCoefficientsLocation*, default: FactoryCoefficientsLocation.INNER) – The destination of the coefficients of the input operator, as described above.

Returns

TypeVar(OperatorListT, bound= OperatorList) – An OperatorList as described above.

Raises

ValueError – On invalid input to the coefficients_location parameter.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op = QubitOperator("X0", 2.) + QubitOperator("Z1", 2.)
>>> qol = QubitOperatorList.from_Operator(op)
>>> print(qol)
1.0       [(2.0, X0)],
1.0       [(2.0, Z1)]
>>> op = QubitOperator("X0", 2.) + QubitOperator("Z1", 2.)
>>> qol = QubitOperatorList.from_Operator(op,coefficients_location='outer')
>>> print(qol)
2.0       [(1.0, X0)],
2.0       [(1.0, Z1)]
```

classmethod from_list (*ops*, *symbol_format='term{}'*)

Converts a list of QubitOperators to a QubitOperatorList.

Each QubitOperator in the list will be a separate entry in the generated QubitOperatorList. Fresh symbols will be generated to represent the "outer" coefficients of the generated QubitOperatorList.

Parameters

- **ops** (List[*QubitOperator*]) – A list of QubitOperators which will comprise the terms of the generated QubitOperatorList.
- **symbol_format** – A raw string containing one positional substitution (in which a numerical index will be placed), used to generate each symbolic coefficient.

Returns

QubitOperatorList – A QubitOperatorList with terms corresponding to the input QubitOperators, and coefficients as newly generated symbols.

classmethod from_string (*input_string*)

Constructs a child class instance from a string.

Parameters

input_string (str) – String in the format `coeff1 [(coeff1_1, term1_1), ..., (coeff1_n, term1_n)], ..., coeffn [(coeffn_1, termn_1), ...]`.

Returns

Child class object.

incompatibility_matrix(*abs_tol=1e-10*)

Returns the incompatibility matrix of the operator list.

This matrix is the adjacency matrix of the incompatibility graph of the OperatorList. It is a boolean N*N matrix where N is the number of operators in the OperatorList. Element (n,m) is False if operators n and m commute, otherwise True.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold used for determining commutativity - see QubitOperator.commutator() for details.

Returns
ndarray – The incompatibility matrix of the operator list.

is_empty()

Return True if operator is 0, else False.

Return type
bool

is_symmetry_of(*operator*)

Check if operator is symmetry of given operator.

Checks by determining if operator commutes with all terms of other operator. True if it commutes, False otherwise.

Parameters
operator (*QubitOperator*) – Operator to compare to.

Returns
bool – True if this is a symmetry of operator, otherwise False.

Danger: This calls *to_symmetry_operator_pauli()*. For standard chemical purposes using Z2 symmetries this should be ok, but may scale exponentially when in advanced usage.

items()

Returns internal list.

Return type
List[Tuple[Any, Union[int, float, complex, Expr]]]

make_hashable()

Return a hashable representation of the object.

Currently simply returns a stringified object.

Return type
str

map(*mapping*)

Updates list items right values in-place, using a mapping function provided.

Parameters
mapping (Callable[[Union[int, float, complex, Expr]], Union[int, float, complex, Expr]]) – A callable object which takes each original value and returns the corresponding new value.

Return type
LinearListCombiner

property n_symbols: int

Returns the number of free symbols in the object.

Return type
int

operator_class

alias of *QubitOperator*

parallelity_matrix(*abs_tol=1e-10*)

Returns the "parallelity matrix" of the operator list.

This matrix is a boolean N*N matrix where N is the number of operators in the OperatorList. Element (n,m) is True if operators n and m are parallel - i.e. they are a scalar multiple of each other. Otherwise, the element in the returned matrix is False.

Parameters
abs_tol (Optional[float], default: 1e-10) – Threshold of comparing numeric values. Set to None to test for exact equivalence.

Returns
ndarray – The "parallelity matrix" of the operator list.

print_table()

Print internal list formatted as a table.

Return type
None

qubitwise_compatibility_matrix()

Returns the qubit-wise compatibility matrix of the operator list.

This matrix is the adjacency matrix of the qubit-wise compatibility graph of the OperatorList. It is a boolean N*N matrix where N is the number of operators in the OperatorList. Element (n,m) is True if operators n and m qubit-wise commute, otherwise False. See QubitOperator.qubitwise_commutes_with() for further details on qubit-wise commutativity.

Note that individual QubitOperators within the OperatorList must be comprised of single terms for this to be well-defined.

Returns
ndarray – The qubit-wise compatibility matrix of the operator list.

qubitwise_incompatibility_matrix()

Returns the qubit-wise incompatibility matrix of the operator list.

This matrix is the adjacency matrix of the qubit-wise incompatibility graph of the OperatorList. It is a boolean N*N matrix where N is the number of operators in the OperatorList. Element (n,m) is False if operators n and m qubit-wise commute, otherwise True. See QubitOperator.qubitwise_commutes_with() for further details on qubit-wise commutativity.

Note that individual QubitOperators within the OperatorList must be comprised of single terms for this to be well-defined.

Returns: The qubit-wise incompatibility matrix of the operator list.

Return type
ndarray

reduce_exponents_by_commutation (*abs_tol=1e-10*)

Given a QubitOperatorList representing a product of exponentials, combine terms by commutation.

Given a QubitOperatorList, this method interprets it as a product of exponentials of the component operators. This means that for a QubitOperatorList consisting of terms (c_i,:math:*hat{O}_i*), it is interpreted as $\prod_i e^{c_i \hat{O}_i}$. This method attempts to combine terms which are scalable multiples of one another. Each term is commuted backwards through the QubitOperatorList until it reaches an operator that it does not commute with. If it encounters an operator which is a scalable multiple of the term, then the terms are combined. Otherwise, the term is left unchanged.

Parameters

abs_tol (`float`, default: `1e-10`) – Tolerance threshold used for determining commutativity - see QubitOperator.commutator() for details.

Returns

QubitOperatorList – A reduced form of the QubitOperatorList as described above.

retrotterize (*new_trotter_number*, *initial_trotter_number=1*, *new_trotter_order=1*, *initial_trotter_order=1*, *constant=1.0*, *inner_coefficients=False*)

Retrotterize an expression given a OperatorList representing a product of exponentials.

This method assumes that the OperatorList represents a product of exponentials, with each Operator in the list corresponding to an exponent of a single exponential in the product. Scalar factors within the OperatorList are treated as scalar multipliers within each exponent.

The OperatorList is first untrotterized using the provided `initial_trotter_number` and `initial_trotter_order`, then subsequently Trotterized using the provided `new_trotter_number` and `new_trotter_order`. The returned OperatorList corresponds to the generated product of exponentials, in a similar manner to the original OperatorList.

Parameters

- **new_trotter_number** (`int`) – The desired number of Trotter steps in the final Trotter-Suzuki expansion.
- **initial_trotter_number** (`int`, default: `1`) – The number of Trotter steps in the original Trotter-Suzuki expansion.
- **new_trotter_order** (`int`, default: `1`) – The order of the original Trotter-Suzuki expansion used. Currently, only a first order (ABABAB…) expansion is supported.
- **initial_trotter_order** (`int`, default: `1`) – The desired order of the final Trotter-Suzuki expansion. Currently, only a first order (ABABAB…) expansion is supported.
- **constant** (`Union[float, complex]`, default: `1.0`) – An additional constant multiplier in the exponent.
- **inner_coefficients** (`bool`, default: `False`) – By default, generated scalar factors in each exponent are stored in the coefficients of the generated OperatorList, with the coefficient of each inner Operator unchanged. Set this to True to instead store all scalar factors as coefficients in each Operator, with the outer coefficients of the OperatorList left unchanged. See examples for a comparison.

Returns

`TypeVar(OperatorListT, bound= OperatorList)` – The exponential product retrotterized with the provided new Trotter number and order. Each element is an individual exponent.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> retrotterised = qol.retrotterize(new_trotter_number=4,initial_trotter_
↪number=2)
>>> print(retrotterised)
0.25       [(1.0, X0 X1)],
0.25       [(1.0, Z0)],
0.25       [(1.0, X0 X1)],
0.25       [(1.0, Z0)],
0.25       [(1.0, X0 X1)],
0.25       [(1.0, Z0)],
0.25       [(1.0, X0 X1)],
0.25       [(1.0, Z0)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> retrotterised = qol.retrotterize(new_trotter_number=4,initial_trotter_
↪number=2,inner_coefficients=True)
>>> print(retrotterised)
0.5        [(0.5, X0 X1)],
0.5        [(0.5, Z0)],
0.5        [(0.5, X0 X1)],
0.5        [(0.5, Z0)],
0.5        [(0.5, X0 X1)],
0.5        [(0.5, Z0)],
0.5        [(0.5, X0 X1)],
0.5        [(0.5, Z0)]
```

reversed_order()

Reverses internal list order and returns it as a new object.

Return type
LinearListCombiner

simplify(**args*, ***kwargs*)

Simplifies expressions stored in left and right values of list items.

Parameters

- **args** (Any) – Args to be passed to *sympy.simplify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.simplify()*.

Returns
LinearListCombiner – Updated instance of LinearListCombiner.

split()

Generates pair objects from list items.

Return type
Iterator[LinearListCombiner]

split_totally_commuting_set(*abs_tol=1e-10*)

For a QubitOperatorList, separate it into a totally commuting part and a remainder.

This will return two QubitOperatorList - the first comprised of all the component QubitOperators which commute with all other terms, the second comprised of all other terms. An empty QubitOperatorList will be returned if either of these sets is empty.

Parameters

abs_tol (float, default: 1e-10) – Tolerance threshold used for determining commutativity - see QubitOperator.commutator() for details.

Returns

Two QubitOperatorLists - the first representing the totally commuting set, the second representing the rest of the operator.

sublist (*sublist_indices*)

Returns a new instance containing a subset of the terms of the original object.

Parameters

sublist_indices (list[int]) – List of indices of the operator list elements to constitute a new object.

Returns

TypeVar(OperatorListT, bound= OperatorList) – A new instance of OperatorList, being a sublist of the original operator.

Raises

ValueError – If sublist_indices contains indices not contained in self, or self is empty.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 Y1 Z3", 4.6)
>>> op2 = QubitOperator("Z0", -1.6j)
>>> op3 = QubitOperator("Z1 Z2 Z3 Z5", -5.6j)
>>> long_operator = QubitOperatorList([(1, op1), (1, op2), (1, op3)])
>>> short_operator = long_operator.sublist([0, 2])
>>> print(short_operator)
1          [(4.6, X0 Y1 Z3)],
1          [(-5.6j, Z1 Z2 Z3 Z5)]
```

---

subs (*symbol_map*)

Returns a new objects with symbols substituted.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type

TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution (*symbol_map=None*)

Substitutes free symbols for numerical values according to a map.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) –

Return type
LinearListCombiner

symmetry_sector(*state*)

Find the symmetry sector that a qubit state is in by direct expectation value calculation.

As all terms commute, we take the product of their individual expectation values. For certain symmetry operators (e.g. parity operators) this should be polynomially hard.

Parameters
state (*QubitState*) – Input qubit state.

Returns
int – The symmetry sector of the state (i.e. the expectation value).

Danger: Due to direct expectation value calculation, this may scale exponentially in the general case, although it should be okay for sparse states and operators e.g. Z2 symmetries on number states.

sympify(**args*, ***kwargs*)

Sympifies left and right values of list items.

Replaces left and right values with their corresponding symbolic expressions.

Parameters

- **args** (Any) – Args to be passed to *sympy.sympify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.sympify()*.

Returns
LinearListCombiner – Updated instance of LinearListCombiner.

Raises
RuntimeError – when sympification fails.

to_sparse_matrices(*qubits=None*)

Returns a list of sparse matrices representing each element of the QubitOperatorList.

Outer coefficients are treated by multiplying their corresponding QubitOperators. Otherwise, this method acts largely as a wrapper for QubitOperator.to_sparse_matrix(), which derives from the base pytket QubitPauliOperator.to_sparse_matrix() method. The qubits parameter specifies the ordering scheme for qubits. Note that if no explicit qubits are provided, we use the set of all qubits included in any operator in the QubitOperatorList (i.e. `self.all_qubits`), ordered ILO-BE as per pytket. From the pytket docs:

- When `qubits` is an explicit list, the qubits are ordered with `qubits[0]` as the most significant qubit for indexing into the matrix.
- If `None`, then no padding qubits are introduced and we use the ILO-BE convention, e.g. `Qubit("a",0)` is more significant than `Qubit("a",1)` or `Qubit("b")`.
- Giving a number specifies the number of qubits to use in the final operator, treated as sequentially indexed from 0 in the default register (padding with identities as necessary) and ordered by ILO-BE so `Qubit(0)` is the most significant.

Parameters
qubits (Union[List[Qubit], int, None], default: None) – Sequencing of qubits in the matrix, either as an explicit list, number of qubits to pad to, or infer from the operator list.

Returns
csc_matrix – A sparse matrix representation of the operator.

to_symmetry_operator_pauli()

Convert to a *SymmetryOperatorPauli*.

> **Warning:** For standard chemical purposes using Z2 symmetries this should be ok, but may scale exponentially when in advanced usage.

Return type

SymmetryOperatorPauli

trotterize_as_linear_combination(*trotter_number*, *trotter_order=1*, *constant=1.0*, *inner_coefficients=False*)

Trotterize an exponent linear combination of Operators.

This method assumes that the OperatorList represents the exponential of a linear combination of Operators, with each Operator within the OperatorList corresponding to a term in this linear combination. Trotterization is performed at the level of these Operators. The Operators contained within the returned OperatorList correspond to exponents within the Trotter sequence.

Parameters

- **trotter_number** (int) – The number of Trotter steps in the Trotter-Suzuki expansion.
- **trotter_order** (int, default: 1) – The order of the Trotter-Suzuki approximation to be used. Presently, only first-order (ABABAB…) is supported.
- **constant** (Union[float, complex], default: 1.0) – An additional constant multiplier in the exponent.
- **inner_coefficients** (bool, default: False) – By default, generated scalar factors in each exponent are stored in the coefficients of the generated OperatorList, with the coefficient of each inner Operator unchanged. Set this to True to instead store all scalar factors as coefficients in each Operator, with the outer coefficients of the OperatorList left unchanged. See examples for a comparison.

Returns

TypeVar(OperatorListT, bound= OperatorList) – A Trotterized form of the exponential product, where each element is an individual exponent.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1.,op1),(1.,op2)])
>>> result = qol.trotterize_as_linear_combination(2)
>>> print(result)
0.5       [(1.0, X0 X1)],
0.5       [(1.0, Z0)],
0.5       [(1.0, X0 X1)],
0.5       [(1.0, Z0)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1.,op1),(1.,op2)])
>>> result = qol.trotterize_as_linear_combination(2,inner_coefficients=True)
>>> print(result)
```

(continues on next page)

(continued from previous page)

```
1.0        [(0.5, X0 X1)],
1.0        [(0.5, Z0)],
1.0        [(0.5, X0 X1)],
1.0        [(0.5, Z0)]
```

unsympify()

Unsympifies dicionary values.

Replaces symbolic expressions with their corresponding numeric values.

Returns

LinearListCombiner – Updated instance of LinearListCombiner.

Raises

TypeError – When unsympification fails.

untrotterize(*trotter_number*, *trotter_order=1*)

Reverse a Trotter-Suzuki expansion given an OperatorList representing a product of exponentials.

This method assumes that the OperatorList represents a product of exponentials, with each Operator in the list corresponding to an exponent of a single exponential in the product. Scalar factors within the OperatorList are treated as scalar multipliers within each exponent. An Operator corresponding to the exponent of a single, untrotterized exponential is returned.

Parameters

- **trotter_number** (int) – The number of Trotter steps within the Trotter expansion to be reversed.
- **trotter_order** (int, default: 1) – The order of the Trotter-Suzuki expansion used. Currently, only a first order (ABABAB…) expansion is supported.

Returns

TypeVar(OperatorT, bound= Operator) – The exponent of the untrotterised operator.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> untrotterised = qol.untrotterize(2)
>>> print(untrotterised)
(1.0, X0 X1), (1.0, Z0)
```

untrotterize_partitioned(*trotter_number*, *trotter_order=1*, *inner_coefficients=False*)

Reverse a Trotter-Suzuki expansion given a OperatorList representing a product of exponentials, maintaining separation of exponents.

This method assumes that the OperatorList represents a product of exponentials, with each Operator in the list corresponding to an exponent of a single exponential in the product. Scalar factors within the OperatorList are treated as scalar multipliers within each exponent. A OperatorList is returned wherein each term represents a single term in the exponent of the single, untrotterized exponential.

Parameters

- **trotter_number** (int) – The number of Trotter steps within the Trotter expansion to be reversed.
- **trotter_order** (int, default: 1) – The order of the Trotter-Suzuki expansion used. Currently, only a first order (ABABAB…) expansion is supported.
- **inner_coefficients** (bool, default: False) – By default, generated scalar factors in each exponent are stored in the coefficients of the generated OperatorList, with the coefficient of each inner Operator unchanged. Set this to True to instead store all scalar factors as coefficients in each Operator, with the outer coefficients of the OperatorList left unchanged. See examples for a comparison.

Returns

TypeVar(OperatorListT, bound= OperatorList) – The terms in the exponent of the untrotterised operator as a OperatorList.

Raises

ValueError – If the provided Trotter number is not compatible with the OperatorList.

Examples

```
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> untrotterised = qol.untrotterize_partitioned(2)
>>> print(untrotterised)
1.0       [(1.0, X0 X1)],
1.0       [(1.0, Z0)]
>>> from inquanto.operators import QubitOperator,QubitOperatorList
>>> op1 = QubitOperator("X0 X1", 1.)
>>> op2 = QubitOperator("Z0",1.)
>>> qol = QubitOperatorList([(1./2.,op1),(1./2.,op2),(1./2.,op1),(1./2.,op2)])
>>> untrotterised = qol.untrotterize_partitioned(2,inner_coefficients=True)
>>> print(untrotterised)
0.5       [(2.0, X0 X1)],
0.5       [(2.0, Z0)]
```

class UnrestrictedOneBodyRDM(*rdm1_aa*, *rdm1_bb*)

Bases: OneBodyRDM

One-body reduced density matrix in a spin unrestricted representation.

Parameters

- **rdm1_aa** (ndarray) – Reduced one-body density matrix for the alpha spin channel. $1\text{RDM}^{\alpha}_{ij} = \langle c^{\dagger}_{i,\alpha} c_{j,\alpha} \rangle$
- **rdm1_bb** (ndarray) – Reduced one-body density matrix for the beta spin channel. $1\text{RDM}^{\beta}_{ij} = \langle c^{\dagger}_{i,\beta} c_{j,\beta} \rangle$

copy()

Performs a deep copy of object.

Return type

RDM

get_block(*mask*)

Return a new RDM spanning a subset of the original's orbitals.

All orbitals not specified in *mask* are ignored.

Parameters
mask (ndarray) – List of indices of orbitals to retain.

Returns
UnrestrictedOneBodyRDM – New, smaller RDM with only the target orbitals.

classmethod load_h5(*name*)

Loads RDM object from .h5 file.

Parameters
name (Union[str, Group]) – Name of .h5 file to be loaded.

Returns
RDM – Loaded RDM object.

mean_field_rdm2()

Calculate the mean-field two-body RDM object.

Returns
UnrestrictedTwoBodyRDM – Two body RDM object.

n_orb()

Returns number of spatial orbitals.

Return type
int

n_spin_orb()

Returns number of spin-orbitals.

Return type
int

rotate(*rotation_aa*, *rotation_bb=None*)

Rotate the density matrix to a new basis.

Parameters

- **rotation_aa** (ndarray) – Alpha rotation matrix as 2D array.
- **rotation_bb** (Optional[ndarray], default: None) – Beta rotation matrix as 2D array, if unspecified is set equal to *rotation_aa*.

Returns
UnrestrictedOneBodyRDM – RDM after rotation.

save_h5(*name*)

Dumps RDM object to .h5 file.

Parameters
name (Union[str, Group]) – Destination filename of .h5 file.

Return type
None

set_block(*mask*, *rdm*)

Set the RDM entries for a specified set of orbitals.

Parameters

- **mask** (ndarray) – List of indices of orbitals to be edited.
- **rdm** (*UnrestrictedOneBodyRDM*) – RDM object to replace target orbitals.

Returns

UnrestrictedOneBodyRDM – Updated RDM object with target orbitals overwritten.

trace()

Return the trace of the 1-RDM.

Return type

float

class UnrestrictedTwoBodyRDM(*rdm2_aaaa*, *rdm2_bbbb*, *rdm2_aabb*, *rdm2_bbaa*)

Bases: RDM

Two-body reduced density matrix in a spin unrestricted representation.

Parameters

- **rdm2_aaaa** (ndarray) – Reduced two-body density matrix (2-RDM) for spatial orbitals in the aaaa (where a=alpha) spin channels as a 4D array. $\text{2RDM}^{\alpha\alpha}_{ijkl} = \langle c^\dagger_{i,\alpha} c^\dagger_{k,\alpha} c_{l,\alpha} c_{j,\alpha} \rangle$
- **rdm2_bbbb** (ndarray) – 2-RDM for spatial orbitals in the bbbb spin channels as a 4D array. $\text{2RDM}^{\beta\beta}_{ijkl} = \langle c^\dagger_{i,\beta} c^\dagger_{k,\beta} c_{l,\beta} c_{j,\beta} \rangle$
- **rdm2_aabb** (ndarray) – 2-RDM for spatial orbitals in the aabb spin channels as a 4D array. $\text{2RDM}^{\alpha\beta}_{ijkl} = \langle c^\dagger_{i,\alpha} c^\dagger_{k,\beta} c_{l,\beta} c_{j,\alpha} \rangle$
- **rdm2_bbaa** (ndarray) – 2-RDM for spatial orbitals in the bbaa spin channels as a 4D array. $\text{2RDM}^{\beta\alpha}_{ijkl} = \langle c^\dagger_{i,\beta} c^\dagger_{k,\alpha} c_{l,\alpha} c_{j,\beta} \rangle$

copy()

Performs a deep copy of object.

Return type

RDM

classmethod load_h5(*name*)

Loads RDM object from .h5 file.

Parameters

name (Union[str, Group]) – Name of .h5 file to be loaded.

Returns

RDM – Loaded RDM object.

n_orb()

Returns number of spatial orbitals.

Return type

int

n_spin_orb()

Returns number of spin-orbitals.

Return type

int

rotate (*rotation_aa*, *rotation_bb*)

Rotate the density matrix to a new basis.

Parameters

- **rotation_aa** (ndarray) – Alpha rotation matrix as 2D array.
- **rotation_bb** (ndarray) – Beta rotation matrix as 2D array.

Returns

UnrestrictedTwoBodyRDM – RDM after rotation.

save_h5 (*name*)

Dumps RDM object to .h5 file.

Parameters

name (Union[str, Group]) – Destination filename of .h5 file.

Return type

None

22.11 inquanto.protocols

22.11.1 Protocols for Operator Averaging

class ProtocolDirect

Bases: _ProtocolPauliAveraging

Calculate the expectation value of a Hermetian operator by operator averaging the system register.

Implements the 'Operator Averaging' procedure (see: arXiv:1407.7863, arXiv:1510.04279).

Given a Hermitian operator expressed as a weighted sum of Pauli terms, construct a set of measurement circuits from commuting sets of Pauli terms. The measurement circuits are appended directly to the system register. Multiple circuits may be measured, depending on the number of Pauli terms in the operator. The final expectation value of the operator is then the sum over the product of Pauli term weights with probabilities of observing some eigenstate of the circuit.

apply (*supporter*)

Apply one or more error mitigation *Supporter* classes.

A supporter will modify the final shot table given its implementation.

Parameters

supporter (*Supporter*) – Error mitigation *Supporter*

Return type

_ProtocolPauliAveraging

copy ()

Returns a deep copy of the protocol.

Return type

_ProtocolPauliAveraging

class ProtocolIndirect

Bases: _ProtocolPauliAveraging

Calculate the expectation value of a Hermetian operator by operator averaging, with projective measurement.

Given a Hermitian operator expressed as a weighted sum of Pauli terms, construct a set of measurement circuits from commuting sets of Pauli terms where each Pauli gate in the measurement circuit is controlled on an ancilla qubit. Only the ancilla qubit is measured, this mesurement therefore represents the probability of observing an eigenstate of circuit under on of the commuting sets. The expectation value of the operator is then the sum over the product of weighted Pauli strings with the probabilites of observing some eigenstate of the circuit.

apply(*supporter*)

Apply one or more error mitigation *Supporter* classes.

A supporter will modify the final shot table given its implementation.

Parameters
supporter (*Supporter*) – Error mitigation *Supporter*

Return type
_ProtocolPauliAveraging

copy()

Returns a deep copy of the protocol.

Return type
_ProtocolPauliAveraging

22.11.2 Protocols for Overlaps

class ProtocolVacuum

Bases: _ProtocolPauliAveraging

Use operator averaging to measure the overlap between two states.

Implement the procedure described in arXiv:1810.02327.

Let $\Psi_1(\theta) = \hat{U}_1|0\rangle$ where the state $\Psi_1(\theta)$ is prepared by applying the unitary circuit $\hat{U}_1$ to the vacuum state. The overlap between two states, $|\langle\Psi_0|\Psi_1\rangle|$ can be calculated by measuring the quantity $\langle 0|\hat{U}_0^\dagger\hat{U}_1|0\rangle$. $\hat{U}_0^\dagger$ is the inverse of the circuit which prepares state $|\Psi_0\rangle$.

apply(*supporter*)

Apply one or more error mitigation *Supporter* classes.

A supporter will modify the final shot table given its implementation.

Parameters
supporter (*Supporter*) – Error mitigation *Supporter*

Return type
_ProtocolPauliAveraging

copy()

Returns a deep copy of the protocol.

Return type
_ProtocolPauliAveraging

class ProtocolDSP

Bases: _ProtocolPauliAveraging

Destructive Swap Test to calculate the overlap between two distinct states without ancillas.

Implements the destructive swap test procedure from arXiv:1303.6814.

The ansatz circuits for the states are palced in parallel registers. Swap gates are added between corresponding qubits which for the two states, effectively implementing an n qubit swap gate. The total phase shift between the states is given as $\pi \sum_{i=1}^{n} M_i^1 M_i^2$, where M_i^r is the measurement outcome on the i-th qubit of the first or second state. The swap test succeeded if the outcome of a bitwise AND on the state registers has even parity, the protocol therefore does not require an ancilla qubit.

apply(*supporter*)

Apply one or more error mitigation *Supporter* classes.

A supporter will modify the final shot table given its implementation.

Parameters
supporter (*Supporter*) – Error mitigation *Supporter*

Return type
_ProtocolPauliAveraging

copy()

Returns a deep copy of the protocol.

Return type
_ProtocolPauliAveraging

class ProtocolCSP

Bases: _ProtocolPauliAveraging

Evaluate an expectation value expression with Pauli averaging if the bra and ket states are orthogonal.

The cannonical swap test to test for ovarlap is described here.

The procedure inserts a swap operation between registers of the bra and ket state, controlled on the ancilla qubit in the $|+\rangle$ state. Applying another Hadamard gate to the ancilla and measuring gives the probability of the bra and ket states being orthogonal as $P(|0\rangle) = \frac{1}{2} + \frac{1}{2}|\langle\psi|\phi\rangle|^2$.

apply(*supporter*)

Apply one or more error mitigation *Supporter* classes.

A supporter will modify the final shot table given its implementation.

Parameters
supporter (*Supporter*) – Error mitigation *Supporter*

Return type
_ProtocolPauliAveraging

copy()

Returns a deep copy of the protocol.

Return type
_ProtocolPauliAveraging

22.11.3 Statevector-Based Protocols

class ProtocolStateVectorSparse(*cache=None*, *caching_decorator=cached*)

Bases: StateVectorProtocol

Provides tools for sparse statevector calculations using caching.

Uses an externally provided cache-handling class and an external caching decorator to wrap its methods to be cached.

Parameters

- **cache** (Optional[Cache], default: None) – A cache-handling class object.
- **caching_decorator** (Callable[[Callable[..., Any]], Callable[..., Any]], default: cached) – A caching decorator, that needs to take both an arbitrary function and a cache-handler as parameters.

property cache: Cache

Returns the cache-handling object.

Return type
Cache

cache_hit_report()

Returns a cache hit report in the Pandas DataFrame format.

Return type
Optional[DataFrame]

clear(*keep_cache=False*)

Clears object by nullifying member variables.

Parameters
keep_cache (bool, default: False) – Whether to keep cache intact.

Return type
None

copy()

Returns a deep copy of the protocol.

Return type
Protocol

property parameters: *SymbolDict*

Returns current protocol parameters.

Return type
SymbolDict

class ProtocolStateVectorBackendSupport

Bases: StateVectorProtocol

ProtocolStateVector utilising backend functionality to enable statevector simulations.

copy()

Returns a deep copy of the protocol.

Return type
Protocol

22.11.4 Protocols for Derivatives

class ProtocolHadamardDerivativeOverlap

Bases: _ProtocolPauliAveraging

Calculate the derivative of an expectation value expression, using an ancilla qubit.

Implements the linear combination of unitaries method from arXiv:1811.11184.

The derivative of a unitary gate can be written as a linear combination of unitary matrices: $\partial_\mu G = \sum_{k=1}^{K} a_k A_k$. Where a_k is some real coefficient and A_k a unitary matrix. The derivative of an expectation value expression becomes $\partial_\mu f = \sum_{k=1}^{K} a_k(\langle\Psi|G^\dagger \hat{O} A_k|\Psi\rangle) + h.c.$, where $\hat{O}$ is a Hermitian observable. Each element of the sum can be evaluated on a device with a Hadamard test where each *G* and A is controlled on the ancilla qubit.

apply(*supporter*)

Apply one or more error mitigation *Supporter* classes.

A supporter will modify the final shot table given its implementation.

Parameters
supporter (*Supporter*) – Error mitigation *Supporter*

Return type
_ProtocolPauliAveraging

copy()

Returns a deep copy of the protocol.

Return type
_ProtocolPauliAveraging

class ProtocolHadamardDirectPauliY

Bases: _ProtocolPauliAveraging

Measure the gradient of an expectation value expression with respect to a Hamiltonian.

The required ancilla is measured in the Y basis.

Implements the procedure described in arXiv:1701.01450.

The circuit prepares the state $|\Psi_\mu\rangle = \frac{1}{2}[(|\sigma\rangle + i\hat{w}_n^P \hat{\sigma}_\mu^{(G)} \hat{W}_1^{n-1}|\phi_0\rangle) \otimes |0\rangle) + (|\sigma\rangle - i\hat{w}_n^P \hat{\sigma}_\mu^{(G)} \hat{W}_1^{n-1}|\phi_0\rangle) \otimes |1\rangle)]$.

Where $\hat{W}_n^k = U_k(\sigma_k) \ldots U_n(\sigma_n)$ and is the ansatz circuit acting on state $|\phi_0\rangle$. *G* is the generator of Pauli gates representing the device register. Given $\langle i|\hat{\sigma}_y^{(a)}|j\rangle = (-1)^j \delta_{ij}$, we can measure $\langle\Psi_\mu|\hat{\sigma}_v^{(C)} \otimes \hat{\sigma}_y^{(a)}|\Psi_\mu\rangle$ where $\hat{\sigma}_v^{(C)}$ are the pauli terms of the Hamiltonian measured directly on the system register and $\hat{\sigma}_y^{(a)}$ is the Pauli Y gate applied to the ancilla register.

apply(*supporter*)

Apply one or more error mitigation *Supporter* classes.

A supporter will modify the final shot table given its implementation.

Parameters
supporter (*Supporter*) – Error mitigation *Supporter*

Return type
_ProtocolPauliAveraging

copy()

Returns a deep copy of the protocol.

Return type
_ProtocolPauliAveraging

class ProtocolHadamardDirectPauliZ

Bases: _ProtocolPauliAveraging

Measure the gradient of an expectation value expression with respect to a Hamiltonian.

The required ancilla is measured in the Z basis.

Implements the procedure described in arXiv:1701.01450.

The circuit prepares the state $|\Psi_\mu\rangle = \frac{1}{2}[(|\sigma\rangle + i\hat{w}_n^P \hat{\sigma}_\mu^{(G)} \hat{W}_1^{n-1}|\phi_0\rangle) \otimes |0\rangle) + (|\sigma\rangle - i\hat{w}_n^P \hat{\sigma}_\mu^{(G)} \hat{W}_1^{n-1}|\phi_0\rangle) \otimes |1\rangle)]$.

Where $\hat{W}_n^k = U_k(\sigma_k) \dots U_n(\sigma_n)$ and is the ansatz circuit acting on state $|\phi_0\rangle$. *G* is the generator of Pauli gates representing the device register. Given $\langle i|\hat{\sigma}_z^{(a)}|j\rangle = (-1)^j \delta_{ij}$, we can measure $\langle\Psi_\mu|\hat{\sigma}_v^{(C)} \otimes \hat{\sigma}_z^{(a)}|\Psi_\mu\rangle$ where $\hat{\sigma}_v^{(C)}$ are the pauli terms of the Hamiltonian measured directly on the system register and $\hat{\sigma}_z^{(a)}$ is the Pauli Z gate applied to the ancilla register.

apply(*supporter*)

Apply one or more error mitigation *Supporter* classes.

A supporter will modify the final shot table given its implementation.

Parameters
supporter (*Supporter*) – Error mitigation *Supporter*

Return type
_ProtocolPauliAveraging

copy()

Returns a deep copy of the protocol.

Return type
_ProtocolPauliAveraging

class ProtocolHadamardIndirectPauliY

Bases: _ProtocolPauliAveraging

Measure the gradient of an expectation value with respect to a Hamiltonian operator.

Uses an ancilla qubit to apply an effective derivative of Pauli terms in the Hamiltonian and an indirect measurement of the system register with a second ancilla.

Implements a variant of the procedure described in arXiv:1701.01450.

Prepare the same effective state as *ProtocolHadamardDirectPauliY*, use the procedure described in *ProtocolIndirect* to measure the system register.

apply(*supporter*)

Apply one or more error mitigation *Supporter* classes.

A supporter will modify the final shot table given its implementation.

Parameters
supporter (*Supporter*) – Error mitigation *Supporter*

Return type
_ProtocolPauliAveraging

copy()

Returns a deep copy of the protocol.

Return type
_ProtocolPauliAveraging

class ProtocolHadamardIndirectPauliZ

Bases: _ProtocolPauliAveraging

Measure the gradient of an expectation value with respect to a Hamiltonian operator.

Uses an ancilla qubit to apply an effective derivative of Pauli terms in the Hamiltonian and an indirect measurement of the system register with a second ancilla.

Implements a variant of the procedure described in arXiv:1701.01450.

Prepare the same effective state as *ProtocolHadamardDirectPauliZ*, use the procedure described in *ProtocolIndirect* to measure the system register.

apply(*supporter*)

Apply one or more error mitigation *Supporter* classes.

A supporter will modify the final shot table given its implementation.

Parameters
supporter (*Supporter*) – Error mitigation *Supporter*

Return type
_ProtocolPauliAveraging

copy()

Returns a deep copy of the protocol.

Return type
_ProtocolPauliAveraging

class ProtocolMidMeasurementGradient

Bases: _ProtocolPauliAveraging

Calculate gradient of an expectation value expression using mid-circuit measurement.

apply(*supporter*)

Apply one or more error mitigation *Supporter* classes.

A supporter will modify the final shot table given its implementation.

Parameters
supporter (*Supporter*) – Error mitigation *Supporter*

Return type
_ProtocolPauliAveraging

copy()

Returns a deep copy of the protocol.

Return type
_ProtocolPauliAveraging

class ProtocolPhaseShift

Bases: _ProtocolPauliAveraging

Calculate gradient of an expectation value expression using a gate parameter shift.

Implements the parameter-shift rule described in arXiv:1811.11184.

The derivative of an expectation value expression can be written as $\partial_\mu f = \langle\Psi|G^\dagger\hat{O}(\partial_\mu G)|\Psi\rangle + h.c.$, where $G(\mu) = e^{-i\mu G}$ is the gate generated from a Hermitian operator G and $\hat{O}$ is a Hermitian observable. We assume

that a single parameter affects a single gate. The derivative of the gate is given as $\partial_\mu G = -iGe^{-i\mu G}$. Where the parametrised gate decomposition of a variational circuit unitary has two distinct eigenvalues, using the identity $G(\frac{\pi}{4r}) = \frac{1}{\sqrt{2}}(\mathbb{1} - ir^{-1}G)$, where r are the gate's eigenvalues, $\partial_\mu f$ can be estimated using two circuit evaluations by placing either the gate $G(\frac{\pi}{4})$ or $G(\frac{-\pi}{4})$ next to the gate being differentiated.

The final measurement follows the procedure from *ProtocolDirect*.

apply(*supporter*)

Apply one or more error mitigation *Supporter* classes.

A supporter will modify the final shot table given its implementation.

Parameters
supporter (*Supporter*) – Error mitigation *Supporter*

Return type
_ProtocolPauliAveraging

copy()

Returns a deep copy of the protocol.

Return type
_ProtocolPauliAveraging

class ProtocolVacuumPhaseShift

Bases: _ProtocolPauliAveraging

Calculate gradient of an expectation value expression using a gate parameter shift.

Where the bra and ket are the vacuum state $|0\rangle$.

Implements the parameter-shift rule described in arXiv:1811.11184.

The derivative of an expectation value expression can be written as $\partial_\mu f = \langle\Psi|G^\dagger\hat{O}(\partial_\mu G)|\Psi\rangle + h.c.$, where $G(\mu) = e^{-i\mu G}$ is the gate generated from a Hermitian operator G and $\hat{O}$ is a Hermitian observable. We assume that a single parameter affects a single gate. The derivative of the gate is given as $\partial_\mu G = -iGe^{-i\mu G}$. Where the parametrised gate decomposition of a variational circuit unitary has two distinct eigenvalues, using the identity $G(\frac{\pi}{4r}) = \frac{1}{\sqrt{2}}(\mathbb{1} - ir^{-1}G)$, where r are the gate's eigenvalues, $\partial_\mu f$ can be estimated using two circuit evaluations by placing either the gate $G(\frac{\pi}{4})$ or $G(\frac{-\pi}{4})$ next to the gate being differentiated.

The final measurement follows the procedure from *ProtocolDirect*.

apply(*supporter*)

Apply one or more error mitigation *Supporter* classes.

A supporter will modify the final shot table given its implementation.

Parameters
supporter (*Supporter*) – Error mitigation *Supporter*

Return type
_ProtocolPauliAveraging

copy()

Returns a deep copy of the protocol.

Return type
_ProtocolPauliAveraging

22.11.5 Supporters

Inquanto.protocol.suuporters module provides supporting logic to apply error mitigation to protocols.

class SupporterPMSV(*symmetries*)

Bases: Supporter

Partition Measurement Symmetry Verification appends physical symmmetries of a system into the pauli-strings measured by a quantum circuit.

Implements the PMSV error mitigation procedure described in arXiv:2109.08401

Shots are discarded when the parity of qubits which correspond to the measured Pauli words, does not match the expected parity given the symmetry of the provided *QubitOperatorStrings*.

Parameters

symmetries (List[*QubitOperator*]) – QubitOperatorString is the symmetry and coeff is (-1) ** x, where x is the target parity of the measured distribution.

calibrate()

The PMSV Supporter does not require calibration.

Return type

SupporterPMSV

class SupporterSPAM(*nodes*)

Bases: Supporter

Interface to tket's State Preparation And Measurement (SPAM) correction utility.

See pytket SPAM API reference

Parameters

nodes (List[Node]) –

calibrate(*backend*, *n_shots=8192*)

Generate the SPAM transition matrix by running tomography circuits.

Parameters

- **backend** (Backend) – pytket Backed to use for calibration.
- **n_shots** (int, default: 8192) – Number of shots to run for calibration circuits.

Return type

SupporterSPAM

22.11.6 PauliCollection

class PauliCollection(**elements*)

Bases: object

Constructs a Partitioned Set of QubitPauliStrings where each partition is a commuting set.

Parameters

elements (Union[List[QubitPauliOperator], List[QubitPauliString], QubitPauliOperator, QubitPauliString, *PauliCollection*]) – elements of the partitioned set of Pauli strings.QubitPauliOperator object, QubitPauliString object, or a list of objects of listed types

add(*elements*)

Add new Pauli strings to the collection object.

Parameters

elements (Union[List[QubitPauliOperator], List[QubitPauliString], QubitPauliOperator, QubitPauliString, *PauliCollection*]) – QubitPauliOperator object, QubitPauliString object, or a list of objects of listed types

copy()

Return a copy of this object.

Return type

PauliCollection

df()

Return PauliCollection as a pandas DataFrame.

get(*pauli*)

Access QubitPauliStrings.

Parameters

pauli (QubitPauliString) –

map(*func*)

Map function to QubitPauliString collection.

partitioning(*conflicting_sets=True*)

Partition collection of QubitPauliStrings into commuting or non conflicting sets.

Parameters

conflicting_sets (bool, default: True) –

property partitions: List[List[QubitPauliString]]

Access commuting sets of QubitPauliStrings.

Return type

List[List[QubitPauliString]]

22.12 inquanto.spaces

This module provides a class for operations related to particle occupation-space and generic qubit-space.

class FermionSpace(*n_spin_orb*, *point_group=None*, *orb_irreps=None*)

Bases: OccupationSpace

A class representing utilities associated with a fermionic occupation space.

Parameters

- **n_spin_orb** (int) – The number of spin orbitals in the system.
- **point_group** (Union[*PointGroup*, str, None], default: None) – The point group symmetry of the system.
- **orb_irreps** (Optional[List[str]], default: None) – The point group irreducible representations of the orbitals.

COLUMN_ORB = 1

The orbital column in the table underlying this object.

COLUMN_SPIN = 0

The spin column in the table underlying this object.

SPIN_ALPHA = 0

Integer representation of alpha spin.

SPIN_BETA = 1

Integer representation of beta spin.

SPIN_DOWN = 1

Integer representation of spin down.

SPIN_UP = 0

Integer representation of spin up.

construct_contraction_mask_from_operators(*operators*)

Constructs a contraction mask based on the occurring indices in the operators.

Parameters

operators (Union[*FermionOperator*, List[*FermionOperator*]]) – An operator or list of operators.

Returns

List[int] – The mask defining the contraction.

construct_double_excitation_operators(*fermion_state*)

State-specific, occupied-to-virtual double excitations conserving azimuthal spin.

This corresponds to $\hat{a}_p^\dagger \hat{a}_q \hat{a}_r^\dagger \hat{a}_s$.

Parameters

fermion_state (*FermionState*) – Representation of the reference fermionic number occupation vector.

Returns

FermionOperatorList – The double excitation operators stored independently.

construct_double_ucc_operators(*fermion_state*)

Generate a list of anti-hermitian double excitation operators based on a reference fermion state.

Each excitation is chemist ordered and consists of an "excitation" - "de-excitation", $a_p^\dagger a_q a_r^\dagger a_s - a_s^\dagger a_r a_q^\dagger a_p$, where p is virtual, q is occupied, r is virtual and s is occupied.

Parameters

fermion_state (*FermionState*) – Reference fermionic occupation state.

Returns

FermionOperatorList – The double excitation UCC operators stored independently.

construct_generalised_double_excitation_operators()

Construct generalised double excitations conserving azimuthal spin.

Generalised excitations include occupied-to-occupied and virtual-to-virtual excitation operators.

$\hat{a}_p^\dagger \hat{a}_q \hat{a}_r^\dagger \hat{a}_s$.

Returns

FermionOperatorList – Generalised double excitation operators stored independently.

construct_generalised_double_ucc_operators()

Generate a list of generalised anti-hermitian double excitation operators.

Each UCC excitation consists of an "excitation - de-excitation", $\left(a_p^\dagger a_q - a_q^\dagger a_p\right)$, where p, q are generalised indices, virtual-virtual, occupied-occupied, and occupied-virtual excitations are all allowed.

Notes

Excitations no longer distinguish occupied and virtual spaces, therefore fermion_state is not used here. Note that spin crossover excitations (that change number of alphas/betas) are not allowed. See arxiv 1810.02327.

Returns

FermionOperatorList – Generalised double UCC operators stored independently.

construct_generalised_pair_double_excitation_operators()

Generalised pair-double excitations from molecular orbital to molecular orbital.

Generalised pair-double excitations include occupied-occupied, virtual-virtual and occupied-virtual double excitations from two spin orbitals with equivalent spatial components, to two spin orbitals with equivalent spatial components. This corresponds to $\hat{a}_p^\dagger \hat{a}_q \hat{a}_r^\dagger \hat{a}_s$.

Returns

FermionOperatorList – Generalised MO-MO double excitations

construct_generalised_pair_double_ucc_operators()

Generate a list of generalised anti-hermitian pair-double excitation operators.

Each UCC excitation consists of an "excitation - de-excitation", $a_p^\dagger a_q a_r^\dagger a_s - a_s^\dagger a_r a_q^\dagger a_p$, where p, q, r, s are generalised indices, but restricted to paired spins, so that p, r are in one spatial orbital, and q, s are in another spatial orbital.

Notes

Excitations no longer distinguish occupied and virtual spaces, therefore fermion_state is not used here. Note that spin crossover excitations (that change number of alphas/betas) are not allowed. See arXiv:1810.02327.

Returns

FermionOperatorList – Generalised MO-MO double excitations stored independently

construct_generalised_single_excitation_operators()

Construct generalised single excitations conserving azimuthal spin.

Generalised excitations include occupied-to-occupied and virtual-to-virtual excitations. This corresponds to $\hat{a}_p^\dagger \hat{a}_q \hat{a}_r^\dagger \hat{a}_s$.

Returns

FermionOperatorList – Generalised single excitation operators stored independently.

construct_generalised_single_ucc_operators()

Generate a list of generalised anti-hermitian single excitation operators.

Each UCC excitation consists of an "excitation - de-excitation", $\left(a_p^\dagger a_q - a_q^\dagger a_p\right)$, where p, q are generalised indices, virtual-virtual, occupied-occupied, and occupied-virtual excitations are all allowed.

Notes

Excitations no longer distinguish occupied and virtual spaces, therefore fermion_state is not used here. Note that spin crossover excitations (that change number of alphas/betas) are not allowed. See arXiv:1810.02327.

Returns
FermionOperatorList – Generalised single UCC operators stored independently

construct_n_body_spinless_pdm_operators(*rank*)

Return a matrix of FermionOperators for measurement of spin-traced n-PDM.

PDM is what is called Pre-Density Matrix in Orca and returned by pyscf.fci.rdm.make_dm1234. In this function, 2pdm[p,q,r,s] is $\langle p^\dagger q r^\dagger s \rangle$; 3pdm[p,q,r,s,t,u] is $\langle p^\dagger q r^\dagger s t^\dagger u \rangle$; 4pdm[p,q,r,s,t,u,v,w] is $\langle p^\dagger q r^\dagger s t^\dagger u v^\dagger w \rangle$. 1pdm is the same as the normal one.

Parameters
rank (int) – Rank of the n-PDM.

Returns
A 2n-dimensional matrix of FermionOperators.

construct_n_body_spinless_rdm_operators(*rank*, *ordering='p^r^sq'*)

Return a matrix of FermionOperators for measurement of spin-traced n-RDM.

Parameters

- **rank** (int) – Rank of the n-RDM.
- **ordering** (str, default: "p^r^sq") – Ordering of indices, either p^r^sq or p^q^sr. The default is PySCF's p^r^sq, meaning that rdm2[p,q,r,s] = <p^r^sq>, rdm3[p,q,r,s,t,u] = <p^r^t^usq> etc. The alternative, p^q^sr, corresponds to the following convention: rdm2[p,q,r,s] = <p^q^sr>, rdm3[p,q,r,s,t,u] = <p^q^r^uts> etc.

Returns
ndarray[Any, dtype[*FermionOperator*]] – A 2n-dimensional matrix of FermionOperators.

construct_number_alpha_operator()

Construct a FermionOperator which represents a number operator acting on all alpha spin orbitals.

Returns
FermionOperator – The number operator acting on all alpha spin-orbitals.

construct_number_beta_operator()

Construct a FermionOperator which represents a number operator acting on all beta spin orbitals.

Returns
FermionOperator – The number operator operator acting on all beta spin-orbitals.

construct_number_operator()

Creates and returns a FermionOperator representation of the number operator.

The FermionOperator acts on the entire spin orbital system.

Returns
FermionOperator – The number operator of the fermionic space.

construct_one_body_operator_from_integral(*one_body_spatial*, *spins*, *spatial_mask=None*)

Constructs a one-body operator from one-body spatial integral.

The indices are transformed to spin-orbital notation, with alternating alpha and beta spins. The one-body operator is given by, $\hat{h} = \sum_{pq} h_p q a_p^\dagger a_q$.

Parameters

- **one_body_spatial** (ndarray[Any, dtype[float]]) – Integrals in chemists notation.
- **spins** (Tuple[int, int]) – The spin configuration, e.g. (0,0) = (SPIN_UP, SPIN_UP).
- **spatial_mask** – Mask filtering the generations of terms in the operator.

Returns

FermionOperator – The one-body Hamiltonian operator.

construct_one_body_spatial_rdm_operators(*spins*, *spatial_mask=None*, *operator_pattern=(1, 0)*)

Construct a rank-2 numpy.ndarray with FermionOperator as the datatype.

Parameters

- **spins** (Tuple[int, int]) – The corresponding spins as (s_i,s_j).
- **spatial_mask** (Optional[List[bool]], default: None) – Mask filtering the generations of terms in the operator.
- **operator_pattern** (Tuple[int, int], default: (1, 0)) – The creation and annihilation ordering of the terms in the array. Must be a length 2 tuple containing ones and zeros, which correspond to creation and annhiliation operators, respectively.

Returns

ndarray[Any, dtype[*FermionOperator*]] – The one body spatial RDM operator as a rank-2 array of FermionOperators.

static construct_operator_from_string(*input*)

Construct an operator from a string.

Parameters

input (str) – Stringified inquanto.operator.FermionOperator.

Returns

FermionOperator – An instance of inquanto.operator.FermionOperator corresponding to the string provided.

construct_orbital_number_operators()

Construct a list of FermionOperators, where each element is a number operator on a specific spin orbital.

The position in the returned list corresponds to the spin orbital index.

Returns

List[*FermionOperator*] – A list of the individual orbital number operators.

static construct_scalar_operator(*value*)

Construct a FermionOperator with only the identity term and a corresponding variable coefficient.

Parameters

value (float) – Corresponding coefficient to the identity term.

Returns

FermionOperator – A fermionic operator containing a single identity term multiplied by a scalar coefficient.

construct_single_excitation_operators(*fermion_state*)

State-specific single excitations conserving azimuthal spin.

This includes only occupied-to-virtual excitations corresponding to the fermion_state argument.

Parameters
fermion_state (*FermionState*) – Representation of the reference fermionic number occupation vector.

Returns
FermionOperatorList – The single excitation operators which excite occupied orbitals in the input state.

construct_single_ucc_operators(*fermion_state*)

Generate a list of anti-hermitian single excitation operators based on a reference fermion state.

Each UCC excitation consists of an "excitation - de-excitation", $\left(a_p^\dagger a_q - a_q^\dagger a_p\right)$, where p is virtual and q is occupied.

Parameters
fermion_state (*FermionState*) – Reference fermionic occupation state.

Returns
FermionOperatorList – The UCC single operators which excite occupied orbitals in the provided state.

construct_singlet_double_excitation_operators(*fermion_state*)

Generate a list of spin-adapted singlet double excitation operators based on a reference fermion_state.

Each excitation is chemist ordered and spin-paired, $a_{p_0}^\dagger a_{q_0} a_{r_0}^\dagger a_{s_0} + a_{p_1}^\dagger a_{q_1} a_{r_0}^\dagger a_{s_0} + a_{p_0}^\dagger a_{q_0} a_{r_1}^\dagger a_{s_1} + a_{p_1}^\dagger a_{q_1} a_{r_1}^\dagger a_{s_1}$, where p is virtual, q is occupied, r is virtual and s is occupied.

Parameters
fermion_state (*FermionState*) – Representation of the reference fermionic number occupation vector.

Returns
FermionOperatorList – The spin-adapted singlet double excitation operators which excite occupied orbitals in the reference state.

construct_singlet_double_ucc_operators(*fermion_state*)

Generate a list of anti-hermitian singlet double excitation operators based on a reference fermion state.

Each excitation is chemist orderedand consists of an "excitation" - "de-excitation", $a_p^\dagger a_q a_r^\dagger a_s - a_s^\dagger a_r a_q^\dagger a_p$, where p is virtual, q is occupied, r is virtual and s is occupied.

Parameters
fermion_state (*FermionState*) – Reference fermionic occupation state.

Returns
FermionOperatorList – The singlet UCC double excitations which excite occupied orbitals in the input state.

construct_singlet_generalised_double_excitation_operators()

Construct generalised double excitations conserving azimuthal spin and adapted for singlet spin symmetry.

Generalised excitations include occupied-to-occupied, virtual-to-virtual and occupied-to-virtual excitations.

Each excitation is chemist ordered and spin-paired, $a_{p_0}^\dagger a_{q_0}^\dagger a_{r_0} a_{s_0} + a_{p_1}^\dagger a_{q_1}^\dagger a_{r_1} a_{s_1}$, where 0 and 1 indicate alpha and beta spin, respectively.

Returns
FermionOperatorList – The generalised double excitations.

construct_singlet_generalised_single_excitation_operators()

Generalised single excitations conserving azimuthal spin and adapted for singlet spin symmetry.

Generalised excitations include occupied-to-occupied, virtual-to-virtual and occupied-to-virtual excitations.

Each excitation is chemist ordered and spin-paired, $a^\dagger_{p_0} a_{q_0} + a^\dagger_{p_1} a_{q1}$, where 0 and 1 indicate alpha and beta spin, respectively.

Returns

FermionOperatorList – The singlet generalised single excitations.

construct_singlet_generalised_single_ucc_operators()

Generate a list of generalised anti-hermitian single singlet spin-adapted excitation operators.

Each UCC excitation consists of an "excitation - de-excitation", $\left(a^\dagger_p a_q - a^\dagger_q a_p\right)$, where both p and q run over occupied and virtual orbitals.

Notes

Excitations no longer distinguish occupied and virtual spaces, therefore fermion_state is not used here. Note that spin crossover excitations (that change number of alphas/betas) are not allowed. See *arXiv:1810.02327 <https://arxiv.org/abs/1810.02327>*.

Returns

FermionOperatorList – The singlet generalised UCC single excitations.

construct_singlet_single_excitation_operators(*fermion_state*)

Generate a list of spin-adapted singlet single excitation operators based on a reference fermion_state.

Each excitation is chemist ordered and spin-paired, $a^\dagger_{p_0} a_{q_0} + a^\dagger_{p_1} a_{q1}$, where p is virtual, q is occupied, and 0 or 1 determine alpha or beta spin, respectively.

Parameters

fermion_state (*FermionState*) – Representation of the reference fermionic number occupation vector.

Returns

FermionOperatorList – The singlet single excitation operators which excite occupied orbitals in the input state.

construct_singlet_single_ucc_operators(*fermion_state*)

Generate a list of anti-hermitian singlet single excitation operators based on a reference fermion state.

Each UCC excitation consists of an "excitation - de-excitation", $\left(a^\dagger_p a_q - a^\dagger_q a_p\right)$, where p is virtual and q is occupied.

Parameters

fermion_state (*FermionState*) – A fermionic occupation state.

Returns

FermionOperatorList – The singlet UCC single excitation operators which excite occupied orbitals in the input state

construct_spin_operator()

This function creates and returns a FermionOperator representation of the spin S^2 operator.

Returns

FermionOperator – The S^2 operator of the fermionic state.

construct_sz_operator()

This function creates and returns a FermionOperator representation of the spin Sz operator.

Returns

FermionOperator – FermionOperator representing the Sz operator.

construct_triplet_generalised_single_excitation_operators()

Construct generalised single excitations conserving azimuthal spin and adapted for triplet spin symmetry.

Each excitation is chemist ordered and spin-paired, $a^\dagger_{p_0} a_{q_0} - a^\dagger_{p_1} a_{q1}$, $a^\dagger_{p_0} a_{q_1}$, or $a^\dagger_{p_1} a_{q_0}$, where p is virtual, q is occupied, and 0 or 1 determine alpha or beta spin, respectively.

Returns

FermionOperatorList – The triplet generalised single excitation operators.

construct_triplet_generalised_single_ucc_operators()

Generate a list of generalised anti-hermitian single excitation operators, spin adapted to a triplet.

Each UCC excitation consists of an "excitation - de-excitation", $\left(a^\dagger_p a_q - a^\dagger_q a_p\right)$, where p, q are generalised indices.

Notes

Excitations no longer distinguish occupied and virtual spaces, therefore fermion_state is not used here. Note that spin crossover excitations (that change number of alphas/betas) are not allowed. See *arXiv:1810.02327 <https://arxiv.org/abs/1810.02327>*

Returns

FermionOperatorList – Generalised triplet anti-hermitian single excitation operators

construct_two_body_operator_from_integral(*two_body_spatial*, *spins*, *spatial_mask=None*)

Constructs a two-body operator from two-body spatial integral.

$\sum_{ijkl}(ij|kl)a^\dagger_{i,s_i}a^\dagger_{k,s_k}a_{l,s_l}a_{j,s_j}$ where $(s_i, s_j, s_k, s_l) =$ spins.

Notes

The integrals are in chemists notation, that is two_body_spatial[i,j,k,l] = (ij|kl).

Parameters

- **two_body_spatial** (ndarray[Any, dtype[float]]) – Integrals in chemists notation.
- **spins** (Tuple[int, ...]) – The corresponding spins as (s_i,s_j,s_k,s_l).
- **spatial_mask** (Optional[List[bool]], default: None) – mask filtering the generations of terms in the operator.

Returns

FermionOperator – The two-body Hamiltonian operator.

construct_two_body_operator_from_tensor(*two_body_tensor*, *spins*, *spatial_mask=None*)

Constructs a two-body operator from two-body spatial integral.

That is, $\sum_{ijkl}(ij|kl)a^\dagger_{i,s_i}a^\dagger_{k,s_k}a_{l,s_l}a_{j,s_j}$ where $(s_i, s_j, s_k, s_l) =$ spins.

Notes

The integrals are in chemists notation, that is two_body_tensor[i,k,l,j] = (ij|kl). The difference from two_body_spatial in construct_two_body_operator_from_integral(…) is how the two electron integral is encoded in a tensor format. It is recommended to use construct_two_body_operator_from_integral() and two_body_spatial[i,j,k,l] = (ij|kl). This method is for compatibility.

Parameters

- **two_body_tensor** – Integrals in chemists notation.
- **spins** – The corresponding spins as (s_i,s_j,s_k,s_l).
- **spatial_mask** (Optional[List[bool]], default: None) – Mask filtering the generations of terms in the operator.

Returns

FermionOperator – The two-body Hamiltonian operator.

construct_two_body_spatial_rdm_operators(*spins*, *spatial_mask=None*, *operator_pattern=(1, 1, 0, 0)*)

Construct a rank-4 array of FermionOperators corresponding to two-particle reduced density matrix elements.

Parameters

- **spins** (Tuple[int]) – The corresponding spins as (s_i,s_j,s_k,s_l).
- **spatial_mask** (Optional[List[bool]], default: None) – Mask filtering the generations of terms in the operator.
- **operator_pattern** (Tuple[int], default: (1, 1, 0, 0)) – The creation and annihilation ordering of the terms in the array. Must be a length 4 tuple containing ones and zeros, which correspond to creation and annhiliation operators, respectively.

Returns

ndarray[Any, dtype[*FermionOperator*]] – The two body spatial RDM operators in a numpy array.

contract_occupation_space(*active_spatial_orbs*)

Generate an occupation space with a set of restricted spatial orbitals.

Parameters

active_spatial_orbs (List[int]) – Indices of spatial orbitals considered to be active.

Returns

Tuple[*FermionSpace*, List[bool]] – A tuple containing the contracted occupation space and the contraction mask.

static contract_occupation_state(*occupation_state*, *contraction_mask*)

Contracts fermion state according to a mask.

Parameters

- **occupation_state** (*FermionState*) – A FermionState to be contracted.
- **contraction_mask** (List[bool]) – Mask, where True, the spin-orbital is frozen.

Returns

FermionState – The contracted state.

static contract_operator(*fermion_state*, *contraction_mask*, *operator*)

Contracts operator according to a mask and the fermion state.

Freezes spin-orbitals according to the known occupation numbers in fermion state.

Notes

This currently works only with inquanto.operators.FermionOperator.

Parameters

- **fermion_state** (*FermionState*) – The reference fermionic state.
- **contraction_mask** (List[bool]) – Mask, where True, the spin-orbital is frozen.
- **operator** (Union[List[*FermionOperator*], *FermionOperator*]) – The operator to be contracted.

Returns

Union[List[*FermionOperator*], *FermionOperator*] – The contracted operator.

static contract_state_mask(*state_mask*, *contraction_mask*)

Contracts mask according to another mask.

Parameters

- **state_mask** – A state mask.
- **contraction_mask** (List[bool]) – Mask defining a contraction.

Returns

List[bool] – The contracted mask.

contracted_system(*contraction_mask*, *fermion_state*, **operators*)

Contract system (space, state and operator(s)) according to the reference state and a contraction mask.

Parameters

- **contraction_mask** (List[bool]) – Mask, where True, the orbital is frozen.
- **fermion_state** (*FermionState*) – Fermionic occupation state to be contracted.
- **operators** (*FermionOperator*) – One or more FermionOperators to be contracted.

Returns

A tuple containing the contracted FermionSpace, contracted FermionState, contracted FermionOperator objects.

static convert_mask_to_index_map(*mask*)

Convert a mask into an index mapping.

Parameters

mask (Union[ndarray[Any, dtype[bool]], List[bool]]) – A boolean mask with the length of the fock space.

Returns

List[int] – A list that maps to the indices where the mask is True.

count (*fermion_state=None*, *state_value=1*, *column=None*, *column_value=None*, *state_mask=None*)

Counts spin orbitals or modes according to the relevant quantum number and state value.

This is a depreciated legacy method and will be removed in subsequent releases.

Notes

If some value is None, then it is counted.

Parameters

- **fermion_state** (Optional[*FermionState*], default: None) – A FermionState object.
- **state_value** (int, default: 1) – 1/0 occupied/unoccupied, respectively.
- **column** (Optional[int], default: None) – The column index in the quantum number table.
- **column_value** (Optional[int], default: None) – The value to be counted.
- **state_mask** (Optional[List[bool]], default: None) – State mask defining which spin orbitals or modes are included in the count.

Returns

int – Number of orbitals or modes satisfying the input arguments.

static from_state (*fermion_state*)

Initialize a *FermionSpace* from an input fermion state.

Parameters

fermion_state (*FermionState*) – A inquanto.operator.FermionState instance representing occupation of a fermionic system.

Returns

FermionSpace –:

An instance of *FermionSpace* with C1 point group symmetry
and n_spin_orb equal to the size of the FermionState input.

generate_cyclic_masks (*window*, *shift=None*)

Generate a complete set of masks with windows shifted by window/2 if the shift argument is not provided.

Parameters

- **window** (int) – Number of spin orbitals in the window.
- **shift** (Optional[int], default: None) – Size of the steps to take along the occupation number vector when defining the windows.

Returns

A list of masks.

Examples

```
>>> space = FermionSpace(8)
>>> space.generate_cyclic_masks(window=4, shift=2)
array([[ True,  True,  True,  True, False, False, False, False],
```

(continues on next page)

(continued from previous page)

```
       [False, False,  True,  True,  True,  True, False, False],
       [False, False, False, False,  True,  True,  True,  True],
       [ True,  True, False, False, False, False,  True,  True]])
```

generate_cyclic_window_mask(*window*, *shift=0*)

Generate a mask with a window and shift.

Parameters

- **window** (int) – number of spin orbitals within the window.
- **shift** – The position in the occupation number vector to begin the window.

Returns

A list of booleans defining the mask.

Examples

```
>>> space = FermionSpace(8)
>>> space.generate_cyclic_window_mask(window=4, shift=2)
array([False, False,  True,  True,  True,  True, False, False])
```

generate_occupation_state(*n_fermion=0*, *multiplicity=None*)

Generate an instance of FermionState with a variable number of fermions.

Fermions are placed in lower indexed orbitals first. The number of unpaired alpha electrons is given by the multiplicity - 1.

Parameters

- **n_fermion** (int, default: 0) – Number of Fermions to include in FermionState.
- **multiplicity** (Optional[int], default: None) – Multitplicity of the generated FermionState.

Returns

FermionState – The generated fermionic state.

generate_occupation_state_from_list(*occupation_state_list=None*)

Convert a list of occupation numbers to a FermionState.

Parameters

occupation_state_list (Optional[List[int]], default: None) – Represents spin-orbital occupations. Default value is None, which returns all zeros FermionState.

Returns

FermionState – The generated fermionic state.

generate_occupation_state_from_spatial_occupation(*occupation*)

Generates a state from spatial orbital occupation numbers.

Parameters

occupation (List[int]) – List of the length of the number of spatial orbitals. Entries are {0,1,2} and indicate the occupation number of the corresponding spatial orbital.

Returns

FermionState – The generated fermionic state.

generate_subspace_singles()

Sequentially yield all single excitation operators as FermionOperator objects.

Yields
Fermion operator objects corresponding to single excitations.

Return type
Generator[*FermionOperator*, None, None]

generate_subspace_singlet_singles()

Sequentially yield spin-adapted singlet-single excitation operators to maintaining spin symmetry.

Yields
FermionOperator objects corresponding to spin-adapted singlet single excitation operators.

Return type
Generator[*FermionOperator*, None, None]

generate_subspace_triplet_singles()

Sequentially yield triplet-single excitation operators (subspace of the singlet-singles operators).

Yields
FermionOperators corresponding to spin-adapted, triplet single excitation operators.

Return type
Generator[*FermionOperator*, None, None]

index(*orb*, *spin*)

Accepts spatial orbital index and spin to convert to a spin orbital index.

Parameters

- **orb** (int) – Spatial orbital index.
- **spin** (int) – Alpha spin (0) or beta spin (1).

Returns
int – The spin-orbital index.

classmethod load_h5(*name*)

Loads operator object from .h5 file.

Parameters
name (Union[str, Group]) – The filename to load from.

property n_orb: int

Number of spatial orbitals.

Return type
int

property n_spin_orb: int

Return the number of spin-orbitals.

Return type
int

operator_to_latex(*fermion_operator*, ***kwargs*)

Generate a LaTeX representation of an operator in this occupation space.

Operator indices will show degrees of freedom relevant to parent occupation space.

Parameters

- **fermion_operator** (Union[*FermionOperator*, *FermionOperatorString*]) – Operator to convert to latex representation.
- ****kwargs** – Keyword arguments passed to the `FermionOperatorString.to_latex()` method of component operator strings, `FermionOperatorString`.

Returns

LaTeX compilable equation string.

Examples

```
>>> from inquanto.spaces import FermionSpace
>>> fs = FermionSpace(10)
>>> fo = FermionOperator([(1.0, "F1 F1^"), (2+4j, "F2^ F0")])
>>> print(fs.operator_to_latex(fo))
a_{0\downarrow} a_{0\downarrow}^{\dagger} + (2.0+ 4.0\text{i}) a_{1\uparrow}^
↪{\dagger} a_{0\uparrow}
```

print_state (*fermion_state*, **state_masks*)

Prints a fermion state with quantum number information.

If the state_masks argument is provided, a column is added to the print for, with an X marking the elements of the mask which are True.

Parameters

- **fermion_state** (*FermionState*) – Fermion state to print.
- **state_masks** (List[bool]) – A mask to include in the print, as described above.

Examples

```
>>> from inquanto.spaces import FermionSpace
>>> space = FermionSpace(8)
>>> mask = space.generate_cyclic_window_mask(window=4, shift=2)
>>> state = space.generate_occupation_state_from_list([1, 1, 1, 1, 0, 0, 0,␣
↪0])
>>> space.print_state(state, mask)
 0 0a         :  1    .
 1 0b         :  1    .
 2 1a         :  1    X
 3 1b         :  1    X
 4 2a         :  0    X
 5 2b         :  0    X
 6 3a         :  0    .
 7 3b         :  0    .
```

Return type

None

quantum_label (*i*)

Generates a label for a given spin-orbital comprised of quantum numbers.

Parameters

i (int) – Spin orbital index.

Returns
str – Label storing all quantum numbers.

quantum_number(*i*)

Converts the spin orbital index to a tuple of quantum numbers.

These quantum numbers are spatial orbital and spin index corresponding to the input spin orbital index.

Parameters
i (int) – Spin orbital index.

Returns
Tuple[int, int] – Tuple containing spatial orbital and spin index.

quantum_number_orb(*i*)

Get spatial orbital index from a spin orbital index.

Parameters
i (int) – Spin orbital index.

Returns
int – The spatial orbital index.

quantum_number_spin(*i*)

Get spin index from an spin orbital index.

Parameters
i (int) – Spin orbital index.

Returns
int – Spin index, alpha spin is 0, and beta spin is 1.

save_h5(*name*)

Dumps operator object to .h5 file.

Parameters
name (Union[str, Group]) – The filename to save to.

select(*fermion_state=None*, *state_value=1*, *column=None*, *column_value=None*, *state_mask=None*)

Select elements from the underlying table which satisfy the input arguments.

For example, in a 2-electron, 4-spin orbital FermionSpace, the underlying table is [[0, 0], [0, 1], [1, 0], [1, 1]], where the first element of each list corresponds to a spin (0 for alpha, 1 for beta) and the second element corresponds to the underlying spatial orbital. So to select the table elements for a given spin, the arguments are column=0, column_value=0 for spin alpha, and column=0, column_value=1 for spin beta. To select elements of the table corresponding to a given spatial orbital x, the arguments are column=1 and column_value=x.

Parameters

- **fermion_state** (Optional[*FermionState*], default: None) – A FermionState object.
- **state_value** – State value.
- **column** (Optional[int], default: None) – The column in the underlying table to select from, 0 for spin, 1 for orbital.
- **column_value** (Optional[int], default: None) – The value of interest for the given column.
- **state_mask** (Optional[List[bool]], default: None) – A boolean mask for for the provided FermionState.

Yields
Elements of the underlying table which satisfy the input arguments.

Return type
Iterator[int]

symmetry_operators_z2(*spin_ordering='abab'*, *point_group=True*, *return_factorized=False*)

Construct symmetry operators corresponding to point group, spin- and particle-number parity Z2 symmetries.

Warning: Where return_factorized is set to False (the default behaviour!) this may blow up exponentially. Set return_factorized to True to avoid this problem by returning symmetry operators as SymmetryOperatorFermionicFactorised.

Parameters

- **spin_ordering** (Optional[str], default: "abab") – The spin-orbital ordering - "abab" for alpha-beta-alpha-beta or "aabb" for alpha-alpha-beta-beta. Pass None to skip spin parity symmetry entirely.
- **point_group** (bool, default: True) – Set to False to not return point group symmetries.
- **return_factorized** (bool, default: False) – Set to True to return symmetry operators as SymmetryOperatorFermionicFactorised. This avoids exponential growth in terms when expanding the symmetry operator, but returns as the less general SymmetryOperatorFermionicFactorised class.

Returns
List[Union[*SymmetryOperatorFermionic*, *SymmetryOperatorFermionicFactorised*]] – A list of the Z2 symmetry operators in format determined by the return_factorized argument.

symmetry_operators_z2_in_sector(*fermion_state*, *spin_ordering='abab'*, *point_group=True*, *return_factorized=False*)

Return a list of symmetry operators.

Symmetry operators returned correspond to the point group, spin parity and particle number parity Z2 symmetries which stabilize a desired symmetry sector.

Global phases on the symmetry operators here are such that the returned operators stabilize the provided fermion_state - i.e. if the symmetry operator has an expectation value of -1 with the provided fermion state, then the symmetry operator will be multiplied by -1. This is in contrast to symmetry_operators_z2, which returns operators without an additional global phase.

Warning: Where return_factorized is set to False (the default behaviour!) this may blow up exponentially. Set return_factorized to True to avoid this problem by returning symmetry operators as SymmetryOperatorFermionicFactorised.

Parameters

- **fermion_state** (*FermionState*) – A reference state required to define a symmetry sector.
- **spin_ordering** (Optional[str], default: "abab") – The spin-orbital ordering - "abab" for alpha-beta-alpha-beta or "aabb" for alpha-alpha-beta-beta. Pass None to skip spin parity symmetry entirely.

- **point_group** (bool, default: True) – Set to False to not return point group symmetries.
- **return_factorized** (bool, default: False) – Set to True to return symmetry operators as SymmetryOperatorFermionicFactorised. This avoids exponential growth in terms when expanding the symmetry operator, but returns as the less general SymmetryOperatorFermionicFactorised class.

Returns

List[Union[*SymmetryOperatorFermionic*, *SymmetryOperatorFermionicFactorised*]] – A list of the Z2 symmetry operators in format determined by the return_factorized argument.

class FermionSpaceBrillouin(*n_spin_orb*, *n_kp*, *conservation*)

Bases: OccupationSpace

FermionSpace for the momentum representation of a periodic system.

Parameters

- **n_spin_orb** (int) – The Number of spin orbitals.
- **n_kp** (int) – The Number of k-points.
- **conservation** (List[List[List[int]]]) – Momentum conservation table generated by PySCF.

COLUMN_KP = 0

Column of the underlying table corresponding to k-points.

COLUMN_ORB = 1

Column of the underlying table corresponding to orbitals.

COLUMN_SPIN = 2

Column of the underlying table corresponding to spin.

SPIN_ALPHA = 0

Integer representation of alpha spin.

SPIN_BETA = 1

Integer representation of beta spin.

SPIN_DOWN = 1

Integer representation of spin down.

SPIN_UP = 0

Integer representation of spin up.

construct_contraction_mask_from_operators(*operators*)

Constructs a contraction mask based on the occurring indices in the operators.

Parameters

operators (Union[*FermionOperator*, List[*FermionOperator*]]) – An operator or list of operators.

Returns

List[int] – The mask defining the contraction.

construct_number_operator(*kp*)

Creates and returns a FermionOperator representation of the number operator for a given k-point.

Parameters
kp (int) – The k-point index.

Returns
FermionOperator – The number operator for the provided k-point.

construct_one_body_operator_from_integral(*one_body_spatial*, *spins*, *kpoints*, *spatial_mask=None*)

Constructs a one-body operator from one-body spatial integral.

Parameters

- **one_body_spatial** (ndarray[Any, dtype[float]]) – Integrals in chemists notation.
- **spins** (Tuple[int, int]) – The spins, e.g. (0,0) = (SPIN_UP, SPIN_UP).
- **kpoints** (Tuple[int, int]) – The kpoints, e.g. (0,1) = (ki, kj).
- **spatial_mask** (Optional[List[bool]], default: None) – Mask filtering the generations of terms in the operator.

Returns
A one-body operator.

static construct_scalar_operator(*value*)

Construct a scalar FermionOperator object.

Parameters
value (float) – The coefficient in the scalar operator.

Returns
A FermionOperator corresponding to the provided value.

construct_two_body_operator_from_integral(*two_body_spatial*, *spins*, *kpoints*, *spatial_mask=None*)

Constructs a two-body operator from two-body spatial integral.

That is

$$\sum_{ijkl}(ij|kl)a^{\dagger}_{i,s_i,k_i}a^{\dagger}_{k,s_k,k_k}a_{l,s_l,k_l}a_{j,s_j,k_j} \tag{22.1}$$

where (s_i,s_j,s_k,s_l) = spins and (k_i,k_j,k_k,k_l) = kpoints.

Notes

The integrals are in chemists notation, that is two_body_spatial[i,j,k,l] = (ij|kl).

Parameters

- **two_body_spatial** (ndarray[Any, dtype[float]]) – Integrals in chemists notation.
- **spins** (Tuple[int]) – The corresponding spins as (s_i,s_j,s_k,s_l).
- **kpoints** (Tuple[int]) – The corresponding k-points as (k_i,k_j,k_k,k_l).
- **spatial_mask** (Optional[List[bool]], default: None) – A Mask filtering the generations of terms in the operator.

Returns
Two-body operator.

contract_occupation_space(*active_orbs*, *active_kps=None*)

Contract the occupation space with respect to the given active space.

Parameters

- **active_orbs** (list) – List of active spatial orbitals.
- **active_kps** (Optional[List[int]], default: None) – List of active k-points.

Returns
Tuple[*FermionSpaceBrillouin*, List[bool]] – Tuple containing the contracted occupation space object and the contraction mask.

static contract_occupation_state(*occupation_state*, *contraction_mask*)

Contracts fermion state according to a mask.

Parameters

- **occupation_state** (*FermionState*) – A FermionState object.
- **contraction_mask** (List[bool]) – The contraction mask.

Returns
ndarray[Any, dtype[int]] – An array corresponding to a contracted occupation number vector.

static contract_operator(*fermion_state*, *contraction_mask*, *operator*)

Contracts operator according to a mask and the fermion state.

That is, freezes spin-orbitals according to the known occupation numbers in fermion state.

Notes

This currently works only with FermionOperator.

Parameters

- **fermion_state** (*FermionState*) – Fermion state.
- **contraction_mask** (List[bool]) – Mask, where True, the spin-orbital is frozen.
- **operator** (Union[List[*FermionOperator*], *FermionOperator*]) – Operator to be contracted.

Returns
Union[List[*FermionOperator*], *FermionOperator*] – The contracted operator.

static contract_state_mask(*state_mask*, *contraction_mask*)

Contracts mask according to another mask.

Parameters

- **state_mask** – A state mask.
- **contraction_mask** (List[bool]) – Mask defining a contraction.

Returns
List[bool] – The contracted mask.

static convert_mask_to_index_map(*mask*)

Convert a mask into an index mapping.

Parameters

mask (Union[ndarray[Any, dtype[bool]], List[bool]]) – A boolean mask with the length of the fock space.

Returns

List[int] – A list that maps to the indices where the mask is True.

count(*fermion_state=None*, *state_value=1*, *column=None*, *column_value=None*, *state_mask=None*)

Counts spin orbitals or modes according to the relevant quantum number and state value.

This is a depreciated legacy method and will be removed in subsequent releases.

Notes

If some value is None, then it is counted.

Parameters

- **fermion_state** (Optional[*FermionState*], default: None) – A FermionState object.
- **state_value** (int, default: 1) – 1/0 occupied/unoccupied, respectively.
- **column** (Optional[int], default: None) – The column index in the quantum number table.
- **column_value** (Optional[int], default: None) – The value to be counted.
- **state_mask** (Optional[List[bool]], default: None) – State mask defining which spin orbitals or modes are included in the count.

Returns

int – Number of orbitals or modes satisfying the input arguments.

generate_occupation_state(*n_fermion=0*, *multiplicity=None*)

Generates a Fock state with specified number of fermions and multiplicity.

Parameters

- **n_fermion** (int, default: 0) – Number of fermions per k-point.
- **multiplicity** (Optional[int], default: None) – Multiplicity of the state per k-point.

Returns

FermionState – The generated fermion state.

generate_occupation_state_from_list(*fermion_state_list=None*)

Convert a list of occupations to an occupation state.

Parameters

fermion_state_list (Optional[List[int]], default: None) – A list of occupation numbers.

Returns

FermionState – The generated fermion state.

generate_occupation_state_from_spatial_occupation(*occupation*)

Generates a fermion state from spatial orbital occupations per k-point.

Parameters

occupation (List[List[int]]) – List of lists of occupations 0, 1, or 2. The number of length of the occupation argument should be the number of k-points.

Returns

FermionState – The generated fermionic state.

index(*kp*, *orb*, *spin*)

Converts the set of quantum numbers to an index.

Parameters

- **kp** (int) – The k-point index.
- **orb** (int) – Orbital index.
- **spin** (int) – Spin.

Returns

int – Index of the occupation state.

classmethod load_h5(*name*)

Loads operator object from .h5 file.

Parameters

name (Union[str, Group]) – The filename to load from.

property n_kp: int

Number of k-points.

Returns

int – Number of k-points.

property n_spin_orb: int

Return the number of spin-orbitals.

Return type

int

operator_to_latex(*fermion_operator*, ***kwargs*)

Generate a LaTeX representation of an operator in this occupation space.

Operator indices will show degrees of freedom relevant to parent occupation space.

Parameters

- **fermion_operator** (Union[*FermionOperator*, *FermionOperatorString*]) – Operator to convert to latex representation.
- ****kwargs** – Keyword arguments passed to the FermionOperatorString.to_latex() method of component operator strings, FermionOperatorString.

Returns

LaTeX compilable equation string.

Examples

```
>>> from inquanto.spaces import FermionSpace
>>> fs = FermionSpace(10)
>>> fo = FermionOperator([(1.0, "F1 F1^"), (2+4j, "F2^ F0")])
>>> print(fs.operator_to_latex(fo))
a_{0\downarrow} a_{0\downarrow}^{\dagger} + (2.0+ 4.0\text{i}) a_{1\uparrow}^
↪{\dagger} a_{0\uparrow}
```

print_state (*fermion_state*, **state_masks*)

Prints a fermion state with quantum number information.

If the state_masks argument is provided, a column is added to the print for, with an X marking the elements of the mask which are True.

Parameters

- **fermion_state** (*FermionState*) – Fermion state to print.
- **state_masks** (List[bool]) – A mask to include in the print, as described above.

Examples

```
>>> from inquanto.spaces import FermionSpace
>>> space = FermionSpace(8)
>>> mask = space.generate_cyclic_window_mask(window=4, shift=2)
>>> state = space.generate_occupation_state_from_list([1, 1, 1, 1, 0, 0, 0,␣
↪0])
>>> space.print_state(state, mask)
 0 0a          :  1    .
 1 0b          :  1    .
 2 1a          :  1    X
 3 1b          :  1    X
 4 2a          :  0    X
 5 2b          :  0    X
 6 3a          :  0    .
 7 3b          :  0    .
```

Return type
None

quantum_label (*i*)

Generates a label for a given basis state index, based on the quantum numbers.

Parameters
i (int) – Serial index.

Returns
str – A label for a given serial index.

quantum_number (*i*)

Converts the index to a tuple of quantum numbers.

Parameters
i (int) – Index.

Returns
Tuple[int, int, int] – Tuple containing the quantum number for k-points, the quantum number for spatial orbitals, and the quantum number for spins.

quantum_number_kp(*i*)

Get k-point quantum number from an index.

Parameters
i (int) – Serial index.

Returns
int – The k-point quantum number.

quantum_number_orb(*i*)

Get spatial orbital quantum number from an index.

Parameters
i (int) – Serial index.

Returns
int – Spatial orbital quantum number.

quantum_number_spin(*i*)

Get spin quantum number from an index.

Parameters
i (int) – Serial index.

Returns
qn_spins – Spin quantum number.

save_h5(*name*)

Dumps operator object to .h5 file.

Parameters
name (Union[str, Group]) – The filename to save to.

select(*fermion_state=None*, *state_value=1*, *column=None*, *column_value=None*, *state_mask=None*)

Select elements from the underlying table which satisfy the input arguments.

For example, in a 2-electron, 4-spin orbital FermionSpace, the underlying table is [[0, 0], [0, 1], [1, 0], [1, 1]], where the first element of each list corresponds to a spin (0 for alpha, 1 for beta) and the second element corresponds to the underlying spatial orbital. So to select the table elements for a given spin, the arguments are column=0, column_value=0 for spin alpha, and column=0, column_value=1 for spin beta. To select elements of the table corresponding to a given spatial orbital x, the arguments are column=1 and column_value=x.

Parameters

- **fermion_state** (Optional[*FermionState*], default: None) – A FermionState object.
- **state_value** – State value.
- **column** (Optional[int], default: None) – The column in the underlying table to select from, 0 for spin, 1 for orbital.
- **column_value** (Optional[int], default: None) – The value of interest for the given column.
- **state_mask** (Optional[List[bool]], default: None) – A boolean mask for for the provided FermionState.

Yields
Elements of the underlying table which satisfy the input arguments.

Return type
Iterator[int]

class FermionSpaceSupercell(*n_spin_orb*, *n_rp*)

Bases: OccupationSpace

Fermionic Fock Space with periodicity in real space.

Parameters

- **n_spin_orb** (int) – The Number of spatial orbitals.
- **n_rp** (int) – The number of regional blocks.

COLUMN_ORB = 1

The column in the underlying table which corresponds to orbitals.

COLUMN_RP = 0

The column in the underlying table which corresponds to regional blocks.

COLUMN_SPIN = 2

The column in the underlying table which corresponds to spin.

SPIN_ALPHA = 0

Integer representation of spin alpha.

SPIN_BETA = 1

Integer representation of spin beta.

SPIN_DOWN = 1

Integer representation of spin down.

SPIN_UP = 0

Integer representation of spin up.

check_translation_invariance(*operator*)

Checks the translational invariance of an operator according to this supercell.

Parameters
operator (*FermionOperator*) – A fermion operator.

Returns
Union[complex, float] – The norm of the difference between the operator and its translated version.

construct_contraction_mask_from_operators(*operators*)

Constructs a contraction mask based on the occurring indices in the operators.

Parameters
operators (Union[*FermionOperator*, List[*FermionOperator*]]) – An operator or list of operators.

Returns
List[int] – The mask defining the contraction.

construct_number_operator(*rp*, *spin=None*)

Creates and returns a FermionOperator representation of the number operator for a cell with spin.

Parameters

- **rp** (int) – Cell index.
- **spin** (Optional[int], default: None) – A 0, 1 or None. If None then the resulting number operator will be the sum of up and down.

Returns
FermionOperator – The number operator.

construct_one_body_operator_from_big_integral(*one_body_spatial*, *spins*, *spatial_mask=None*)

Constructs a one-body operator from one-body spatial integral for the supercell.

Parameters

- **one_body_spatial** (ndarray[Any, dtype[TypeVar(ScalarType, bound=generic, covariant=True)]]) – Integrals in chemists notation for the supercell.
- **spins** (Tuple[int, int]) – The spins, e.g. (0,0) = (SPIN_UP, SPIN_UP).
- **spatial_mask** – Mask filtering the generations of terms in the operator.

Returns
FermionOperator – The one-body operator.

construct_permutation_operator(*permutation*)

Construct a fermionic permutation operator from a permutation list.

The iterable of length two iterables must contain the two indices to be swapped.

Parameters
permutation (Union[ndarray[Any, dtype[int]], List[int]]) – Permutation list.

Returns
FermionOperator – The generated permutation operator.

Examples

```
>>> space = FermionSpaceSupercell(4, 1)
>>> print(space.construct_permutation_operator([1, 0]))
(1.0, ), (-1.0, F0^ F0 ), (-1.0, F1^ F1 ), (1.0, F0^ F1 ), (1.0, F1^ F0 )
```

---

construct_reverse_rp_permutation_operator()

Construct a permutation operator of the indices that reverses the order of the cells.

Returns
FermionOperator – The permutation operator.

static construct_scalar_operator(*value*)

Construct a FermionOperator with identity operation and the value input as a coefficient.

Parameters
value (float) – Coefficient of the scalar operator.

Returns
FermionOperator – A scalar fermionic operator.

construct_shift_rp_permutation_operator(*shift*)

Construct a permutation operator of the indices that translate cells by shift cell.

Parameters
shift (int) – Number of cells to be translated with.

Returns
FermionOperator – The permutation operator.

construct_swap_rp_permutation_operator(*rp1*, *rp2*)

Construct a permutation operator of the indices that swap two cells.

Parameters

- **rp1** (int) – Index of cell 1.
- **rp2** (int) – Index of cell 2.

Returns
FermionOperator – The permutation operator.

construct_two_body_operator_from_big_integral(*two_body_spatial*, *spins*, *spatial_mask=None*)

Constructs a two-body operator from two-body spatial integral for the supercell.

That is

$$\sum_{ijkl}(ij|kl)a^\dagger_{i,s_i}a^\dagger_{k,s_k}a_{l,s_l}a_{j,s_j}. \tag{22.2}$$

Where (s_i,s_j,s_k,s_l) are the spins.

Notes

The integrals are in chemists notation, that is two_body_spatial[i,j,k,l] = (ij|kl).

Parameters

- **two_body_spatial** (ndarray[Any, dtype[TypeVar(ScalarType, bound=generic, covariant=True)]]) – Integrals in chemists notation for the supercell.
- **spins** – The corresponding spins as (s_i,s_j,s_k,s_l).
- **spatial_mask** – Mask filtering the generations of terms in the operator.

Returns
FermionOperator – The two-body operator.

static contract_occupation_state(*occupation_state*, *contraction_mask*)

Contracts fermion state according to a mask.

Parameters

- **occupation_state** (*FermionState*) – A FermionState object.
- **contraction_mask** (List[bool]) – The contraction mask.

Returns
ndarray[Any, dtype[int]] – An array corresponding to a contracted occupation number vector.

static contract_operator(*fermion_state*, *contraction_mask*, *operator*)

Contracts operator according to a mask and the fermion state.

That is, freezes spin-orbitals according to the known occupation numbers in fermion state.

Notes

This currently works only with FermionOperator.

Parameters

- **fermion_state** (*FermionState*) – Fermion state.
- **contraction_mask** (List[bool]) – Mask, where True, the spin-orbital is frozen.
- **operator** (Union[List[*FermionOperator*], *FermionOperator*]) – Operator to be contracted.

Returns

Union[List[*FermionOperator*], *FermionOperator*] – The contracted operator.

static contract_state_mask(*state_mask*, *contraction_mask*)

Contracts mask according to another mask.

Parameters

- **state_mask** – A state mask.
- **contraction_mask** (List[bool]) – Mask defining a contraction.

Returns

List[bool] – The contracted mask.

static convert_mask_to_index_map(*mask*)

Convert a mask into an index mapping.

Parameters

mask (Union[ndarray[Any, dtype[bool]], List[bool]]) – A boolean mask with the length of the fock space.

Returns

List[int] – A list that maps to the indices where the mask is True.

count(*fermion_state=None*, *state_value=1*, *column=None*, *column_value=None*, *state_mask=None*)

Counts spin orbitals or modes according to the relevant quantum number and state value.

This is a depreciated legacy method and will be removed in subsequent releases.

Notes

If some value is None, then it is counted.

Parameters

- **fermion_state** (Optional[*FermionState*], default: None) – A FermionState object.
- **state_value** (int, default: 1) – 1/0 occupied/unoccupied, respectively.
- **column** (Optional[int], default: None) – The column index in the quantum number table.
- **column_value** (Optional[int], default: None) – The value to be counted.
- **state_mask** (Optional[List[bool]], default: None) – State mask defining which spin orbitals or modes are included in the count.

Returns

int – Number of orbitals or modes satisfying the input arguments.

generate_cyclic_masks(*window*, *shift=None*)

Generate a complete set of masks with windows shifted by window/2.

Parameters

- **window** (int) – Number of spin orbitals in the window.
- **shift** (Optional[int], default: None) – Shift along the state indices.

Returns

ndarray[Any, dtype[ndarray[Any, dtype[bool]]]] – A list of masks.

Examples

```
>>> fss = FermionSpaceSupercell(n_spin_orb=4, n_rp=2)
>>> fss.generate_cyclic_masks(window=2, shift=2)
array([[ True,  True, False, False, False, False, False, False],
       [False, False,  True,  True, False, False, False, False],
       [False, False, False, False,  True,  True, False, False],
       [False, False, False, False, False, False,  True,  True]])
```

---

generate_cyclic_window_mask(*window*, *shift=0*)

Generate a mask with a window and shift.

Parameters

- **window** (int) – Number of spin orbitals within the window.
- **shift** (int, default: 0) – Shift along the state indices.

Returns

ndarray[Any, dtype[bool]] – A list of booleans defining the mask.

Examples

```
>>> fss = FermionSpaceSupercell(n_spin_orb=4, n_rp=2)
>>> fss.generate_cyclic_window_mask(window=2, shift=2)
array([False, False,  True,  True, False, False, False, False])
```

---

generate_fock_state_from_list(*fock_state_list=None*)

Create a FermionState object corresponding to the list in the input.

Parameters

fock_state_list (Optional[List[int]], default: None) – A list representation of the desired occupation number vector.

Returns

FermionState – A FermionState object corresponding to the occupation number vector represented by fock_state_list.

generate_fock_state_from_spatial_big_occupation(*occupation*)

Generates a FermionState from spatial orbital occupations of the supercell.

Parameters
occupation (List[int]) – A list with values of {0,1,2} with the length of the number of spatial orbitals.

Returns
FermionState – The generated fermionic state.

generate_fock_state_from_spatial_occupation(*occupation*)

Generates a FermionState from list of spatial orbital occupations of each cell in the supercell.

Parameters
occupation (List[List[int]]) – A list of lists, each with values of {0,1,2} with the length of the number of spatial orbitals.

Returns
FermionState – The generated fermionic state.

generate_occupation_state(*n_fermion=0*, *multiplicity=None*)

Generates a fermion state with the specified number of fermions and multiplicity.

Parameters
- **n_fermion** (int, default: 0) – Number of fermions per regions.
- **multiplicity** (Optional[int], default: None) – Multiplicity of the state per regions.

Returns
FermionState – FermionState object representing the Fock state.

index(*rp*, *orb*, *spin*)

Get the index of the space corresponding to the input arguments.

Parameters
- **rp** (int) – Index of the regional block of interest.
- **orb** (int) – Index of the spatial orbital of interest.
- **spin** (int) – The spin of interest. 0 for alpha, 1 for beta.

Returns
int – The index of the space corresponding to the input arguments.

static is_operator_permutation_invariant(*operator*, *permutation*, *threshold=1e-8*)

Check if the operator is invariant under a permutation.

Parameters
- **operator** (*FermionOperator*) – A fermion operator.
- **permutation** (Union[ndarray[Any, dtype[int]], List[int]]) – A list with the permuted order.
- **threshold** (float, default: 1e-8) – Numerical threshold for invariance.

Returns
bool – True if the operator is permutationally invariant, else False.

classmethod load_h5(*name*)

Loads operator object from .h5 file.

Parameters
name (Union[str, Group]) – The filename to load from.

property n_rp: int

Return the number of regional blocks.

Return type
int

property n_spin_orb: int

Return the number of spin-orbitals.

Return type
int

operator_to_latex (*fermion_operator*, ***kwargs*)

Generate a LaTeX representation of an operator in this occupation space.

Operator indices will show degrees of freedom relevant to parent occupation space.

Parameters

- **fermion_operator** (Union[*FermionOperator*, *FermionOperatorString*]) – Operator to convert to latex representation.
- ****kwargs** – Keyword arguments passed to the `FermionOperatorString.to_latex()` method of component operator strings, `FermionOperatorString`.

Returns
LaTeX compilable equation string.

Examples

```
>>> from inquanto.spaces import FermionSpace
>>> fs = FermionSpace(10)
>>> fo = FermionOperator([(1.0, "F1 F1^"), (2+4j, "F2^ F0")])
>>> print(fs.operator_to_latex(fo))
a_{0\downarrow} a_{0\downarrow}^{\dagger} + (2.0+ 4.0\text{i}) a_{1\uparrow}^
↪{\dagger} a_{0\uparrow}
```

---

permutation (*swaps*)

Convert a list of swap pairs into a permutation list.

Parameters
swaps (List[Tuple[int, int]]) – List of index pairs to be swapped.

Returns
ndarray[Any, dtype[int]] – An array of indices which have been permuted according to the swaps argument.

permutation_matrix (*permutation*)

Convert a permutation list to a permutation matrix.

Parameters
permutation (Union[List[int], ndarray[Any, dtype[int]]]) – As a list of permuted indices.

Returns
ndarray[Any, dtype[float]] – The permutation matrix.

print_state(*fermion_state*, **state_masks*)

Prints a fermion state with quantum number information.

If the state_masks argument is provided, a column is added to the print for, with an X marking the elements of the mask which are True.

Parameters

- **fermion_state** (*FermionState*) – Fermion state to print.
- **state_masks** (List[bool]) – A mask to include in the print, as described above.

Examples

```
>>> from inquanto.spaces import FermionSpace
>>> space = FermionSpace(8)
>>> mask = space.generate_cyclic_window_mask(window=4, shift=2)
>>> state = space.generate_occupation_state_from_list([1, 1, 1, 1, 0, 0, 0,␣
↪0])
>>> space.print_state(state, mask)
 0 0a         :  1    .
 1 0b         :  1    .
 2 1a         :  1    X
 3 1b         :  1    X
 4 2a         :  0    X
 5 2b         :  0    X
 6 3a         :  0    .
 7 3b         :  0    .
```

Return type

None

quantum_label(*i*)

Get the quantum label corresponding to the element of the space indexed by the argument i.

Parameters

i (int) – The index of interest.

Returns

str – The r-point, spatial orbital and spin quantum numbers as a formatted string.

quantum_number(*i*)

Compute the quantum numbers of the ith index of the space.

Parameters

i (int) – The index of interest.

Returns

Tuple[int, int, int] – A tuple of ints corresponding to the regional block, spatial orbital and spin quantum numbers of the ith index of the space.

quantum_number_orb(*i*)

Compute the orbital quantum number for the ith index of the space.

Parameters

i (int) – The index of interest.

Returns

int – The orbital quantum number of the ith index of the space.

quantum_number_rp(*i*)

Compute the regional block quantum number for the ith element of the space.

Parameters
i (int) – The index of interest.

Returns
int – The regional block quantum number of the ith element of the space.

quantum_number_spin(*i*)

Compute the spin quantum number for the ith index of the space.

Parameters
i (int) – The index of interest.

Returns
int – The spin quantum number of the ith index of the space.

reverse_rp_permutation()

Construct a permutation list of the indices that reverses the order of the cells.

Returns
ndarray[Any, dtype[int]] – Permutation list.

save_h5(*name*)

Dumps operator object to .h5 file.

Parameters
name (Union[str, Group]) – The filename to save to.

select(*fermion_state=None*, *state_value=1*, *column=None*, *column_value=None*, *state_mask=None*)

Select elements from the underlying table which satisfy the input arguments.

For example, in a 2-electron, 4-spin orbital FermionSpace, the underlying table is [[0, 0], [0, 1], [1, 0], [1, 1]], where the first element of each list corresponds to a spin (0 for alpha, 1 for beta) and the second element corresponds to the underlying spatial orbital. So to select the table elements for a given spin, the arguments are column=0, column_value=0 for spin alpha, and column=0, column_value=1 for spin beta. To select elements of the table corresponding to a given spatial orbital x, the arguments are column=1 and column_value=x.

Parameters

- **fermion_state** (Optional[*FermionState*], default: None) – A FermionState object.
- **state_value** – State value.
- **column** (Optional[int], default: None) – The column in the underlying table to select from, 0 for spin, 1 for orbital.
- **column_value** (Optional[int], default: None) – The value of interest for the given column.
- **state_mask** (Optional[List[bool]], default: None) – A boolean mask for for the provided FermionState.

Yields
Elements of the underlying table which satisfy the input arguments.

Return type
Iterator[int]

shift_rp_permutation(*shift*)

Construct a permutation list of the indices that translate cells by a given number of cells.

Parameters

shift (int) – Number of cells to be translated with.

Returns

ndarray[Any, dtype[int]] – Permutation list.

swap_rp_permutation(*rp1*, *rp2*)

Construct a permutation list of the indices that swap two cells.

Parameters

- **rp1** (int) – Index of cell 1.
- **rp2** (int) – Index of cell 2.

Returns

ndarray[Any, dtype[int]] – Permutation list.

translate_operator(*operator*, *translation=None*)

Constructs a new operator that is translated by the cell translation function.

The default cell translation is R->R+1, so it shifts to the next indices. In 1D this is a simple translation along the axis.

Parameters

- **operator** (*FermionOperator*) – Fermion operator.
- **translation** – Cell translation function.

Returns

FermionOperator – The translated fermion operator.

class ParaFermionSpace(*n_spin_orb*, *point_group=None*, *orb_irreps=None*)

Bases: *QubitSpace*

A representation of the Qubit Hilbert space for ParaFermions. Contains related utilities.

Parameters

- **n_spin_orb** (int) – Number of spin orbitals.
- **point_group** (Union[*PointGroup*, str, None], default: None) – Point group of the molecule of interest.
- **orb_irreps** (Optional[List[str]], default: None) – Point group irreducible representations of the orbitals.

construct_double_qubit_excitation_operators()

Generate a list of all unique double qubit excitation operators T_{ijkl}.

The list of double qubit excitations is based on a given number of qubits. Fresh parameter symbols are generated for each unique excitation operator.

Warning: JW encoding of Hamiltonian is assumed: state of qubit i represents occupation of spin orbital i.

Returns

QubitOperatorList – A list of every unique double qubit excitation.

construct_imag_pauli_exponent_operators()

Constructs a QubitOperatorList containing QubitOperators with an even number of Pauli-Ys and a unit imaginary coefficient.

Each created QubitOperator in the QubitOperatorList is associated with a fresh coefficient symbol.

Returns
A QubitOperatorList containing QubitOperators with an even number of Pauli-Ys and a unit imaginary coefficient.

static construct_operator_from_string(*input*)

Construct a QubitOperator from the string provided as input.

Parameters
input (str) – Stringified QubitOperator.

Returns
QubitOperator – The generated qubit operator.

construct_real_pauli_exponent_operators()

Constructs a list of QubitOperators containing an odd number of Pauli-Ys and a coefficient of 1.

Each created QubitOperator in the QubitOperatorList is associated with a fresh coefficient symbol.

Returns
QubitOperatorList – A QubitOperatorList containing QubitOperators with an odd number of Pauli-Ys and a coefficient of 1.

static construct_scalar_operator(*value*)

Generates QubitOperator representing a scalar multiplier.

Parameters
value (Union[complex, float]) – A scalar multiplier.

Returns
An operator representing a scalar multiplier.

construct_single_qubit_excitation_operators()

Generate a list of all unique single qubit excitation operators T_ik.

The list of unique single qubit excitation operators is based on a given number of qubits. Each operator is equivalent to an exchange-interaction operation between two qubits i and k, T_ik = 0.5*([X_i Y_k] - [Y_i X_k]). Fresh parameter symbols are generated for each operator.

Warning: JW encoding of Hamiltonian is assumed: state of qubit i represents occupation of spin orbital i.

Returns
QubitOperatorList – A list of all unique single qubit excitation operators.

static count_spin(*qstate*)

Counts number of up and down spins in a QubitState (assuming JW mapping).

The Value returned should equal <S_z>. Will not distinguish open and closed shells.

Parameters
qstate (*QubitState*) – A qubit space object.

Returns
int – The sum of the spins.

index(*orb*, *spin*)

Converts the set of quantum numbers to an index.

Parameters

- **orb** (int) – The orbital quantum number of interest.
- **spin** (int) – The spin quantum number of interest.

Returns
int – The element of the space corresponding to the provided orbital and spin quantum numbers.

classmethod load_h5(*name*)

Loads operator object from .h5 file.

Parameters
name (Union[str, Group]) – The filename to load from.

n_ones(*fock_state*)

Count the number of ones in the given Fock state.

Returns
int – Integer count of the number of ones in the provided fock state.

property n_orb: int

Returns the number of spatial orbitals.

Return type
int

property n_spin_orb: int

Returns the number of spin-orbitals.

Return type
int

property paulis: Set[str]

Returns a set of python strings {“I”, “X”, “Y”, “Z”}.

Return type
Set[str]

quantum_label(*i*)

Generates a label for a given basis i.

Parameters
i (int) – Indexed element of the space to be labelled.

Returns
str – Quantum label of the element indexed by i as a string, with format {orbital quantum number}{spin (a or b)}.

quantum_number(*i*)

Converts the index to a tuple of quantum numbers.

Parameters
i (int) – Index.

Returns
Tuple[int, int] – Tuple containing the spatial orbital number and the spin quantum number of the index.

quantum_number_orb(*i*)

Get spatial orb from an index.

Parameters
i (int) – Index.

Returns
int – Spatial orbital quantum number.

quantum_number_spin(*i*)

Get the spin quantum number from an index.

Parameters
i (int) – Index.

Returns
int – The spin quantum number.

save_h5(*name*)

Dumps operator object to .h5 file.

Parameters
name (Union[str, Group]) – The filename to save to.

static symmetry_operators_z2(*operator*)

Given a QubitOperator, find an independent generating set for the Z2 symmetries of the operator.

The independent generating set is found by finding the kernel of the symplectic form of the operator. See arXiv 1701.08213 for more details.

Parameters
operator (*QubitOperator*) – A qubit operator to find the Z2 symmetries of.

Returns
List[*SymmetryOperatorPauli*] – A list of the Pauli representations of the Z2 symmetry operators.

static symmetry_operators_z2_in_sector(*operator*, *state*)

Find an independent generating set for the Z2 symmetries of the provided operator.

The generating set in the desired symmetry sectors is found by finding the kernel of the symplectic form of the operator (arXiv 1701.08213) and return operators which stabilize a desired symmetry sector.

Global phases on the symmetry operators here are such that the returned operators stabilize the provided qubit state - i.e. if the symmetry operator has an expectation value of -1 with the provided qubit state, then the found symmetry operator will be multiplied by -1. This is in contrast to symmetry_operators_z2, which returns operators without an additional global phase.

Parameters

- **operator** (*QubitOperator*) – A qubit operator to find the Z2 symmetries of.
- **state** (*QubitState*) – A state with which the returned symmetry operators will have an eigenvalue of +1.

Returns
List[*SymmetryOperatorPauli*] – A list of the Pauli representations of the Z2 symmetry operators which stabilize the desired symmetry sectors.

class QubitSpace(*n_qubits*)

Bases: object

2**N Dimensional Hilbert Space described in the basis of Pauli-vectors, where N is the number of qubits.

Parameters
n_qubits (int) – The number of qubits.

construct_imag_pauli_exponent_operators()

Constructs a QubitOperatorList containing QubitOperators with an even number of Pauli-Ys and a unit imaginary coefficient.

Each created QubitOperator in the QubitOperatorList is associated with a fresh coefficient symbol.

Returns
A QubitOperatorList containing QubitOperators with an even number of Pauli-Ys and a unit imaginary coefficient.

static construct_operator_from_string(*input*)

Construct a QubitOperator from the string provided as input.

Parameters
input (str) – Stringified QubitOperator.

Returns
QubitOperator – The generated qubit operator.

construct_real_pauli_exponent_operators()

Constructs a list of QubitOperators containing an odd number of Pauli-Ys and a coefficient of 1.

Each created QubitOperator in the QubitOperatorList is associated with a fresh coefficient symbol.

Returns
QubitOperatorList – A QubitOperatorList containing QubitOperators with an odd number of Pauli-Ys and a coefficient of 1.

classmethod load_h5(*name*)

Loads operator object from .h5 file.

Parameters
name (Union[str, Group]) – The filename to load from.

property paulis: Set[str]

Returns a set of python strings {“I”, “X”, “Y”, “Z”}.

Return type
Set[str]

save_h5(*name*)

Dumps operator object to .h5 file.

Parameters
name (Union[str, Group]) – The filename to save to.

static symmetry_operators_z2(*operator*)

Given a QubitOperator, find an independent generating set for the Z2 symmetries of the operator.

The independent generating set is found by finding the kernel of the symplectic form of the operator. See arXiv 1701.08213 for more details.

Parameters
operator (*QubitOperator*) – A qubit operator to find the Z2 symmetries of.

Returns

List[*SymmetryOperatorPauli*] – A list of the Pauli representations of the Z2 symmetry operators.

static symmetry_operators_z2_in_sector(*operator*, *state*)

Find an independent generating set for the Z2 symmetries of the provided operator.

The generating set in the desired symmetry sectors is found by finding the kernel of the symplectic form of the operator (arXiv 1701.08213) and return operators which stabilize a desired symmetry sector.

Global phases on the symmetry operators here are such that the returned operators stabilize the provided qubit state - i.e. if the symmetry operator has an expectation value of -1 with the provided qubit state, then the found symmetry operator will be multiplied by -1. This is in contrast to symmetry_operators_z2, which returns operators without an additional global phase.

Parameters

- **operator** (*QubitOperator*) – A qubit operator to find the Z2 symmetries of.
- **state** (*QubitState*) – A state with which the returned symmetry operators will have an eigenvalue of +1.

Returns

List[*SymmetryOperatorPauli*] – A list of the Pauli representations of the Z2 symmetry operators which stabilize the desired symmetry sectors.

chain_filters(**functions*)

Combine the callable fermionic excitation filters into one function.

Parameters

functions – Callable functions. Each of which returns a filtered list of fermionic excitations.

Returns

Callable[[Generator[*FermionOperatorString*, None, None]], Generator[*FermionOperatorString*, None, None]] – A fermionic excitation filter accounting for all filters provided in the *functions argument.

22.13 inquanto.states

InQuanto representation of quantum states.

class FermionState(*data=None*, *coeff=1.0*)

Bases: LinearDictCombiner

Handles a linear combination of fermionic reference states.

State is stored as a dict, with the keys being FermionStateString objects (configurations) and values being the corresponding configuration coefficients. The FermionState is initialised from a list of integers or a FermionStateString and a coefficient, a tuple of tuples, containing a coefficient and a FermionStateString object each, or a dict, with FermionStateString objects as keys and coefficients as values.

Examples

```
>>> fs0 = FermionState([1, 1, 0, 0], 1)
>>> print(fs0)
(1, [1, 1, 0, 0])
>>> fss = FermionStateString([1, 1, 0, 0])
```

(continues on next page)

(continued from previous page)

```
>>> fs1 = FermionState(fss, 1)
>>> print(fs1)
(1, [1, 1, 0, 0])
>>> fss0 = FermionStateString([1, 1, 0, 0])
>>> fss1 = FermionStateString([1, 0, 1, 0])
>>> fs2 = FermionState(((0.9, fss0), (0.1, fss1)))
>>> print(fs2)
(0.9, [1, 1, 0, 0]), (0.1, [1, 0, 1, 0])
>>> fss0 = FermionStateString([1, 1, 0, 0])
>>> fss1 = FermionStateString([1, 0, 1, 0])
>>> fs3 = FermionState({fss0: 0.9, fss1: 0.1})
>>> print(fs3)
(0.9, [1, 1, 0, 0]), (0.1, [1, 0, 1, 0])
```

Parameters

- **data** (Union[List[int], *FermionStateString*, Tuple[Tuple[Union[int, float, complex, Expr], *FermionStateString*]], Dict[*FermionStateString*, Union[int, float, complex, Expr]], None], default: None) –
- **coeff** (Union[int, float, complex, Expr], default: 1.0) –

approx_equal_to(*other*, *abs_tol=1e-10*)

Checks if object's dictionary values are numerically identical to the other object values.

Parameters

- **other** (LinearDictCombiner) – Object to compare to.
- **abs_tol** (float, default: 1e-10) – Threshold of comparing numeric values.

Raises

TypeError – When comparison of two values can't be done due to types mismatch.

Return type

bool

approx_equal_to_by_random_subs(*other*, *order=1*, *abs_tol=1e-10*)

Checks if object's dictionary values are numerically identical to the other object values.

Symbols contained in the difference of the two objects, if any, are substituted by random numeric values prior to norm check.

Parameters

- **other** (LinearDictCombiner) – Object to compare to.
- **order** (int, default: 1) – Parameter specifying the norm formula (see numpy.linalg.norm documentation).
- **abs_tol** (float, default: 1e-10) – Threshold vs which the norm of the difference is checked.

Return type

bool

clone()

Performs shallow copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

property coefficients: List[int | float | complex | Expr]

Returns dictionary values.

Return type
List[Union[int, float, complex, Expr]]

compress(*abs_tol=1e-10*, *symbol_sub_type=CompressSymbolSubType.NONE*)

Combines duplicate terms, removing those with negligible coefficient.

Parameters

- **abs_tol** (float, default: 1e-10) – Tolerance for comparing values to zero.
- **symbol_sub_type** (Union[CompressSymbolSubType, str], default: CompressSymbolSubType.NONE) – Defines the behaviour for dealing with symbolic expressions in coefficients. If "none", symbolic expressions are left intact. If "unity", substitutes all free symbols with 1, and removes all imaginary and real components below tolerance. If "random", substitutes all free symbols with a random number between 0 and 1, and removes imaginary and real components below tolerance.

Warning: When :code: *symbol_sub_type != "none"*, this method assumes significant expression structure is known a priori, and is best suited to operators which have simple product expressions, such as excitation operators for VQE ansätze and digital quantum simulation. Otherwise, it may remove terms relevant to computation. Each expression is of the form $f(a_1, a_2, \ldots, a_n)$ for some symbols a_i. $|f(a_1, a_2, \ldots, a_n)|$ is assumed to monotonically increase in both real and imaginary components for all $a_i \in [0, 1]$.

Returns
LinearDictCombiner – Updated instance of LinearDictCombiner.

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

df()

Returns a Pandas DataFrame object of the dictionary.

Return type
DataFrame

empty()

Checks if dictionary is empty.

Return type
bool

evalf(**args*, ***kwargs*)

Evaluates symbolic expressions stored in dict values and replaces them with the results.

Parameters

- **args** (Any) – Args to be passed to *sympy.evalf()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.evalf()*.

Returns
`LinearDictCombiner` – Updated instance of LinearDictCombiner.

free_symbols()
Returns the free symbols in the coefficient values.

free_symbols_ordered()
Returns the free symbols in the dict, ordered alphabetically.

Returns
SymbolSet – Ordered set of symbols.

classmethod from_string(*input_string*)
Constructs a child class instance from a string.

Parameters
input_string (str) – String in the format `coeff1 [(coeff1_1, term1_1), ..., (coeff1_n, term1_n)], ..., coeffn [(coeffn_1, termn_1), ...]`.

Returns
Child class object.

is_all_coeff_complex()
Check if all coefficients are complex values.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type
bool

is_all_coeff_imag()
Check if all coefficients are complex values.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type
bool

is_all_coeff_real()
Check if all coefficients are real values.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type
bool

is_all_coeff_symbolic()
Check if all coefficients contain free symbols.

Return type
bool

is_any_coeff_complex()

Check if any coefficient is a complex value.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type
bool

is_any_coeff_imag()

Check if any coefficient is an imaginary value.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type
bool

is_any_coeff_real()

Check if any coefficient is a real value.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type
bool

is_any_coeff_symbolic()

Check if any coefficient contains a free symbol.

Return type
bool

is_normalized(*ndigits=14*)

Checks if the FermionState is normalized to 1 up to a given precision.

Parameters
ndigits (int, default: 14) – number of digits to round sum of squares of coefficients.

Returns
bool – True if FermionState is normalized up to given precision, else False.

is_parallel_with(*other*, *abs_tol=1e-10*)

Returns True if other is parallel with this (i.e. a scalar multiple of this), otherwise False.

Parameters

- **other** (LinearDictCombiner) – The other object to compare against
- **abs_tol** (Optional[float], default: 1e-10) – Tolerance threshold for comparison. Set to None to test for exact equivalence.

Returns
bool – True if other is parallel with this, otherwise False.

is_pure()

Checks if only one configuration is stored in state object.

Return type
bool

is_unit_1norm(*abs_tol=1e-10*)

Returns True if operator has unit 1-norm, else False.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold for comparison with unity.

Return type
bool

is_unit_2norm(*abs_tol=1e-10*)

Returns True if operator has unit 1-norm, else False.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold for comparison with unity.

Return type
bool

is_unit_norm(*order=2*, *abs_tol=1e-10*)

Returns True if operator has unit p-norm, else False.

Parameters

- **order** (int, default: 2) – Norm order.
- **abs_tol** (float, default: 1e-10) – Tolerance threshold for comparison with unity.

Raises
ValueError – Coefficients contain free symbols.

Return type
bool

items()

Returns dictionary items.

Return type
ItemsView[Any, Union[int, float, complex, Expr]]

static key_from_str(*key_str*)

Returns a FermionStateString instance initiated from the input string.

Parameters
key_str (str) –

Return type
FermionStateString

make_hashable()

Return a hashable representation of the object.

Currently simply returns a stringified object.

Return type
str

map(*mapping*)

Updates dictionary values, using a mapping function provided.

Parameters
mapping (Callable[[Union[int, float, complex, Expr]], Union[int, float, complex, Expr]]) – Mapping function to update the dict.

Returns
LinearDictCombiner – None.

property n_symbols: int

Returns the number of free symbols in the object.

Return type
int

norm_coefficients(*order=2*)

Returns the p-norm of the coefficients.

Parameters
order (int, default: 2) – Norm order.

Return type
Union[complex, float]

normalized(*norm_value=1.0*, *norm_order=2*)

Returns a copy of this object with normalised coefficients.

Parameters

- **norm_value** (float, default: 1.0) – The desired norm of the returned operator.
- **norm_order** (int, default: 2) – The order of the norm to be used.

Returns
LinearDictCombiner – A copy of the object with coefficients normalised to the desired value.

print_table()

Print dictionary formatted as a table.

Return type
NoReturn

remove_global_phase(*phase=0.0*)

Returns a copy with a global phase applied such that the first element has the desired phase.

Parameters
phase (float, default: 0.0) – The phase to yield on the first element, in half-turns (i.e. multiples of pi).

Returns
LinearDictCombiner – A copy of the object with the desired global phase applied.

reversed_order()

Reverses the order of terms and returns it as a new object.

Return type
LinearDictCombiner

simplify (**args*, ***kwargs*)

Simplifies expressions stored in dictionary values.

Parameters

- **args** (Any) – Args to be passed to *sympy.simplify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.simplify()*.

Returns

LinearDictCombiner – Updated instance of LinearDictCombiner.

property single_term

Returns the stored configuration as a list, if only a single configuration is stored.

Raises

RuntimeError – if more than one configuration is stored

Returns

the single configuration occupation vector

split ()

Generates FermionState objects containing only single configurations, preserving coefficients.

subs (*symbol_map*)

Returns a new objects with symbols substituted.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type

TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution (*symbol_map=None*)

Substitutes free symbols for numerical values according to a map.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) –

Return type

LinearDictCombiner

sympify (**args*, ***kwargs*)

Sympifies dictionary values.

Replaces values with their corresponding symbolic expressions.

Parameters

- **args** (Any) – Args to be passed to *sympy.sympify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.sympify()*.

Returns

LinearDictCombiner – Updated instance of LinearDictCombiner.

Raises

RuntimeError – When sympification fails.

property terms: List[Any]

Returns dictionary keys.

Return type
List[Any]

unsympify()

Unsympifies dicionary values.

Replaces symbolic expressions with their corresponding numeric values.

Returns
LinearDictCombiner – Updated instance of LinearDictCombiner.

Raises
TypeError – When unsympification fails.

vdot(*other*)

Calculates dot product with the other FermionState object.

The other FermionState is treated as the ket, this one is treated as the bra.

Parameters
other (*FermionState*) – another fermionic state forming the ket of the inner product

Returns
Union[int, float, complex, Expr] – the inner product between self and other

classmethod zero()

Return object with a zero dict entry.

Examples

```
>>> print(LinearDictCombiner.zero())
(0)
```

---

class FermionStateString(*initializer: bytes | int*)

Bases: bytes

Handles a single fermionic reference state.

Is intrinsically a byte string.

capitalize() → copy of B

Return a copy of B with only its first character capitalized (ASCII) and the rest lower-cased.

center(*width*, *fillchar=b' '*, */*)

Return a centered string of length width.

Padding is done using the specified fill character.

count(*sub*[, *start*[, *end*]]) → int

Return the number of non-overlapping occurrences of subsection sub in bytes B[start:end]. Optional arguments start and end are interpreted as in slice notation.

decode(*encoding='utf-8'*, *errors='strict'*)

Decode the bytes using the codec registered for encoding.

encoding
The encoding with which to decode the bytes.

errors
The error handling scheme to use for the handling of decoding errors. The default is 'strict' meaning that decoding errors raise a UnicodeDecodeError. Other possible values are 'ignore' and 'replace' as well as any other name registered with codecs.register_error that can handle UnicodeDecodeErrors.

endswith (*suffix*[, *start*[, *end*]]) → bool

Return True if B ends with the specified suffix, False otherwise. With optional start, test B beginning at that position. With optional end, stop comparing B at that position. suffix can also be a tuple of bytes to try.

expandtabs (*tabsize=8*)

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find (*sub*[, *start*[, *end*]]) → int

Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

fromhex ()

Create a bytes object from a string of hexadecimal numbers.

Spaces between two numbers are accepted. Example: bytes.fromhex('B9 01EF') -> b'\xb9\x01\xef'.

hex ()

Create a str of hexadecimal numbers from a bytes object.

sep
An optional single character or byte to separate hex bytes.

bytes_per_sep
How many bytes between separators. Positive values count from the right, negative values count from the left.

Example: >>> value = b'xb9x01xef' >>> value.hex() 'b901ef' >>> value.hex(':') 'b9:01:ef' >>> value.hex(':', 2) 'b9:01ef' >>> value.hex(':', -2) 'b901:ef'

index (*sub*[, *start*[, *end*]]) → int

Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the subsection is not found.

isalnum () → bool

Return True if all characters in B are alphanumeric and there is at least one character in B, False otherwise.

isalpha () → bool

Return True if all characters in B are alphabetic and there is at least one character in B, False otherwise.

isascii () → bool

Return True if B is empty or all characters in B are ASCII, False otherwise.

isdigit () → bool

Return True if all characters in B are digits and there is at least one character in B, False otherwise.

islower () → bool

Return True if all cased characters in B are lowercase and there is at least one cased character in B, False otherwise.

isspace() → bool

Return True if all characters in B are whitespace and there is at least one character in B, False otherwise.

istitle() → bool

Return True if B is a titlecased string and there is at least one character in B, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

isupper() → bool

Return True if all cased characters in B are uppercase and there is at least one cased character in B, False otherwise.

join(*iterable_of_bytes*, /)

Concatenate any number of bytes objects.

The bytes whose method is called is inserted in between each pair.

The result is returned as a new bytes object.

Example: b'.'.join([b'ab', b'pq', b'rs']) -> b'ab.pq.rs'.

ljust(*width*, *fillchar=b' '*, /)

Return a left-justified string of length width.

Padding is done using the specified fill character.

lower() → copy of B

Return a copy of B with all ASCII characters converted to lowercase.

lstrip(*bytes=None*, /)

Strip leading bytes contained in the argument.

If the argument is omitted or None, strip leading ASCII whitespace.

static maketrans(*frm*, *to*, /)

Return a translation table useable for the bytes or bytearray translate method.

The returned table will be one where each byte in frm is mapped to the byte at the same position in to.

The bytes objects frm and to must be of the same length.

partition(*sep*, /)

Partition the bytes into three parts using the given separator.

This will search for the separator sep in the bytes. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original bytes object and two empty bytes objects.

removeprefix(*prefix*, /)

Return a bytes object with the given prefix string removed if present.

If the bytes starts with the prefix string, return bytes[len(prefix):]. Otherwise, return a copy of the original bytes.

removesuffix(*suffix*, /)

Return a bytes object with the given suffix string removed if present.

If the bytes ends with the suffix string and that suffix is not empty, return bytes[:-len(prefix)]. Otherwise, return a copy of the original bytes.

replace(*old*, *new*, *count=-1*, */*)

Return a copy with all occurrences of substring old replaced by new.

count
Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind(*sub*[, *start*[, *end*]]) → int

Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(*sub*[, *start*[, *end*]]) → int

Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Raise ValueError when the subsection is not found.

rjust(*width*, *fillchar=b' '*, */*)

Return a right-justified string of length width.

Padding is done using the specified fill character.

rpartition(*sep*, */*)

Partition the bytes into three parts using the given separator.

This will search for the separator sep in the bytes, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty bytes objects and the original bytes object.

rsplit(*sep=None*, *maxsplit=-1*)

Return a list of the sections in the bytes, using sep as the delimiter.

sep
The delimiter according which to split the bytes. None (the default value) means split on ASCII whitespace characters (space, tab, return, newline, formfeed, vertical tab).

maxsplit
Maximum number of splits to do. -1 (the default value) means no limit.

Splitting is done starting at the end of the bytes and working to the front.

rstrip(*bytes=None*, */*)

Strip trailing bytes contained in the argument.

If the argument is omitted or None, strip trailing ASCII whitespace.

split(*sep=None*, *maxsplit=-1*)

Return a list of the sections in the bytes, using sep as the delimiter.

sep
The delimiter according which to split the bytes. None (the default value) means split on ASCII whitespace characters (space, tab, return, newline, formfeed, vertical tab).

maxsplit
Maximum number of splits to do. -1 (the default value) means no limit.

splitlines(*keepends=False*)

Return a list of the lines in the bytes, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

startswith(*prefix*[, *start*[, *end*]]) → bool

Return True if B starts with the specified prefix, False otherwise. With optional start, test B beginning at that position. With optional end, stop comparing B at that position. prefix can also be a tuple of bytes to try.

strip(*bytes=None*, */*)

Strip leading and trailing bytes contained in the argument.

If the argument is omitted or None, strip leading and trailing ASCII whitespace.

swapcase() → copy of B

Return a copy of B with uppercase ASCII characters converted to lowercase ASCII and vice versa.

title() → copy of B

Return a titlecased version of B, i.e. ASCII words start with uppercase characters, all remaining cased characters have lowercase.

translate(*table*, */*, *delete=b''*)

Return a copy with each character mapped by the given translation table.

table
Translation table, which must be a bytes object of length 256.

All characters occurring in the optional argument delete are removed. The remaining characters are mapped through the given translation table.

upper() → copy of B

Return a copy of B with all ASCII characters converted to uppercase.

zfill(*width*, */*)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The original string is never truncated.

class QubitState(*data=None*, *coeff=1.0*)

Bases: `LinearDictCombiner`, `Representable`

Handles a linear combination of reference states acting on qubits.

State is stored as a dict, with the keys being QubitStateString objects (configurations) and values being the corresponding configuration coefficients.

May be constructed from a list of integers or a QubitStateString and a coefficient, a tuple of tuples, containing a coefficient and a QubitStateString object each, or a dict, with QubitStateString objects as keys and coefficients as values.

Examples

```
>>> qs0 = QubitState([1, 1, 0, 0], 1)
>>> print(qs0)
(1, [1, 1, 0, 0])
>>> qss = QubitStateString([1,1,0,0])
>>> qs1 = QubitState(qss, 1)
>>> print(qs1)
(1, [1, 1, 0, 0])
```

(continues on next page)

(continued from previous page)

```
>>> qss0 = QubitStateString([1,1,0,0])
>>> qss1 = QubitStateString([1,0,1,0])
>>> qs2 = QubitState(((0.9, qss0), (0.1, qss1)))
>>> print(qs2)
(0.9, [1, 1, 0, 0]), (0.1, [1, 0, 1, 0])
>>> qss0 = QubitStateString([1,1,0,0])
>>> qss1 = QubitStateString([1,0,1,0])
>>> qs3 = QubitState({qss0: 0.9, qss1: 0.1})
>>> print(qs3)
(0.9, [1, 1, 0, 0]), (0.1, [1, 0, 1, 0])
```

Parameters

- **data** (Union[List[int], *QubitStateString*, Tuple[Tuple[Union[int, float, complex, Expr], *QubitStateString*]], Dict[*QubitStateString*, Union[int, float, complex, Expr]], None], default: None) –
- **coeff** (Union[int, float, complex, Expr], default: 1.0) –

approx_equal_to(*other*, *abs_tol=1e-10*)

Checks if object's dictionary values are numerically identical to the other object values.

Parameters

- **other** (LinearDictCombiner) – Object to compare to.
- **abs_tol** (float, default: 1e-10) – Threshold of comparing numeric values.

Raises

TypeError – When comparison of two values can't be done due to types mismatch.

Return type

bool

approx_equal_to_by_random_subs(*other*, *order=1*, *abs_tol=1e-10*)

Checks if object's dictionary values are numerically identical to the other object values.

Symbols contained in the difference of the two objects, if any, are substituted by random numeric values prior to norm check.

Parameters

- **other** (LinearDictCombiner) – Object to compare to.
- **order** (int, default: 1) – Parameter specifying the norm formula (see numpy.linalg.norm documentation).
- **abs_tol** (float, default: 1e-10) – Threshold vs which the norm of the difference is checked.

Return type

bool

clone()

Performs shallow copy of the object.

Return type

TypeVar(SYMBOLICTYPE, bound= Symbolic)

property coefficients: List[int | float | complex | Expr]

Returns dictionary values.

Return type
List[Union[int, float, complex, Expr]]

compress(*abs_tol=1e-10*, *symbol_sub_type=CompressSymbolSubType.NONE*)

Combines duplicate terms, removing those with negligible coefficient.

Parameters

- **abs_tol** (float, default: 1e-10) – Tolerance for comparing values to zero.
- **symbol_sub_type** (Union[CompressSymbolSubType, str], default: CompressSymbolSubType.NONE) – Defines the behaviour for dealing with symbolic expressions in coefficients. If "none", symbolic expressions are left intact. If "unity", substitutes all free symbols with 1, and removes all imaginary and real components below tolerance. If "random", substitutes all free symbols with a random number between 0 and 1, and removes imaginary and real components below tolerance.

Warning: When :code: *symbol_sub_type != "none"*, this method assumes significant expression structure is known a priori, and is best suited to operators which have simple product expressions, such as excitation operators for VQE ansätze and digital quantum simulation. Otherwise, it may remove terms relevant to computation. Each expression is of the form $f(a_1, a_2, \ldots, a_n)$ for some symbols a_i. $|f(a_1, a_2, \ldots, a_n)|$ is assumed to monotonically increase in both real and imaginary components for all $a_i \in [0, 1]$.

Returns
LinearDictCombiner – Updated instance of LinearDictCombiner.

copy()

Performs deep copy of the object.

Return type
TypeVar(SYMBOLICTYPE, bound= Symbolic)

df()

Returns a Pandas DataFrame object of the dictionary.

Return type
DataFrame

empty()

Checks if dictionary is empty.

Return type
bool

evalf(**args*, ***kwargs*)

Evaluates symbolic expressions stored in dict values and replaces them with the results.

Parameters

- **args** (Any) – Args to be passed to *sympy.evalf()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.evalf()*.

Returns
LinearDictCombiner – Updated instance of LinearDictCombiner.

free_symbols()

Returns the free symbols in the coefficient values.

free_symbols_ordered()

Returns the free symbols in the dict, ordered alphabetically.

Returns

SymbolSet – Ordered set of symbols.

classmethod from_ndarray(*statevector*, *bitorder='big'*)

Convert a numpy ndarray to an InQuanto QubitState.

This method will take a state vector as a ndarray column vector and convert it to a QubitState. In keeping with numpy convention, 1-D arrays and 2-D arrays with shape (N,1) are treated as column vectors. The length of the vector must be a power of 2 or 0. Differently shaped arrays are not accepted.

Parameters

- **statevector** (ndarray) – The statevector to be converted, of shape described above.
- **bitorder** – The endianness of the input ndarray relative to the desired InQuanto QubitState. Options are "little" and "big".

Returns

QubitState – An InQuanto QubitState representing the provided state vector.

Raises

ValueError – If the provided ndarray cannot be interpreted as a column vector as described above.

Example

```
>>> state_array = numpy.array([[0.],[0.],[1.],[0.]])
>>> state = QubitState.from_ndarray(state_array,bitorder="big")
>>> print(state)
(1.0, [1, 0])
>>> state_array = numpy.array([[0.],[0.],[1.],[0.]])
>>> state = QubitState.from_ndarray(state_array,bitorder="little")
>>> print(state)
(1.0, [0, 1])
```

---

classmethod from_string(*input_string*)

Constructs a child class instance from a string.

Parameters

input_string (str) – String in the format `coeff1 [(coeff1_1, term1_1), ..., (coeff1_n, term1_n)], ..., coeffn [(coeffn_1, termn_1), ...]`.

Returns

Child class object.

get_numeric_representation(*symbol_map=None*, ***, *space=None*, *backend=None*, *dtype=complex*)

Constructs a single numeric matrix/vector representation.

Note: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object. Currently this is not used, and the index of the representation is encoded with big endian.
- **backend** (Optional[Backend], default: None) – An optional backend to use to build the representation.
- **dtype** (Union[dtype[Any], None, Type[Any], _SupportsDType[dtype[Any]], str, Tuple[Any, int], Tuple[Any, Union[SupportsIndex, Sequence[SupportsIndex]]], List[Any], _DTypeDict, Tuple[Any, Any]], default: complex) – Specifies what dtype the return array should be converted.

Returns

Union[_SupportsArray[dtype], _NestedSequence[_SupportsArray[dtype]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]] – A matrix/vector representing the object.

get_symbolic_representation(*symbol_map=None*, *, *space=None*)

Constructs a single symbolic matrix/vector representation.

Note: This is an exponentially exploding method!

Parameters

- **symbol_map** (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) – A symbol substitution map before constructing the representation.
- **space** (Optional[Any], default: None) – Basis information to represent the object. Currently this is not used, and the index of the representation is encoded with big endian.

Returns

Expr – A symbolic expression as a representation, which is a symbolic NDArray.

is_all_coeff_complex()

Check if all coefficients are complex values.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type

bool

is_all_coeff_imag()

Check if all coefficients are complex values.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type

bool

is_all_coeff_real()

Check if all coefficients are real values.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type
bool

is_all_coeff_symbolic()

Check if all coefficients contain free symbols.

Return type
bool

is_any_coeff_complex()

Check if any coefficient is a complex value.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type
bool

is_any_coeff_imag()

Check if any coefficient is an imaginary value.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type
bool

is_any_coeff_real()

Check if any coefficient is a real value.

> **Warning:** Returns None if there is a free symbol in a coefficient.

Return type
bool

is_any_coeff_symbolic()

Check if any coefficient contains a free symbol.

Return type
bool

is_basis_state()

Returns True if state is multiple of a single computational basis state, else False.

Return type
bool

is_normalized(*ndigits=14*)

Checks if the QubitState is normalized to 1 up to a given precision.

Parameters
ndigits – number of digits to round sum of squares of coefficients

Returns
bool – True if state is normalised up to the given precision, else False.

is_parallel_with(*other*, *abs_tol=1e-10*)

Returns True if other is parallel with this (i.e. a scalar multiple of this), otherwise False.

Parameters

- **other** (LinearDictCombiner) – The other object to compare against
- **abs_tol** (Optional[float], default: 1e-10) – Tolerance threshold for comparison. Set to None to test for exact equivalence.

Returns
bool – True if other is parallel with this, otherwise False.

is_unit_1norm(*abs_tol=1e-10*)

Returns True if operator has unit 1-norm, else False.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold for comparison with unity.

Return type
bool

is_unit_2norm(*abs_tol=1e-10*)

Returns True if operator has unit 1-norm, else False.

Parameters
abs_tol (float, default: 1e-10) – Tolerance threshold for comparison with unity.

Return type
bool

is_unit_norm(*order=2*, *abs_tol=1e-10*)

Returns True if operator has unit p-norm, else False.

Parameters

- **order** (int, default: 2) – Norm order.
- **abs_tol** (float, default: 1e-10) – Tolerance threshold for comparison with unity.

Raises
ValueError – Coefficients contain free symbols.

Return type
bool

items()

Returns dictionary items.

Return type
ItemsView[Any, Union[int, float, complex, Expr]]

static key_from_str(*key_str*)

Generates a key from a string.

Parameters
key_str (str) –

Return type
QubitStateString

make_hashable()

Return a hashable representation of the object.

Currently simply returns a stringified object.

Return type
str

map(*mapping*)

Updates dictionary values, using a mapping function provided.

Parameters
mapping (Callable[[Union[int, float, complex, Expr]], Union[int, float, complex, Expr]]) – Mapping function to update the dict.

Returns
LinearDictCombiner – None.

property n_qubits: int

Returns number of qubits in state space.

Return type
int

property n_symbols: int

Returns the number of free symbols in the object.

Return type
int

norm_coefficients(*order=2*)

Returns the p-norm of the coefficients.

Parameters
order (int, default: 2) – Norm order.

Return type
Union[complex, float]

normalized()

Returns the normalized qubit state.

num_qubits()

Returns number of qubits in state space.

Return type
int

print_table()

Print dictionary formatted as a table.

Return type
NoReturn

remove_global_phase(*phase=0.0*)

Returns a copy with a global phase applied such that the first element has the desired phase.

Parameters

phase (float, default: 0.0) – The phase to yield on the first element, in half-turns (i.e. multiples of pi).

Returns

LinearDictCombiner – A copy of the object with the desired global phase applied.

reversed_order()

Reverses the order of terms and returns it as a new object.

Return type

LinearDictCombiner

simplify(**args*, ***kwargs*)

Simplifies expressions stored in dictionary values.

Parameters

- **args** (Any) – Args to be passed to *sympy.simplify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.simplify()*.

Returns

LinearDictCombiner – Updated instance of LinearDictCombiner.

property single_term

Returns the stored single computational basis state as a list, if only a single computational basis state is stored.

Coefficient is preserved.

Raises

RuntimeError if more than one configuration is stored –

split()

Generates QubitState objects containing only single configurations, preserving coefficients.

property state_symbols

Returns free symbols in coefficients.

subs(*symbol_map*)

Returns a new objects with symbols substituted.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str]) –

Return type

TypeVar(SYMBOLICTYPE, bound= Symbolic)

symbol_substitution(*symbol_map=None*)

Substitutes free symbols for numerical values according to a map.

Parameters

symbol_map (Union[*SymbolDict*, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None) –

Return type
LinearDictCombiner

sympify(**args*, ***kwargs*)

Sympifies dictionary values.

Replaces values with their corresponding symbolic expressions.

Parameters

- **args** (Any) – Args to be passed to *sympy.sympify()*.
- **kwargs** (Any) – Kwargs to be passed to *sympy.sympify()*.

Returns
LinearDictCombiner – Updated instance of LinearDictCombiner.

Raises
RuntimeError – When sympification fails.

property terms: List[Any]

Returns dictionary keys.

Return type
List[Any]

to_ndarray(*bitorder='big'*, *dtype=complex*)

Converts the state to a numpy ndarray.

Assumes a generic register of qubits indexed 0-len(self). Parameter bitorder determines the endianness of the yielded state vector indices.

Parameters

- **bitorder** – the desired endianness of the output bitstring. Options are “little” and “big”.
- **dtype** – the numpy dtype of the resulting ndarray

Returns
ndarray – a numpy ndarray representing the qubit state with requested endianness and dtype

---

Example

```
>>> state = QubitState([1,0])
>>> state_array = state.to_ndarray(bitorder="big")
>>> print(state_array)
[[0.+0.j]
 [0.+0.j]
 [1.+0.j]
 [0.+0.j]]
>>> state = QubitState([1,0])
>>> state_array = state.to_ndarray(bitorder="little")
>>> print(state_array)
[[0.+0.j]
 [1.+0.j]
 [0.+0.j]
 [0.+0.j]]
```

---

unsympify()

Unsympifies dicionary values.

Replaces symbolic expressions with their corresponding numeric values.

Returns
`LinearDictCombiner` – Updated instance of LinearDictCombiner.

Raises
TypeError – When unsympification fails.

vdot(*other*)

Calculates dot product with another QubitState object.

The other QubitState is treated as the ket, this one is treated as the bra.

Parameters
other (*QubitState*) – another qubit state forming the ket of the inner product

Returns
`Union[int, float, complex, Expr]` – the inner product between self and other

classmethod zero()

Return object with a zero dict entry.

Examples

```
>>> print(LinearDictCombiner.zero())
(0)
```

---

class QubitStateString(*initializer: bytes | int | List[int]*)

Bases: `bytes`

Handles a single reference state acting on qubits.

Is intrinsically a byte string.

capitalize() → copy of B

Return a copy of B with only its first character capitalized (ASCII) and the rest lower-cased.

center(*width*, *fillchar=b' '*, */*)

Return a centered string of length width.

Padding is done using the specified fill character.

count(*sub*[, *start*[, *end*]]) → int

Return the number of non-overlapping occurrences of subsection sub in bytes B[start:end]. Optional arguments start and end are interpreted as in slice notation.

decode(*encoding='utf-8'*, *errors='strict'*)

Decode the bytes using the codec registered for encoding.

encoding
The encoding with which to decode the bytes.

errors
The error handling scheme to use for the handling of decoding errors. The default is 'strict' meaning that decoding errors raise a UnicodeDecodeError. Other possible values are 'ignore' and 'replace' as well as any other name registered with codecs.register_error that can handle UnicodeDecodeErrors.

endswith(*suffix*[, *start*[, *end*]]) → bool

Return True if B ends with the specified suffix, False otherwise. With optional start, test B beginning at that position. With optional end, stop comparing B at that position. suffix can also be a tuple of bytes to try.

expandtabs(*tabsize=8*)

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find(*sub*[, *start*[, *end*]]) → int

Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

classmethod from_index(*state_index*, *register_length=None*, *bitorder='big'*)

Generate a QubitStateString from a computational basis state index.

Qubits are assumed to be labelled 0-register_length. Bitorder determines the endianness of the input bitstring - note that internally QubitStateStrings use a little-endian representation.

Parameters

- **state_index** (`int`) – the index of the state as a bitstring
- **register_length** (`Optional[int]`, default: `None`) – the nunber of qubits. Optional: if not provided, minimum length register will be inferred.
- **bitorder** (`str`, default: `"big"`) – the endianness of the input bitstring. Options are "little" and "big"

Returns

QubitStateString – a QubitStateString corresponding to the input index

fromhex()

Create a bytes object from a string of hexadecimal numbers.

Spaces between two numbers are accepted. Example: bytes.fromhex('B9 01EF') -> b'\xb9\x01\xef'.

hex()

Create a str of hexadecimal numbers from a bytes object.

sep
An optional single character or byte to separate hex bytes.

bytes_per_sep
How many bytes between separators. Positive values count from the right, negative values count from the left.

Example: >>> value = b'xb9x01xef' >>> value.hex() 'b901ef' >>> value.hex(':') 'b9:01:ef' >>> value.hex(':', 2) 'b9:01ef' >>> value.hex(':', -2) 'b901:ef'

index(*sub*[, *start*[, *end*]]) → int

Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the subsection is not found.

isalnum() → bool

Return True if all characters in B are alphanumeric and there is at least one character in B, False otherwise.

isalpha() → bool

Return True if all characters in B are alphabetic and there is at least one character in B, False otherwise.

isascii() → bool

Return True if B is empty or all characters in B are ASCII, False otherwise.

isdigit() → bool

Return True if all characters in B are digits and there is at least one character in B, False otherwise.

islower() → bool

Return True if all cased characters in B are lowercase and there is at least one cased character in B, False otherwise.

isspace() → bool

Return True if all characters in B are whitespace and there is at least one character in B, False otherwise.

istitle() → bool

Return True if B is a titlecased string and there is at least one character in B, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

isupper() → bool

Return True if all cased characters in B are uppercase and there is at least one cased character in B, False otherwise.

join(*iterable_of_bytes*, /)

Concatenate any number of bytes objects.

The bytes whose method is called is inserted in between each pair.

The result is returned as a new bytes object.

Example: b'.'.join([b'ab', b'pq', b'rs']) -> b'ab.pq.rs'.

ljust(*width*, *fillchar=b' '*, /)

Return a left-justified string of length width.

Padding is done using the specified fill character.

lower() → copy of B

Return a copy of B with all ASCII characters converted to lowercase.

lstrip(*bytes=None*, /)

Strip leading bytes contained in the argument.

If the argument is omitted or None, strip leading ASCII whitespace.

static maketrans(*frm*, *to*, /)

Return a translation table useable for the bytes or bytearray translate method.

The returned table will be one where each byte in frm is mapped to the byte at the same position in to.

The bytes objects frm and to must be of the same length.

partition(*sep*, /)

Partition the bytes into three parts using the given separator.

This will search for the separator sep in the bytes. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original bytes object and two empty bytes objects.

removeprefix(*prefix*, /)

Return a bytes object with the given prefix string removed if present.

If the bytes starts with the prefix string, return bytes[len(prefix):]. Otherwise, return a copy of the original bytes.

removesuffix(*suffix*, /)

Return a bytes object with the given suffix string removed if present.

If the bytes ends with the suffix string and that suffix is not empty, return bytes[:-len(prefix)]. Otherwise, return a copy of the original bytes.

replace(*old*, *new*, *count=-1*, /)

Return a copy with all occurrences of substring old replaced by new.

count
Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind(*sub*[, *start*[, *end*]]) → int

Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(*sub*[, *start*[, *end*]]) → int

Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Raise ValueError when the subsection is not found.

rjust(*width*, *fillchar=b' '*, /)

Return a right-justified string of length width.

Padding is done using the specified fill character.

rpartition(*sep*, /)

Partition the bytes into three parts using the given separator.

This will search for the separator sep in the bytes, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty bytes objects and the original bytes object.

rsplit(*sep=None*, *maxsplit=-1*)

Return a list of the sections in the bytes, using sep as the delimiter.

sep
The delimiter according which to split the bytes. None (the default value) means split on ASCII whitespace characters (space, tab, return, newline, formfeed, vertical tab).

maxsplit
Maximum number of splits to do. -1 (the default value) means no limit.

Splitting is done starting at the end of the bytes and working to the front.

rstrip(*bytes=None*, /)

Strip trailing bytes contained in the argument.

If the argument is omitted or None, strip trailing ASCII whitespace.

split(*sep=None*, *maxsplit=-1*)

Return a list of the sections in the bytes, using sep as the delimiter.

sep
The delimiter according which to split the bytes. None (the default value) means split on ASCII whitespace characters (space, tab, return, newline, formfeed, vertical tab).

maxsplit
Maximum number of splits to do. -1 (the default value) means no limit.

splitlines(*keepends=False*)

Return a list of the lines in the bytes, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

startswith(*prefix*[, *start*[, *end*]]) → bool

Return True if B starts with the specified prefix, False otherwise. With optional start, test B beginning at that position. With optional end, stop comparing B at that position. prefix can also be a tuple of bytes to try.

strip(*bytes=None*, */*)

Strip leading and trailing bytes contained in the argument.

If the argument is omitted or None, strip leading and trailing ASCII whitespace.

swapcase() → copy of B

Return a copy of B with uppercase ASCII characters converted to lowercase ASCII and vice versa.

title() → copy of B

Return a titlecased version of B, i.e. ASCII words start with uppercase characters, all remaining cased characters have lowercase.

to_index(*bitorder='big'*)

Return the computational basis state index of a single computational basis state.

Bitorder determines the endianness of the resultant index, assuming that self is internally represented in little-endian representation.

Parameters
bitorder (`str`, default: `"big"`) – the desired endianness of the output bitstring

Returns
`int` – the index of the QubitStateString

translate(*table*, */*, *delete=b''*)

Return a copy with each character mapped by the given translation table.

table
Translation table, which must be a bytes object of length 256.

All characters occurring in the optional argument delete are removed. The remaining characters are mapped through the given translation table.

upper() → copy of B

Return a copy of B with all ASCII characters converted to uppercase.

zfill(*width*, */*)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The original string is never truncated.

22.14 inquanto.symmetry

Module for point-group symmetry analysis.

class PointGroup(*point_group*)

Bases: Symmetry

Point group class with associated symmetry processing methods.

Parameters
point_group – The point group label as a string - e.g. "C2v".

compute_representation_components(*representation*)

Finds the combination of irreps which gives the provided representation.

This uses the reduction formula: $n_{row} = \frac{1}{h} * (\sum_N \chi_R \chi_I)$.

Parameters
representation (List[int]) – Representation to be reduced.

Returns
List[Tuple[Any, str]] – Within each tuple, the first element is the coefficient of each irrep in the input representation, and the second element is the irrep symbol.

static get_generators_symbol2irrep_dict(*groupname*)

Retrieve a dictionary which maps the symbols from the provided group to characters of the group generators.

Parameters
groupname (str) – Name of point group of interest.

Returns
Dict[str, Tuple[int]] – Dict whose keys are irrep symbols and whose values are irrep characters.

static get_irrep2symbol_dict(*group*)

Retrieve a dictionary which maps the characters of an irrep to the symbol from the character table.

Parameters
group (str) – Name of point group of interest.

Returns
Dict[Tuple[int], str] – Dictionary whose keys are irrep characters and whose values are irrep symbols.

static get_supported_point_group_dict()

Returns a dictionary yielding all supported point group information, with point group labels as keys.

Returns
Dict – A dictionary with point group labels as keys ("C2v" etc.). Corresponding values are dictionaries containing information regarding the point group (e.g. character table, symmetry element labels).

static get_symbol2irrep_dict(*groupname*)

Retrieve a dictionary which maps the irrep symbols from the provided group to the characters.

Parameters
groupname (str) – Name of point group of interest.

Returns
Dict[str, Tuple[int]] – Dictionary whose keys are irrep symbols and whose values are irrep characters.

irrep_direct_product(*irreps*)

Computes the direct product of the irrep symbols given in irreps.

Parameters

irreps (Iterable[str]) – Iterable containing irrep symbols which belong to the PointGroup object's group.

Returns

Tuple[List[Tuple[int, str]], array] – A tuple where the first element is a list of tuples containing the irrep coefficients and symbols. The second element is the direct product in terms of its characters.

classmethod mini_character_table(*point_group_label*, *irrep_labels*)

For a given point group, return a reduced form of the character table.

This includes only generators of the point_group_label point group and the irreps present in irrep_labels. Generators which are redundant after removal of irreps not in irrep_labels are removed.

Parameters

- **point_group_label** (str) – Label of the point group.
- **irrep_labels** (List[str]) – List of irreducible representation labels.

Returns

Dict[str, Tuple[int]] – A map of irrep labels to characters.

print_character_table()

Pretty print the underlying character table.

static supported_groups()

Return a list of point groups supported by the PointGroup object.

class TapererZ2(*symmetry_operators*, *symmetry_sectors*, *skip_fast_find_x_operators=False*)

Bases: object

Performs Z2 symmetry qubit tapering as described by Bravyi et al. (arXiv:1701.08213 [quant-ph]).

This technique uses Z2 symmetries to reduce qubit requirements for the simulation of a given operator. A TapererZ2 object is instantiated by specifying Pauli symmetry operators as a list of SymmetryOperatorPaulis and a symmetry sector as a list of scalars. Key methods here are .tapered_operator() which tapers operator for a provided set of symmetry operators and sectors, and .tapered_state() which finds a state within the tapered space corresponding to an input state. Note that tapering unitaries and x operators are generated on instantiation and cached. For advanced usage, these may be manually regenerated with the .find_x_operators() and .tapering_unitary() methods.

Parameters

- **symmetry_operators** (List[*SymmetryOperatorPauli*]) – A list of Z2 symmetry operators. Validity is not checked.
- **symmetry_sectors** (List[Union[complex, float, int]]) – Expectation values of each provided symmetry operator for some reference state.
- **skip_fast_find_x_operators** (bool, default: False) – In advanced usage, set to True to skip attempting to find single qubit X operators. Defaults to False.

symmetry_operators

A list of Z2 symmetry operators for which to use tapering.

symmetry_sectors

A list of symmetry sectors for which to use tapering.

taperable_qubits

The qubits which can be removed when tapering an operator or state.

exception XOperatorMinimalError

Bases: Exception

Raised if a valid set of single qubit x operators is requested, but cannot be found.

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

find_x_operators(*skip_minimal=False*, *regenerate=False*, *cache_on_regenerate=True*)

Find a list of X operators for generating tapering unitaries a la arXiv:1701.08213 [quant-ph] eq. 61.

Generated operators include exclusively X or I. The ith returned QubitOperatorString will anticommute with input symmetry operator i, and commute with all other symmetry operators. For efficiency, generated X operators are cached in the TapererZ2 object. With default parameters, this will retrieve the cached operators if possible.

If possible, this method will attempt to find single qubit X operators that fulfil the above criteria. This should work with minimal symmetry operators e.g. from SymmetryZ2 , but not necessarily in all cases e.g. if derived from Fermionic symmetries. If this fails, it currently resorts to brute force, which will be exponentially hard in the number of symmetry operators. Brute force search can be forced by setting skip_minimal=True. To fail in case of failure of the first method, catch BruteForcingXOperatorWarning and except.

Parameters

- **skip_minimal** – Set to True to skip attempting to find single qubit X operators.
- **regenerate** – Set to true to ignore cached x operators and recalculate. Note that if stored x operators are derived from a different setting of skip_minimal, operators will be regenerated regardless of this setting.
- **cache_on_regenerate** (bool, default: True) – If set, cached x operators will be overwritten if x operators are regenerated.

Returns

List[*QubitOperatorString*] – A list of operators including only Xs used to form the tapering unitary.

tapered_operator(*qubit_operator*, *relabel_qubits=False*, *taperable_qubits=None*)

Given a QubitOperator, find the tapered form of the operator.

The tapered operator is as described by Bravyi, Gambetta, Mezzacapo & Temme (arXiv:1701.08213 [quant-ph]), with the strings of Pauli Xs on the taperable qubits replaced by calculated known expectation values. For N independent symmetries (and expectation values) provided, N qubits should be removed.

Parameters

- **qubit_operator** (*QubitOperator*) – The operator to be tapered.
- **relabel_qubits** (bool, default: False) – If True, qubits will be relabelled according to indices of the tapered qubits (e.g. if qubit 1 is tapered, qubit 2 goes to qubit 1, qubit 3 to qubit 2 and so on).
- **taperable_qubits** (Union[List[Qubit], Set[Qubit], None], default: None) – For advanced usage - specify an explicit list of qubits to be tapered rather than using cached taperable qubits determined from the symmetry operators. Validity is not checked - this may lead to unexpected results.

Returns
QubitOperator – The tapered operator.

tapered_state(*state*)

Given a QubitState, find the equivalent QubitState within the tapered space.

This is performed by transforming the state with the tapering unitary and then contracting over the tapered qubits. For N independent symmetries (and expectation values) provided, N qubits should be removed.

Parameters
state (*QubitState*) – The original state in the untapered space.

Returns
QubitState – The tapered state.

tapering_unitary(*regenerate=False*, *cache_on_regenerate=True*, *skip_fast_find_x_operators=False*, *x_operators=None*)

Return the unitary U which transforms an operator to perform qubit tapering.

For an operator with N independent Z2 symmetries, this transforms the operator to an operator with only Pauli X or I acting on a set of N qubits - i.e. U in section VIII of arXiv:1701.08213 [quant-ph]. By default settings, this will be returned from cache - this can be controlled with the regenerate and cache_on_regenerate parameters. x_operators is a list of QubitOperatorStrings including only X or I, where the ith element anticommutes with the ith symmetry_operator and commutes with all other symmetry_operators. By default, this is also retrieved from the cache.

Note that U is Clifford but may not be returned in an efficiently described form.

Parameters

- **regenerate** (bool, default: False) – Set to true to ignore cached unitary and recalculate.
- **cache_on_regenerate** (bool, default: True) – If set, cached unitary will be overwritten if unitary is regenerated.
- **skip_fast_find_x_operators** (bool, default: False) – Set to True to skip attempting to find single qubit X operators. Defaults to False.
- **x_operators** (Optional[List[*QubitOperatorString*]], default: None) – For advanced usage, a list of operators including only Xs used to form the tapering unitary may be provided. If None(default), cached x operators will be used.

Returns
QubitOperator – The Clifford tapering unitary.

CHAPTER

TWENTYTHREE

INQUANTO-EXTENSIONS API REFERENCE

23.1 inquanto-pyscf

InQuanto PySCF extension.

class AVAS (*aolabels*, *threshold=0.2*, *threshold_vir=0.2*, *n_occ=None*, *n_vir=None*, *minao='minao'*, *with_iao=False*, *force_halves_active=False*, *canonicalize=True*, *frozen=[]*, *spin_as=None*, *verbose=None*)

AVAS (Atomic Valence Active Space) / RE (Regional Embedding class) to construct msscf active space.

See arXiv:1701.07862 (AVAS) and https://dx.doi.org/10.1021/acs.jpclett.0c03274 (RE).

Parameters

- **aolabels** (List[str]) – AO labels for AO active space, for example ['Cr 3d', 'Cr 4s'] or ["1 C 2p", "2 C 2p"].
- **threshold** (float, default: 0.2) – Truncating threshold of the AO-projector above which AOs are kept in the active space (occupied orbitals).
- **threshold_vir** (float, default: 0.2) – Truncating threshold of the AO-projector above which AOs are kept in the active space (virtual orbitals).
- **n_occ** (Optional[int], default: None) – None or number of occupied orbitals to keep in the active space. If specified, the value of threshold is ignored.
- **n_vir** (Optional[int], default: None) – None or number of virtual orbitals to keep in the active space. If specified, the value of threshold_vir is ignored.
- **minao** (str, default: "minao") – A reference AO basis for the occupied orbitals in AVAS.
- **with_iao** (bool, default: False) – Whether to use Intrinsic Atomic Orbitals (IAO) localization to construct the reference active AOs.
- **force_halves_active** (bool, default: False) – How to handle singly-occupied orbitals in the active space. The singly-occupied orbitals are projected as part of alpha orbitals if False (default), or completely kept in active space if True. See Section III.E option 2 or 3 of the reference paper for more details.
- **canonicalize** (bool, default: True) – Block-diagonalize and symmetrize the core, active and virtual orbitals.
- **frozen** (Union[List[int], List[List[int]]], default: []) – List of orbitals to be excluded from the AVAS method.
- **spin_as** (Optional[int], default: None) – Number of unpaired electrons in the active space.

- **verbose** (Optional[int], default: None) – Control PySCF verbosity.

Note:

- The reference AO basis for the virtual orbitals is always the computational basis, in contrast to original AVAS but as in the RE method.
- Selecting `force_halves_active=False` for an open-shell system will force double occupations on all orbitals outside of the AVAS active space.

Examples

```
>>> from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF, AVAS
>>> avas= AVAS(aolabels=['Li 2s', 'H 1s'], threshold=0.8, threshold_vir=0.8,␣
↪verbose=5)
>>> driver = ChemistryDriverPySCFMolecularRHF(
...     geometry='Li 0 0 0; H 0 0 1.75', basis='631g',
...     transf=avas, frozen=avas.frozenf)
>>> hamiltonian, space, state = driver.get_system()
```

compute_unitary(*mf*)

Calculate the unitary matrix to transform the MO coefficients.

Returns

ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Unitary matrix to transform the original MO coeffs.

Parameters

mf (HF) –

dump_flags()

Print all modifiable options and their current values.

Return type

None

frozenf(*mf*)

Return a list of frozen orbitals.

Returns

List[int] – Frozen orbital indices.

Parameters

mf (HF) –

property is_transf: bool

Return true if MO transformation is applied.

Returns

bool – True if the orbitals is transformed from HF MOs.

property original: ndarray[Any, dtype[ScalarType]]

Returns the original MO coefficient matrix.

Returns

ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Original MO coefficients of HF.

run (*mf*)

Generate the active space.

Parameters
mf (HF) – PySCF restricted mean-field object.

Raises
NotImplementedError – If the PySCF mean field object is of type UHF.

Returns
Tuple[int, int, ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]] – Number of active orbitals, number of active electrons, orbital coefficients.

transf (*mf*)

Execute the MO transformation to CASSCF natural orbitals.

Parameters
mf (HF) – PySCF mean-field object that can be passed to mcscf.CASSCF.

Return type
None

class CASSCF (**args*, ***kwargs*)

Postprocess orbitals with Complete Active Space Self-Consistent Field (CASSCF) for building molecular integrals.

Parameters

- ***args** – Arguments passed to pyscf.mcscf.CASSCF.
- ****keys** – Keyward arguments passed to pyscf.mcscf.CASSCF.

Examples

```
>>> from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF, CASSCF
>>> # Optimize HOMO-LUMO with CASSCF.
>>> driver = ChemistryDriverPySCFMolecularRHF(geometry='Li 0 0 0; H 0 0 1.75',
...     basis='631g',
...     transf=CASSCF(ncas=2, nelecas=2)
... )
>>> hamiltonian, space, state = driver.get_system()
```

---

compute_unitary (*mf*)

Calculate the unitary matrix to transform the MO coefficients.

Returns
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Unitary matrix to transform the original MO coeffs.

Parameters
mf (HF) –

property is_transf: bool

Return true if MO transformation is applied.

Returns
bool – True if the orbitals is transformed from HF MOs.

property original: ndarray[Any, dtype[ScalarType]]

Returns the original MO coefficient matrix.

Returns
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Original MO coefficients of HF.

transf(*mf*)

Execute the MO transformation to CASSCF natural orbitals.

Parameters
mf (HF) – PySCF mean-field object that can be passed to mcscf.CASSCF.

Return type
None

class ChemistryDriverPySCFEmbeddingRHF(*geometry=None*, *zmatrix=None*, *basis=None*, *ecp=None*, *charge=0*, *frozen=None*, *transf=None*, *verbose=0*, *output=None*, *point_group_symmetry=False*, *functional='b3lyp'*, *transf_inner=None*, *frozen_inner=None*, *level_shift=1.0e6*)

Projective embedding. Partially based on PsiEmbed.

Implements Projection-based embedding, F.R. Manby, M. Stella, J.D. Goodpaster, T.F. Miller. III,

J. Chem. Theory Comput. 2012, 8, 2564.

TODO: Docstrings.

Parameters

- **geometry** (Union[List, str, Geometry, None], default: None) –
- **zmatrix** (Optional[str], default: None) –
- **basis** (Optional[Any], default: None) –
- **ecp** (Optional[Any], default: None) –
- **charge** (int, default: 0) –
- **frozen** (Union[List[int], Callable[[RHF], int], None], default: None) –
- **transf** (Optional[Transf], default: None) –
- **verbose** (int, default: 0) –
- **output** (Optional[str], default: None) –
- **point_group_symmetry** (bool, default: False) –
- **functional** (str, default: "b3lyp") –
- **transf_inner** (Union[Callable[[ndarray], ndarray], Transf, None], default: None) –
- **frozen_inner** (Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]], None], default: None) –
- **level_shift** (float, default: 1.0e6) –

compute_nuclear_dipole()

Compute the nuclear electric dipole.

Returns
Tuple[float, float, float] – x, y, and z components of nuclear electric dipole.

compute_one_electron_operator(*oper*, *origin=(0, 0, 0)*)

Compute a one-electron fermionic operator in atomic units.

For operators with origin-dependent expectation values, origin can be specified. oper values:

- 'kin' - kinetic energy.
- 'nuc' - nucleus-electron attraction energy.
- 'hcore' - one-electron hamiltonian.
- 'ovlp' - one-electron overlap.
- 'r' - electronic first moment (x, y, z).
- 'rr' - electronic second moment (xx, xy, xz, yx, yy, yz, zx, zy, zz).
- 'dm' - electronic dipole moment (x, y, z).

Parameters

- **oper** (`str`) – Key to specify the operator.
- **origin** (`tuple`, default: `(0, 0, 0)`) – Coordinate position of the origin.

Returns

`Union`[*FermionOperator*, `List`[*FermionOperator*]] – One electron operators.

extract_point_group_information(*reduce_infinite_point_groups=True*)

Returns point group information.

Parameters

reduce_infinite_point_groups – Reduce infinite point groups, e.g., Coov -> C2v

Returns

`Tuple`[`str`, `List`[`str`]] – Point group symmetry, list of orbital irreps.

Examples

```
>>> from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF
>>> driver = ChemistryDriverPySCFMolecularRHF(
...     geometry='H 0 0 0; H 0 0 0.75',
...     basis='sto3g',
...     point_group_symmetry=True,
... )
>>> ham, space, state = driver.get_system()
>>> driver.extract_point_group_information()
('D2h', ['Ag', 'Ag', 'B1u', 'B1u'])
```

---

classmethod from_mf(*mf*, *frozen=None*, *transf=None*, *transf_inner=None*, *frozen_inner=None*, *level_shift=1.0e6*)

Initialize Projection-embedding driver from a PySCF mean-field object.

Use transf to localise orbitals and frozen to select active orbitals.

Parameters

- **mf** (`RHF`) – PySCF mean-field object, must be RKS or RHF.

- **frozen** (Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]], None], default: None) – Frozen core specified as either list or callable.
- **transf** (Union[Callable[[ndarray], ndarray], Transf, None], default: None) – Orbital transformation function.
- **level_shift** (float, default: 1.0e6) – Projection-based embedding level shift
- **transf_inner** (Union[Callable[[ndarray], ndarray], Transf, None], default: None) –
- **frozen_inner** (Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]], None], default: None) –

Returns
ChemistryDriverPySCFEmbeddingRHF – PySCF driver.

property frozen: List[int] | List[List[int]]

Return the frozen orbital information.

Return type
Union[List[int], List[List[int]]]

generate_report()

Generate report in a hierarchical dictionary format.

PySCF attributes such as mo_coeff are exported if the SCF is converged.

Returns
Dict[str, Any] – Attributes of the internal PySCF mean-field object.

get_cube_density(*density_matrix*, *cube_resolution=0.25*)

Output Gaussian Cube contents for density.

Parameters

- **density_matrix** (ndarray) – One-body reduced density matrix. 2D array for RHF/ROHF, 3D array for UHF.
- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.

Returns
str – Cube file formatted string.

get_cube_orbitals(*cube_resolution=0.25*, *mo_coeff=None*, *orbital_indices=None*)

Output Gaussian Cube file contents for orbitals.

Parameters

- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.
- **mo_coeff** (Optional[array], default: None) – Molecular orbital coefficients for the orbitals to be visualised.
- **orbital_indices** (Optional[List], default: None) – Indices of the molecular orbitals of interest. If None, all orbitals are returned.

Returns
List[str] – List of cube file formatted strings. For UHF, the spinorbitals are returned as [mo1_alpha, mo1_beta, mo2_alpha, mo2_beta...].

get_excitation_amplitudes(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*)

Get the (guess) parameters as excitation amplitudes from a classical calculation.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.

Returns

SymbolDict – Excitation parameters.

get_excitation_operators(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*, *antihermitian=True*)

Get the excitation operators and (guess) parameters.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.
- **antihermitian** (bool, default: True) – Returns anti-Hermitian operators T - T^ if specified.

Returns

FermionOperatorList – Excitation operator list.

get_lowdin_system(*method='lowdin'*)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

method (str, default: "lowdin") – Method passed to PySCF's orth.orth_ao().

Returns

Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*], *FermionSpace*, Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*]] – Fermion Hamiltonian, Fock space, Fock state.

get_mulliken_pop()

Interface for Mulliken population analysis of PySCF.

Returns

Tuple[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]] – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction(*rdms*)

Compute the Strongly contracted NEVPT2 correction to the energy from the provided density matrices.

Parameters

rdms (Tuple) – A tuple of reduced density matrices as PDMs in PySCF's un-reordered format as (pdm1, …, rdm4).

Returns

The NEVPT2 correction to the energy.

get_rdm1_ccsd()

Reduced one-body density matrix in the AO basis from CCSD.

Returns

Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*] – One-body reduced density matrix.

get_rdm2_ccsd()

Reduced two-body density matrix in the AO basis with CCSD.

Returns

ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Two-body reduced density matrix.

Note: This object will be replaced with an RDM2 class not to return a raw 4D tensor.

---

get_system(*symmetry=1*)

TODO.

Parameters

symmetry (Union[str, int], default: 1) –

Return type

Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*, *ChemistryRestrictedIntegralOperatorCompact*, *ChemistryUnrestrictedIntegralOperatorCompact*], *FermionSpace*, *FermionState*]

get_system_ao(*run_hf=True*)

TODO.

Parameters

run_hf (bool, default: True) –

Return type

Tuple[Union[*PySCFChemistryRestrictedIntegralOperator*, *PySCFChemistryUnrestrictedIntegralOperator*], Optional[*FermionSpace*], Optional[*FermionState*]]

property mf_energy: float

Return the total mean-field energy.

Return type

float

property mf_type: str

Return the mean-field type as a string (e.g. "RHF").

Returns
str – Mean-field type name.

property n_electron: int
Number of electrons in the active space.

Returns
int – Number of electrons.

property n_orb: int
Number of spatial orbitals.

Returns
int – Number of spatial orbitals.

print_json_report(**args*, ***kwargs*)
Prints report in json format.

run_casci(***kwargs*)
Calculate the CASCI energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the `mcscf.CASCI` object.

Returns
float – CASCI energy.

run_ccsd(***kwargs*)
Calculate the CCSD energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the `cc.CCSD` object.

Returns
float – CCSD energy

run_hf()
Calculate the HF energy.

Returns
float – HF energy

run_mp2(***kwargs*)
Calculate the MP2 energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the `mp.MP2` object.

Returns
float – MP2 energy

class ChemistryDriverPySCFEmbeddingROHF(*geometry=None*, *zmatrix=None*, *basis=None*, *ecp=None*, *charge=0*, *multiplicity=1*, *frozen=None*, *transf=None*, *verbose=0*, *output=None*, *point_group_symmetry=False*, *embedded_spin=None*, *functional='b3lyp'*, *transf_inner=None*, *frozen_inner=None*, *level_shift=1.0e6*)
Projective embedding. Partially based on PsiEmbed.

Implements Projection-based embedding, F.R. Manby, M. Stella, J.D. Goodpaster, T.F. Miller. III,

J. Chem. Theory Comput. 2012, 8, 2564.

TODO: Docstrings.

Parameters

- **geometry** (Union[List, str, Geometry, None], default: None) –
- **zmatrix** (Optional[str], default: None) –
- **basis** (Optional[Any], default: None) –
- **ecp** (Optional[Any], default: None) –
- **charge** (int, default: 0) –
- **multiplicity** (int, default: 1) –
- **frozen** (Optional[Any], default: None) –
- **transf** (Optional[Transf], default: None) –
- **verbose** (int, default: 0) –
- **output** (Optional[str], default: None) –
- **point_group_symmetry** (bool, default: False) –
- **embedded_spin** (Optional[int], default: None) –
- **functional** (str, default: "b3lyp") –
- **transf_inner** (Union[Callable[[ndarray], ndarray], Transf, None], default: None) –
- **frozen_inner** (Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]], None], default: None) –
- **level_shift** (float, default: 1.0e6) –

compute_nuclear_dipole()

Compute the nuclear electric dipole.

Returns

Tuple[float, float, float] – x, y, and z components of nuclear electric dipole.

compute_one_electron_operator(*oper*, *origin=(0, 0, 0)*)

Compute a one-electron fermionic operator in atomic units.

For operators with origin-dependent expectation values, origin can be specified. oper values:

- 'kin' - kinetic energy.
- 'nuc' - nucleus-electron attraction energy.
- 'hcore' - one-electron hamiltonian.
- 'ovlp' - one-electron overlap.
- 'r' - electronic first moment (x, y, z).
- 'rr' - electronic second moment (xx, xy, xz, yx, yy, yz, zx, zy, zz).
- 'dm' - electronic dipole moment (x, y, z).

Parameters

- **oper** (str) – Key to specify the operator.
- **origin** (tuple, default: (0, 0, 0)) – Coordinate position of the origin.

Returns
Union[*FermionOperator*, List[*FermionOperator*]] – One electron operators.

extract_point_group_information(*reduce_infinite_point_groups=True*)

Returns point group information.

Parameters
reduce_infinite_point_groups – Reduce infinite point groups, e.g., Coov -> C2v

Returns
Tuple[str, List[str]] – Point group symmetry, list of orbital irreps.

Examples

```
>>> from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF
>>> driver = ChemistryDriverPySCFMolecularRHF(
...     geometry='H 0 0 0; H 0 0 0.75',
...     basis='sto3g',
...     point_group_symmetry=True,
... )
>>> ham, space, state = driver.get_system()
>>> driver.extract_point_group_information()
('D2h', ['Ag', 'Ag', 'B1u', 'B1u'])
```

classmethod from_mf(*mf*, *frozen=None*, *transf=None*, *transf_inner=None*, *frozen_inner=None*, *embedded_spin=None*, *level_shift=1.0e6*)

Initialize Projection-embedding driver from a PySCF mean-field object.

Use transf to localise orbitals and frozen to select active orbitals. Note: when creating the IntegralOperator, the 1-electron embedding potential is averaged over spin channels (would have to use UHF to avoid this).

Parameters

- **mf** (ROKS) – PySCF mean-field object, must be ROKS or ROHF.
- **frozen** (Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]], None], default: None) – Embedding (bath) orbitals
- **transf** (Union[Callable[[ndarray], ndarray], Transf, None], default: None) – Orbital transformation function (to be used on the ROKS input).
- **transf_inner** (Union[Callable[[ndarray], ndarray], Transf, None], default: None) – Orbital transformation function (to be used on the embedded system)
- **embedded_spin** (Optional[int], default: None) – Number of unpaired electrons in the embedded system (if different than total)
- **level_shift** (float, default: 1.0e6) – Projection-based embedding level shift
- **frozen_inner** (Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]], None], default: None) –

Returns
ChemistryDriverPySCFEmbeddingROHF – PySCF driver.

property frozen: List[int] | List[List[int]]

Return the frozen orbital information.

Return type
Union[List[int], List[List[int]]]

generate_report()

Generate report in a hierarchical dictionary format.

PySCF attributes such as mo_coeff are exported if the SCF is converged.

Returns
Dict[str, Any] – Attributes of the internal PySCF mean-field object.

get_cube_density(*density_matrix*, *cube_resolution=0.25*)

Output Gaussian Cube contents for density.

Parameters

- **density_matrix** (ndarray) – One-body reduced density matrix. 2D array for RHF/ROHF, 3D array for UHF.
- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.

Returns
str – Cube file formatted string.

get_cube_orbitals(*cube_resolution=0.25*, *mo_coeff=None*, *orbital_indices=None*)

Output Gaussian Cube file contents for orbitals.

Parameters

- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.
- **mo_coeff** (Optional[array], default: None) – Molecular orbital coefficients for the orbitals to be visualised.
- **orbital_indices** (Optional[List], default: None) – Indices of the molecular orbitals of interest. If None, all orbitals are returned.

Returns
List[str] – List of cube file formatted strings. For UHF, the spinorbitals are returned as [mo1_alpha, mo1_beta, mo2_alpha, mo2_beta...].

get_excitation_amplitudes(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*)

Get the (guess) parameters as excitation amplitudes from a classical calculation.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.

Returns
SymbolDict – Excitation parameters.

get_excitation_operators(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*, *antihermitian=True*)

Get the excitation operators and (guess) parameters.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.
- **antihermitian** (bool, default: True) – Returns anti-Hermitian operators T - T^ if specified.

Returns

FermionOperatorList – Excitation operator list.

get_lowdin_system(*method='lowdin'*)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

method (str, default: "lowdin") – Method passed to PySCF's orth.orth_ao().

Returns

Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*], *FermionSpace*, Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*]] – Fermion Hamiltonian, Fock space, Fock state.

get_mulliken_pop()

Interface for Mulliken population analysis of PySCF.

Returns

Tuple[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]] – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction(*rdms*)

Compute the Strongly contracted NEVPT2 correction to the energy from the provided density matrices.

Parameters

rdms (Tuple) – A tuple of reduced density matrices as PDMs in PySCF's un-reordered format as (pdm1, …, rdm4).

Returns

The NEVPT2 correction to the energy.

get_rdm1_ccsd()

Reduced one-body density matrix in the AO basis from CCSD.

Returns

Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*] – One-body reduced density matrix.

get_rdm2_ccsd()

Reduced two-body density matrix in the AO basis with CCSD.

Returns
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Two-body reduced density matrix.

Note: This object will be replaced with an RDM2 class not to return a raw 4D tensor.

get_system(*symmetry=1*)

TODO.

Parameters
symmetry (Union[str, int], default: 1) –

Return type
Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*, *ChemistryRestrictedIntegralOperatorCompact*, *ChemistryUnrestrictedIntegralOperatorCompact*], *FermionSpace*, *FermionState*]

get_system_ao(*run_hf=True*)

TODO.

Parameters
run_hf (bool, default: True) –

Return type
Tuple[Union[*PySCFChemistryRestrictedIntegralOperator*, *PySCFChemistryUnrestrictedIntegralOperator*], Optional[*FermionSpace*], Optional[*FermionState*]]

property mf_energy: float

Return the total mean-field energy.

Return type
float

property mf_type: str

Return the mean-field type as a string (e.g. "RHF").

Returns
str – Mean-field type name.

property n_electron: int

Number of electrons in the active space.

Returns
int – Number of electrons.

property n_orb: int

Number of spatial orbitals.

Returns
int – Number of spatial orbitals.

print_json_report(**args*, ***kwargs*)

Prints report in json format.

run_casci(***kwargs*)

Calculate the CASCI energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the `mcscf.CASCI` object.

Returns

`float` – CASCI energy.

run_ccsd(***kwargs*)

Calculate the CCSD energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the `cc.CCSD` object.

Returns

`float` – CCSD energy

run_hf()

Calculate the HF energy.

Returns

`float` – HF energy

run_mp2(***kwargs*)

Calculate the MP2 energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the `mp.MP2` object.

Returns

`float` – MP2 energy

class ChemistryDriverPySCFEmbeddingROHF_UHF(*geometry=None*, *zmatrix=None*, *basis=None*, *ecp=None*, *charge=0*, *multiplicity=1*, *frozen=None*, *transf=None*, *verbose=0*, *output=None*, *point_group_symmetry=False*, *embedded_spin=None*, *functional='b3lyp'*, *transf_inner=None*, *frozen_inner=None*, *level_shift=1.0e6*)

Projective embedding. Partially based on PsiEmbed.

Implements Projection-based embedding, F.R. Manby, M. Stella, J.D. Goodpaster, T.F. Miller. III,

J. Chem. Theory Comput. 2012, 8, 2564.

TODO: Docstrings before being made public.

Parameters

- **geometry** (`Union[List, str, Geometry, None]`, default: `None`) –
- **zmatrix** (`Optional[str]`, default: `None`) –
- **basis** (`Optional[Any]`, default: `None`) –
- **ecp** (`Optional[Any]`, default: `None`) –
- **charge** (`int`, default: `0`) –
- **multiplicity** (`int`, default: `1`) –
- **frozen** (`Optional[Any]`, default: `None`) –

- **transf** (Optional[Transf], default: None) –
- **verbose** (int, default: 0) –
- **output** (Optional[str], default: None) –
- **point_group_symmetry** (bool, default: False) –
- **embedded_spin** (Optional[int], default: None) –
- **functional** (str, default: "b3lyp") –
- **transf_inner** (Union[Callable[[ndarray], ndarray], Transf, None], default: None) –
- **frozen_inner** (Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]], None], default: None) –
- **level_shift** (float, default: 1.0e6) –

compute_nuclear_dipole()

Compute the nuclear electric dipole.

Returns

Tuple[float, float, float] – x, y, and z components of nuclear electric dipole.

compute_one_electron_operator(*oper*, *origin=(0, 0, 0)*)

Compute a one-electron fermionic operator in atomic units.

For operators with origin-dependent expectation values, origin can be specified. oper values:

- 'kin' - kinetic energy.
- 'nuc' - nucleus-electron attraction energy.
- 'hcore' - one-electron hamiltonian.
- 'ovlp' - one-electron overlap.
- 'r' - electronic first moment (x, y, z).
- 'rr' - electronic second moment (xx, xy, xz, yx, yy, yz, zx, zy, zz).
- 'dm' - electronic dipole moment (x, y, z).

Parameters

- **oper** (str) – Key to specify the operator.
- **origin** (tuple, default: (0, 0, 0)) – Coordinate position of the origin.

Returns

Union[*FermionOperator*, List[*FermionOperator*]] – One electron operators.

extract_point_group_information(*reduce_infinite_point_groups=True*)

Returns point group information.

Parameters

reduce_infinite_point_groups – Reduce infinite point groups, e.g., Coov -> C2v

Returns

Tuple[str, List[str]] – Point group symmetry, list of orbital irreps.

Examples

```
>>> from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF
>>> driver = ChemistryDriverPySCFMolecularRHF(
...     geometry='H 0 0 0; H 0 0 0.75',
...     basis='sto3g',
...     point_group_symmetry=True,
... )
>>> ham, space, state = driver.get_system()
>>> driver.extract_point_group_information()
('D2h', ['Ag', 'Ag', 'B1u', 'B1u'])
```

---

classmethod from_mf(*mf*, *frozen=None*, *transf=None*, *transf_inner=None*, *frozen_inner=None*, *embedded_spin=None*, *level_shift=1.0e6*)

Initialize Projection-embedding driver from a PySCF mean-field object.

Use transf to localise orbitals and frozen to select active orbitals. Note: when creating the IntegralOperator, the 1-electron embedding potential is averaged over spin channels (would have to use UHF to avoid this).

Parameters

- **mf** (ROKS) – PySCF mean-field object, must be ROKS or ROHF.
- **frozen** (Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]], default: None) – Embedding (bath) orbitals
- **transf** (Union[Callable[[ndarray], ndarray], Transf], default: None) – Orbital transformation function (to be used on the ROKS input).
- **transf_inner** (Union[Callable[[ndarray], ndarray], Transf], default: None) – Orbital transformation function (to be used on the embedded system)
- **embedded_spin** (int, default: None) – Number of unpaired electrons in the embedded system (if different than total)
- **level_shift** (float, default: 1.0e6) – Projection-based embedding level shift
- **frozen_inner** (Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]], default: None) –

Returns

ChemistryDriverPySCFEmbeddingROHF – PySCF driver.

property frozen: List[int] | List[List[int]]

Return the frozen orbital information.

Return type

Union[List[int], List[List[int]]]

generate_report()

Generate report in a hierarchical dictionary format.

PySCF attributes such as mo_coeff are exported if the SCF is converged.

Returns

Dict[str, Any] – Attributes of the internal PySCF mean-field object.

get_cube_density(*density_matrix*, *cube_resolution=0.25*)

Output Gaussian Cube contents for density.

Parameters

- **density_matrix** (ndarray) – One-body reduced density matrix. 2D array for RHF/ROHF, 3D array for UHF.
- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.

Returns
str – Cube file formatted string.

get_cube_orbitals (*cube_resolution=0.25*, *mo_coeff=None*, *orbital_indices=None*)

Output Gaussian Cube file contents for orbitals.

Parameters

- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.
- **mo_coeff** (Optional[array], default: None) – Molecular orbital coefficients for the orbitals to be visualised.
- **orbital_indices** (Optional[List], default: None) – Indices of the molecular orbitals of interest. If None, all orbitals are returned.

Returns
List[str] – List of cube file formatted strings. For UHF, the spinorbitals are returned as [mo1_alpha, mo1_beta, mo2_alpha, mo2_beta...].

get_excitation_amplitudes (*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*)

Get the (guess) parameters as excitation amplitudes from a classical calculation.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.

Returns
SymbolDict – Excitation parameters.

get_excitation_operators (*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*, *antihermitian=True*)

Get the excitation operators and (guess) parameters.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.

- **antihermitian** (bool, default: True) – Returns anti-Hermitian operators T - T^ if specified.

Returns
FermionOperatorList – Excitation operator list.

get_lowdin_system(*method='lowdin'*)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters
method (str, default: "lowdin") – Method passed to PySCF's orth.orth_ao().

Returns
Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*], *FermionSpace*, Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*]] – Fermion Hamiltonian, Fock space, Fock state.

get_mulliken_pop()

Interface for Mulliken population analysis of PySCF.

Returns
Tuple[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]] – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction(*rdms*)

Compute the Strongly contracted NEVPT2 correction to the energy from the provided density matrices.

Parameters
rdms (Tuple) – A tuple of reduced density matrices as PDMs in PySCF's un-reordered format as (pdm1, …, rdm4).

Returns
The NEVPT2 correction to the energy.

get_rdm1_ccsd()

Reduced one-body density matrix in the AO basis from CCSD.

Returns
Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*] – One-body reduced density matrix.

get_rdm2_ccsd()

Reduced two-body density matrix in the AO basis with CCSD.

Returns
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Two-body reduced density matrix.

Note: This object will be replaced with an RDM2 class not to return a raw 4D tensor.

---

get_system(*symmetry=1*)

TODO.

Parameters
symmetry (Union[str, int], default: 1) –

Return type
Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*, *ChemistryRestrictedIntegralOperatorCompact*, *ChemistryUnrestrictedIntegralOperatorCompact*], *FermionSpace*, *FermionState*]

get_system_ao(*run_hf=True*)

TODO.

Parameters
run_hf (bool, default: True) –

Return type
Tuple[Union[*PySCFChemistryRestrictedIntegralOperator*, *PySCFChemistryUnrestrictedIntegralOperator*], Optional[*FermionSpace*], Optional[*FermionState*]]

property mf_energy: float

Return the total mean-field energy.

Return type
float

property mf_type: str

Return the mean-field type as a string (e.g. "RHF").

Returns
str – Mean-field type name.

property n_electron: int

Number of electrons in the active space.

Returns
int – Number of electrons.

property n_orb: int

Number of spatial orbitals.

Returns
int – Number of spatial orbitals.

print_json_report(**args*, ***kwargs*)

Prints report in json format.

run_casci(***kwargs*)

Calculate the CASCI energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the mcscf.CASCI object.

Returns
float – CASCI energy.

run_ccsd(***kwargs*)

Calculate the CCSD energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the cc.CCSD object.

Returns
float – CCSD energy

run_hf()

Calculate the HF energy.

Returns
float – HF energy

run_mp2(***kwargs*)

Calculate the MP2 energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the mp.MP2 object.

Returns
float – MP2 energy

class ChemistryDriverPySCFGammaRHF(*geometry=None*, *zmatrix=None*, *cell=None*, *basis=None*, *ecp=None*, *pseudo=None*, *charge=0*, *exp_to_discard=None*, *df='GDF'*, *dimension=3*, *frozen=None*, *output=None*, *verbose=0*)

PySCF driver for Gamma-point RHF calculations.

Parameters

- **geometry** (Union[List, str, *GeometryPeriodic*, None], default: None) – Molecular geometry.
- **zmatrix** (Optional[str], default: None) – Z matrix representation of molecular geometry (Used only if geometry is not specified).
- **cell** (Optional[ndarray], default: None) – Unit cell parameter. If provided, overrides geometry.unit_cell.
- **basis** (Optional[Any], default: None) – Atomic basis set valid for Mole class.
- **ecp** (Optional[Any], default: None) – Effective core potentials.
- **pseudo** (Optional[Any], default: None) – Pseudo potentials.
- **charge** (int, default: 0) – Total charge.
- **exp_to_discard** (Optional[float], default: None) – Exponent to discard a primitive Gaussian.
- **df** (str, default: "GDF") – Density fitting function name.
- **dimension** (int, default: 3) – Number of spatial dimensions.
- **frozen** (Union[List[int], Callable[[RHF], List[int]], None], default: None) – Frozen orbital information.
- **output** (Optional[str], default: None) – Specify log file name. If None, logs are printed to STDOUT.
- **verbose** (int, default: 0) – Control PySCF verbosity.

compute_nuclear_dipole()

Compute the nuclear electric dipole.

Returns
Tuple[float, float, float] – x, y, and z components of nuclear electric dipole.

compute_one_electron_operator(*oper*, *origin=(0, 0, 0)*)

Compute a one-electron fermionic operator in atomic units.

For operators with origin-dependent expectation values, origin can be specified. oper values:

- 'kin' - kinetic energy.
- 'nuc' - nucleus-electron attraction energy.
- 'hcore' - one-electron hamiltonian.
- 'ovlp' - one-electron overlap.
- 'r' - electronic first moment (x, y, z).
- 'rr' - electronic second moment (xx, xy, xz, yx, yy, yz, zx, zy, zz).
- 'dm' - electronic dipole moment (x, y, z).

Parameters

- **oper** (`str`) – Key to specify the operator.
- **origin** (`tuple`, default: `(0, 0, 0)`) – Coordinate position of the origin.

Returns

`Union`[*FermionOperator*, `List`[*FermionOperator*]] – One electron operators.

extract_point_group_information (*reduce_infinite_point_groups=True*)

Returns point group information.

Parameters

reduce_infinite_point_groups – Reduce infinite point groups, e.g., Coov -> C2v

Returns

`Tuple`[`str`, `List`[`str`]] – Point group symmetry, list of orbital irreps.

Examples

```
>>> from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF
>>> driver = ChemistryDriverPySCFMolecularRHF(
...     geometry='H 0 0 0; H 0 0 0.75',
...     basis='sto3g',
...     point_group_symmetry=True,
... )
>>> ham, space, state = driver.get_system()
>>> driver.extract_point_group_information()
('D2h', ['Ag', 'Ag', 'B1u', 'B1u'])
```

---

classmethod from_mf (*mf*, *frozen=None*)

Initialize driver from a PySCF mean-field object.

Parameters

- **mf** (`SCF`) – PySCF mean-field object.
- **frozen** (`Union`[`List`[`int`], `List`[`List`[`int`]], `Callable`[[`SCF`], `Union`[`List`[`int`], `List`[`List`[`int`]]]], `None`], default: `None`) – Frozen core specified as either list or callable.

Returns

ChemistryDriverPySCFGammaRHF – PySCF driver.

property frozen: List[int] | List[List[int]]

Return the frozen orbital information.

Return type
Union[List[int], List[List[int]]]

generate_report()

Generate report in a hierarchical dictionary format.

PySCF attributes such as `mo_coeff` are exported if the SCF is converged.

Returns
Dict[str, Any] – Attributes of the internal PySCF mean-field object.

get_cube_density(*density_matrix*, *cube_resolution=0.25*)

Output Gaussian Cube contents for density.

Parameters

- **density_matrix** (ndarray) – One-body reduced density matrix. 2D array for RHF/ROHF, 3D array for UHF.
- **cube_resolution** (float, default: 0.25) – Resolution to be passed to `cubegen.orbital`.

Returns
str – Cube file formatted string.

get_cube_orbitals(*cube_resolution=0.25*, *mo_coeff=None*, *orbital_indices=None*)

Output Gaussian Cube file contents for orbitals.

Parameters

- **cube_resolution** (float, default: 0.25) – Resolution to be passed to `cubegen.orbital`.
- **mo_coeff** (Optional[array], default: None) – Molecular orbital coefficients for the orbitals to be visualised.
- **orbital_indices** (Optional[List], default: None) – Indices of the molecular orbitals of interest. If None, all orbitals are returned.

Returns
List[str] – List of cube file formatted strings. For UHF, the spinorbitals are returned as `[mo1_alpha, mo1_beta, mo2_alpha, mo2_beta...]`.

get_excitation_amplitudes(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*)

Get the (guess) parameters as excitation amplitudes from a classical calculation.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.

Returns
SymbolDict – Excitation parameters.

get_excitation_operators(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*, *antihermitian=True*)

Get the excitation operators and (guess) parameters.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.
- **antihermitian** (bool, default: True) – Returns anti-Hermitian operators T - T^ if specified.

Returns
FermionOperatorList – Excitation operator list.

get_lowdin_system(*method='lowdin'*)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters
method (str, default: "lowdin") – Method passed to PySCF's orth.orth_ao().

Returns
Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*], *FermionSpace*, Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*]] – Fermion Hamiltonian, Fock space, Fock state.

get_madelung_constant()

Return Madelung constant for Gamma-point calculations.

Returns
float – Madelung constant contribution to the energy.

get_mulliken_pop()

Interface for Mulliken population analysis of PySCF.

Returns
Tuple[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]] – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction(*rdms*)

Compute the Strongly contracted NEVPT2 correction to the energy from the provided density matrices.

Parameters
rdms (Tuple) – A tuple of reduced density matrices as PDMs in PySCF's un-reordered format as (pdm1, …, rdm4).

Returns
The NEVPT2 correction to the energy.

get_rdm1_ccsd()

Reduced one-body density matrix in the AO basis from CCSD.

Returns

Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*] – One-body reduced density matrix.

get_rdm2_ccsd()

Reduced two-body density matrix in the AO basis with CCSD.

Returns

ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Two-body reduced density matrix.

Note: This object will be replaced with an RDM2 class not to return a raw 4D tensor.

---

get_system(*symmetry=1*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Parameters

symmetry (Union[str, int], default: 1) – Code to specify target symmetry for storage of two-body integrals. Uses the same convention as PySCF. Currently, supports s1, s4 and s8 index permutation symmetries.

Warning: For unrestricted integral operators, the aabb and bbaa two-body integrals cannot be compacted with s8 symmetry. If s8 symmetry is requested, the aaaa and bbbb will be stored with s8 symmetry, while the aabb and bbaa integrals will be stored with s4 symmetry.

Returns

Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*, *ChemistryRestrictedIntegralOperatorCompact*, *ChemistryUnrestrictedIntegralOperatorCompact*], *FermionSpace*, *FermionState*] – Fermion Hamiltonian, Fock space, Fock state.

get_system_ao(*run_hf=True*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Output hamiltonian operator will be of type *PySCFChemistryRestrictedIntegralOperator* or *PySCFChemistryUnrestrictedIntegralOperator*, which store the underlying atomic orbitals.

Parameters

run_hf (bool, default: True) – If True, a Hartree-Fock calculation for the system will be executed, and the Fock space and HF state are returned with the hamiltonian operator. If False, no calculation is performed and the Fock space and HF state are returned as None. If the PySCF mean-field object has already been converged prior to calling get_system_ao(), behaviour will follow run_hf=True.

Returns

Tuple[Union[*PySCFChemistryRestrictedIntegralOperator*, *PySCFChemistryUnrestrictedIntegralOperator*], Optional[*FermionSpace*], Optional[*FermionState*]] – Fermion Hamiltonian, Fock space, Fock state.

property mf_energy: float

Return the total mean-field energy.

Return type
float

property mf_type: str

Return the mean-field type as a string (e.g. "RHF").

Returns
str – Mean-field type name.

property n_electron: int

Number of electrons in the active space.

Returns
int – Number of electrons.

property n_orb: int

Number of spatial orbitals.

Returns
int – Number of spatial orbitals.

print_json_report(**args*, ***kwargs*)

Prints report in json format.

run_casci(***kwargs*)

Calculate the CASCI energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the mcscf.CASCI object.

Returns
float – CASCI energy.

run_ccsd(***kwargs*)

Calculate the CCSD energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the cc.CCSD object.

Returns
float – CCSD energy

run_hf()

Calculate the HF energy.

Returns
float – HF energy

run_mp2(***kwargs*)

Calculate the MP2 energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the mp.MP2 object.

Returns
float – MP2 energy

class ChemistryDriverPySCFGammaROHF(*geometry=None*, *zmatrix=None*, *cell=None*, *basis=None*, *ecp=None*, *pseudo=None*, *charge=0*, *multiplicity=1*, *exp_to_discard=None*, *df='GDF'*, *dimension=3*, *frozen=None*, *verbose=0*, *output=None*)

PySCF driver for Gamma-point ROHF calculations.

Parameters

- **geometry** (Union[List, str, *GeometryPeriodic*, None], default: None) – Molecular geometry.
- **zmatrix** (Optional[str], default: None) – Z matrix representation of molecular geometry (Used only is geometry is not specified).
- **cell** (Optional[List[List]], default: None) – Unit cell parameter. If provided, overrides geometry.unit_cell.
- **basis** (Optional[Any], default: None) – Atomic basis set valid for Mole class.
- **ecp** (Optional[Any], default: None) – Effective core potentials.
- **pseudo** (Optional[Any], default: None) – Pseudo potentials.
- **charge** (int, default: 0) – Total charge.
- **multiplicity** (int, default: 1) – Spin multicplicity, 2S+1.
- **exp_to_discard** (Optional[float], default: None) – Exponent to discard a primitive Gaussian.
- **df** (str, default: "GDF") – Density fitting function name.
- **dimension** (int, default: 3) – Number of spatial dimensions.
- **frozen** (Union[List[int], Callable[[RHF], List[int]], None], default: None) – Frozen orbital information.
- **output** (Optional[str], default: None) – Specify log file name. If None, logs are printed to STDOUT.
- **verbose** (int, default: 0) – Control PySCF verbosity.

compute_nuclear_dipole()

Compute the nuclear electric dipole.

Returns

Tuple[float, float, float] – x, y, and z components of nuclear electric dipole.

compute_one_electron_operator(*oper*, *origin=(0, 0, 0)*)

Compute a one-electron fermionic operator in atomic units.

For operators with origin-dependent expectation values, origin can be specified. oper values:

- 'kin' - kinetic energy.
- 'nuc' - nucleus-electron attraction energy.
- 'hcore' - one-electron hamiltonian.
- 'ovlp' - one-electron overlap.
- 'r' - electronic first moment (x, y, z).
- 'rr' - electronic second moment (xx, xy, xz, yx, yy, yz, zx, zy, zz).
- 'dm' - electronic dipole moment (x, y, z).

Parameters

- **oper** (str) – Key to specify the operator.

- **origin** (tuple, default: (0, 0, 0)) – Coordinate position of the origin.

Returns

Union[*FermionOperator*, List[*FermionOperator*]] – One electron operators.

extract_point_group_information(*reduce_infinite_point_groups=True*)

Returns point group information.

Parameters

reduce_infinite_point_groups – Reduce infinite point groups, e.g., Coov -> C2v

Returns

Tuple[str, List[str]] – Point group symmetry, list of orbital irreps.

Examples

```
>>> from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF
>>> driver = ChemistryDriverPySCFMolecularRHF(
...     geometry='H 0 0 0; H 0 0 0.75',
...     basis='sto3g',
...     point_group_symmetry=True,
... )
>>> ham, space, state = driver.get_system()
>>> driver.extract_point_group_information()
('D2h', ['Ag', 'Ag', 'B1u', 'B1u'])
```

---

classmethod from_mf(*mf*, *frozen=None*)

Initialize driver from a PySCF mean-field object.

Parameters

- **mf** (SCF) – PySCF mean-field object
- **frozen** (Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]], None], default: None) – Frozen core specified as either list or callable.

Returns

ChemistryDriverPySCFGammaROHF – PySCF driver.

property frozen: List[int] | List[List[int]]

Return the frozen orbital information.

Return type

Union[List[int], List[List[int]]]

generate_report()

Generate report in a hierarchical dictionary format.

PySCF attributes such as mo_coeff are exported if the SCF is converged.

Returns

Dict[str, Any] – Attributes of the internal PySCF mean-field object.

get_cube_density(*density_matrix*, *cube_resolution=0.25*)

Output Gaussian Cube contents for density.

Parameters

- **density_matrix** (ndarray) – One-body reduced density matrix. 2D array for RHF/ROHF, 3D array for UHF.
- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.

Returns
str – Cube file formatted string.

get_cube_orbitals (*cube_resolution=0.25*, *mo_coeff=None*, *orbital_indices=None*)

Output Gaussian Cube file contents for orbitals.

Parameters

- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.
- **mo_coeff** (Optional[array], default: None) – Molecular orbital coefficients for the orbitals to be visualised.
- **orbital_indices** (Optional[List], default: None) – Indices of the molecular orbitals of interest. If None, all orbitals are returned.

Returns
List[str] – List of cube file formatted strings. For UHF, the spinorbitals are returned as [mo1_alpha, mo1_beta, mo2_alpha, mo2_beta...].

get_excitation_amplitudes (*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*)

Get the (guess) parameters as excitation amplitudes from a classical calculation.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.

Returns
SymbolDict – Excitation parameters.

get_excitation_operators (*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*, *antihermitian=True*)

Get the excitation operators and (guess) parameters.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.

- **antihermitian** (bool, default: True) – Returns anti-Hermitian operators T - T^ if specified.

Returns
FermionOperatorList – Excitation operator list.

get_lowdin_system(*method='lowdin'*)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters
method (str, default: "lowdin") – Method passed to PySCF's orth.orth_ao().

Returns
Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*], *FermionSpace*, Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*]] – Fermion Hamiltonian, Fock space, Fock state.

get_madelung_constant()

Return Madelung constant for Gamma-point calculations.

Returns
float – Madelung constant contribution to the energy.

get_mulliken_pop()

Interface for Mulliken population analysis of PySCF.

Returns
Tuple[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]] – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction(*rdms*)

Compute the Strongly contracted NEVPT2 correction to the energy from the provided density matrices.

Parameters
rdms (Tuple) – A tuple of reduced density matrices as PDMs in PySCF's un-reordered format as (pdm1, …, rdm4).

Returns
The NEVPT2 correction to the energy.

get_rdm1_ccsd()

Reduced one-body density matrix in the AO basis from CCSD.

Returns
Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*] – One-body reduced density matrix.

get_rdm2_ccsd()

Reduced two-body density matrix in the AO basis with CCSD.

Returns
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Two-body reduced density matrix.

Note: This object will be replaced with an RDM2 class not to return a raw 4D tensor.

---

get_system (*symmetry=1*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Parameters

symmetry (Union[str, int], default: 1) – Code to specify target symmetry for storage of two-body integrals. Uses the same convention as PySCF. Currently, supports s1, s4 and s8 index permutation symmetries.

> **Warning:** For unrestricted integral operators, the aabb and bbaa two-body integrals cannot be compacted with s8 symmetry. If s8 symmetry is requested, the aaaa and bbbb will be stored with s8 symmetry, while the aabb and bbaa integrals will be stored with s4 symmetry.

Returns

Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*, *ChemistryRestrictedIntegralOperatorCompact*, *ChemistryUnrestrictedIntegralOperatorCompact*], *FermionSpace*, *FermionState*] – Fermion Hamiltonian, Fock space, Fock state.

get_system_ao (*run_hf=True*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Output hamiltonian operator will be of type *PySCFChemistryRestrictedIntegralOperator* or *PySCFChemistryUnrestrictedIntegralOperator*, which store the underlying atomic orbitals.

Parameters

run_hf (bool, default: True) – If True, a Hartree-Fock calculation for the system will be executed, and the Fock space and HF state are returned with the hamiltonian operator. If False, no calculation is performed and the Fock space and HF state are returned as None. If the PySCF mean-field object has already been converged prior to calling `get_system_ao()`, behaviour will follow `run_hf=True`.

Returns

Tuple[Union[*PySCFChemistryRestrictedIntegralOperator*, *PySCFChemistryUnrestrictedIntegralOperator*], Optional[*FermionSpace*], Optional[*FermionState*]] – Fermion Hamiltonian, Fock space, Fock state.

property mf_energy: float

Return the total mean-field energy.

Return type

float

property mf_type: str

Return the mean-field type as a string (e.g. “RHF”).

Returns

str – Mean-field type name.

property n_electron: int

Number of electrons in the active space.

Returns

int – Number of electrons.

property n_orb: int

Number of spatial orbitals.

Returns
int – Number of spatial orbitals.

print_json_report(**args*, ***kwargs*)
Prints report in json format.

run_casci(***kwargs*)
Calculate the CASCI energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the mcscf.CASCI object.

Returns
float – CASCI energy.

run_ccsd(***kwargs*)
Calculate the CCSD energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the cc.CCSD object.

Returns
float – CCSD energy

run_hf()
Calculate the HF energy.

Returns
float – HF energy

run_mp2(***kwargs*)
Calculate the MP2 energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the mp.MP2 object.

Returns
float – MP2 energy

class ChemistryDriverPySCFIntegrals(*constant*, *h1*, *h2*, *n_electron*, *multiplicity=1*, *initial_dm=None*, *frozen=None*, *transf=None*)

PySCF chemistry driver with electronic integrals as input.

Compatible with restricted spin calculations only.

Parameters

- **constant** (float) – Constant contribution to the Hamiltonian.
- **h1** (ndarray) – One-body integrals.
- **h2** (ndarray) – Two-body integrals.
- **n_electron** (int) – Number of electrons
- **multiplicity** (int, default: 1) – Spin multiplicity
- **initial_dm** (Optional[ndarray], default: None) – Initial density matrix.
- **frozen** (Union[List[int], Callable[[SCF], List[int]], None], default: None) – List of frozen orbitals
- **transf** (Union[Callable[[array], array], Transf, None], default: None) – Orbital transformer.

classmethod from_integral_operator(*hamiltonian_operator*, *n_electron*, **args*, ***kwargs*)

Generate PySCF driver from a `ChemistryRestrictedIntegralOperator`.

Parameters

- **Hamiltonian** – Integral operator object from which integrals are taken.
- **n_electron** (`int`) – Number of electrons.
- ***args** – Arguments to be passed to the *ChemistryDriverPySCFIntegrals* constructor.
- ****kwargs** – Keyword arguments to be passed to the *ChemistryDriverPySCFIntegrals* constructor.
- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) –

Returns

`BasePySCFDriverIntegrals` – PySCF driver.

property frozen: List[int] | List[List[int]]

Return the frozen orbital information.

Return type

`Union[List[int], List[List[int]]]`

generate_report()

Generate report in a hierarchical dictionary format.

PySCF attributes such as `mo_coeff` are exported if the SCF is converged.

Returns

`Dict[str, Any]` – Attributes of the internal PySCF mean-field object.

get_excitation_amplitudes(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*)

Get the (guess) parameters as excitation amplitudes from a classical calculation.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (`float`, default: `0.0`) – Threshold of the amplitude to include the excitation operator.
- **t1** (`Optional[ndarray]`, default: `None`) – Guess 1e excitation amplitudes.
- **t2** (`Optional[ndarray]`, default: `None`) – Guess 2e excitation amplitudes.
- **compact** (`bool`, default: `False`) – Enable compact form of excitation for the restrected wavefunctions.

Returns

SymbolDict – Excitation parameters.

get_excitation_operators(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*, *antihermitian=True*)

Get the excitation operators and (guess) parameters.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (`float`, default: `0.0`) – Threshold of the amplitude to include the excitation operator.

- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.
- **antihermitian** (bool, default: True) – Returns anti-Hermitian operators T - T^ if specified.

Returns
FermionOperatorList – Excitation operator list.

get_lowdin_system(*method='lowdin'*)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters
method (str, default: "lowdin") – Method passed to PySCF's orth.orth_ao().

Returns
Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*], *FermionSpace*, Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*]] – Fermion Hamiltonian, Fock space, Fock state.

get_one_body_rdm()

Compute one-body restricted densoty matrix.

Returns
RestrictedOneBodyRDM – One-body RDM object from the mean-field calculation.

Raises
RunTimeError – If the SCF object is not of type RHF or ROHF.

get_rdm1_ccsd()

Reduced one-body density matrix in the AO basis from CCSD.

Returns
Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*] – One-body reduced density matrix.

get_rdm2_ccsd()

Reduced two-body density matrix in the AO basis with CCSD.

Returns
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Two-body reduced density matrix.

Note: This object will be replaced with an RDM2 class not to return a raw 4D tensor.

---

get_system(*symmetry=1*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Parameters
symmetry (Union[str, int], default: 1) – Code to specify target symmetry for storage of two-body integrals. Uses the same convention as PySCF. Currently, supports s1, s4 and s8 index permutation symmetries.

Warning: For unrestricted integral operators, the aabb and bbaa two-body integrals cannot be compacted with s8 symmetry. If s8 symmetry is requested, the aaaa and bbbb will be stored with s8 symmetry, while the aabb and bbaa integrals will be stored with s4 symmetry.

Returns
Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*, *ChemistryRestrictedIntegralOperatorCompact*, *ChemistryUnrestrictedIntegralOperatorCompact*], *FermionSpace*, *FermionState*] – Fermion Hamiltonian, Fock space, Fock state.

get_system_ao(*run_hf=True*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Output hamiltonian operator will be of type *PySCFChemistryRestrictedIntegralOperator* or *PySCFChemistryUnrestrictedIntegralOperator*, which store the underlying atomic orbitals.

Parameters
run_hf (bool, default: True) – If True, a Hartree-Fock calculation for the system will be executed, and the Fock space and HF state are returned with the hamiltonian operator. If False, no calculation is performed and the Fock space and HF state are returned as None. If the PySCF mean-field object has already been converged prior to calling get_system_ao(), behaviour will follow run_hf=True.

Returns
Tuple[Union[*PySCFChemistryRestrictedIntegralOperator*, *PySCFChemistryUnrestrictedIntegralOperator*], Optional[*FermionSpace*], Optional[*FermionState*]] – Fermion Hamiltonian, Fock space, Fock state.

property mf_energy: float

Return the total mean-field energy.

Return type
float

property mf_type: str

Return the mean-field type as a string (e.g. "RHF").

Returns
str – Mean-field type name.

property mo_coeff: ndarray[Any, dtype[ScalarType]]

Return the molecular orbital coefficient matrix.

Returns
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – MO coefficients.

property n_electron: int

Number of electrons in the active space.

Returns
int – Number of electrons.

property n_orb: int

Number of spatial orbitals.

Returns
int – Number of spatial orbitals.

print_json_report(**args*, ***kwargs*)

Prints report in json format.

run_casci(***kwargs*)

Calculate the CASCI energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the `mcscf.CASCI` object.

Returns

`float` – CASCI energy.

run_ccsd(***kwargs*)

Calculate the CCSD energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the `cc.CCSD` object.

Returns

`float` – CCSD energy

run_hf()

Calculate the HF energy.

Returns

`float` – HF energy

run_mp2(***kwargs*)

Calculate the MP2 energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the `mp.MP2` object.

Returns

`float` – MP2 energy

class ChemistryDriverPySCFMolecularRHF(*geometry=None*, *zmatrix=None*, *basis=None*, *ecp=None*, *charge=0*, *frozen=None*, *transf=None*, *verbose=0*, *output=None*, *point_group_symmetry=False*)

PySCF driver for molecular RHF calculations.

Parameters

- **geometry** (`Union[List, str, Geometry, None]`, default: `None`) – Molecular geometry.
- **zmatrix** (`Optional[str]`, default: `None`) – Z matrix representation of molecular geometry (Used only if `geometry` is not specified).
- **basis** (`Optional[Any]`, default: `None`) – Atomic basis set valid for `Mole` class.
- **ecp** (`Optional[Any]`, default: `None`) – Effective core potentials.
- **charge** (`int`, default: `0`) – Total charge.
- **frozen** (`Union[List[int], Callable[[RHF], int], None]`, default: `None`) – Frozen orbital information.
- **transf** (`Optional[Transf]`, default: `None`) – Orbital transformer.
- **verbose** (`int`, default: `0`) – Control PySCF verbosity.
- **output** (`Optional[str]`, default: `None`) – Specify log file name. If None, logs are printed to STDOUT.

- **point_group_symmetry** (bool, default: False) – Enable point group symmetry.

compute_nuclear_dipole()

Compute the nuclear electric dipole.

Returns

Tuple[float, float, float] – x, y, and z components of nuclear electric dipole.

compute_one_electron_operator(*oper*, *origin=(0, 0, 0)*)

Compute a one-electron fermionic operator in atomic units.

For operators with origin-dependent expectation values, origin can be specified. oper values:

- 'kin' - kinetic energy.
- 'nuc' - nucleus-electron attraction energy.
- 'hcore' - one-electron hamiltonian.
- 'ovlp' - one-electron overlap.
- 'r' - electronic first moment (x, y, z).
- 'rr' - electronic second moment (xx, xy, xz, yx, yy, yz, zx, zy, zz).
- 'dm' - electronic dipole moment (x, y, z).

Parameters

- **oper** (str) – Key to specify the operator.
- **origin** (tuple, default: (0, 0, 0)) – Coordinate position of the origin.

Returns

Union[*FermionOperator*, List[*FermionOperator*]] – One electron operators.

extract_point_group_information(*reduce_infinite_point_groups=True*)

Returns point group information.

Parameters

reduce_infinite_point_groups – Reduce infinite point groups, e.g., Coov -> C2v

Returns

Tuple[str, List[str]] – Point group symmetry, list of orbital irreps.

Examples

```
>>> from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF
>>> driver = ChemistryDriverPySCFMolecularRHF(
...     geometry='H 0 0 0; H 0 0 0.75',
...     basis='sto3g',
...     point_group_symmetry=True,
... )
>>> ham, space, state = driver.get_system()
>>> driver.extract_point_group_information()
('D2h', ['Ag', 'Ag', 'B1u', 'B1u'])
```

---

classmethod from_mf(*mf*, *frozen=None*, *transf=None*)

Initialize driver from a PySCF mean-field object.

Parameters

- **mf** (SCF) – PySCF mean-field object.
- **frozen** (Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]], None], default: None) – Frozen core specified as either list or callable.
- **transf** (Union[Callable[[ndarray], ndarray], Transf, None], default: None) – Orbital transformation function.

Returns
ChemistryDriverPySCFMolecularRHF – PySCF driver.

property frozen: List[int] | List[List[int]]

Return the frozen orbital information.

Return type
Union[List[int], List[List[int]]]

generate_report()

Generate report in a hierarchical dictionary format.

PySCF attributes such as mo_coeff are exported if the SCF is converged.

Returns
Dict[str, Any] – Attributes of the internal PySCF mean-field object.

get_cube_density(*density_matrix*, *cube_resolution=0.25*)

Output Gaussian Cube contents for density.

Parameters

- **density_matrix** (ndarray) – One-body reduced density matrix. 2D array for RHF/ROHF, 3D array for UHF.
- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.

Returns
str – Cube file formatted string.

get_cube_orbitals(*cube_resolution=0.25*, *mo_coeff=None*, *orbital_indices=None*)

Output Gaussian Cube file contents for orbitals.

Parameters

- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.
- **mo_coeff** (Optional[array], default: None) – Molecular orbital coefficients for the orbitals to be visualised.
- **orbital_indices** (Optional[List], default: None) – Indices of the molecular orbitals of interest. If None, all orbitals are returned.

Returns
List[str] – List of cube file formatted strings. For UHF, the spinorbitals are returned as [mo1_alpha, mo1_beta, mo2_alpha, mo2_beta...].

get_excitation_amplitudes(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*)

Get the (guess) parameters as excitation amplitudes from a classical calculation.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.

- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.

Returns
SymbolDict – Excitation parameters.

get_excitation_operators (*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*, *antihermitian=True*)

Get the excitation operators and (guess) parameters.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.
- **antihermitian** (bool, default: True) – Returns anti-Hermitian operators T - T^ if specified.

Returns
FermionOperatorList – Excitation operator list.

get_lowdin_system (*method='lowdin'*)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters
method (str, default: "lowdin") – Method passed to PySCF's orth.orth_ao().

Returns
Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*], *FermionSpace*, Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*]] – Fermion Hamiltonian, Fock space, Fock state.

get_mulliken_pop ()

Interface for Mulliken population analysis of PySCF.

Returns
Tuple[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]] – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction (*rdms*)

Compute the Strongly contracted NEVPT2 correction to the energy from the provided density matrices.

Parameters
rdms (Tuple) – A tuple of reduced density matrices as PDMs in PySCF's un-reordered format as (pdm1, …, rdm4).

Returns
The NEVPT2 correction to the energy.

get_rdm1_ccsd()

Reduced one-body density matrix in the AO basis from CCSD.

Returns
Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*] – One-body reduced density matrix.

get_rdm2_ccsd()

Reduced two-body density matrix in the AO basis with CCSD.

Returns
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Two-body reduced density matrix.

Note: This object will be replaced with an RDM2 class not to return a raw 4D tensor.

---

get_system(*symmetry=1*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Parameters
symmetry (Union[str, int], default: 1) – Code to specify target symmetry for storage of two-body integrals. Uses the same convention as PySCF. Currently, supports s1, s4 and s8 index permutation symmetries.

Warning: For unrestricted integral operators, the aabb and bbaa two-body integrals cannot be compacted with s8 symmetry. If s8 symmetry is requested, the aaaa and bbbb will be stored with s8 symmetry, while the aabb and bbaa integrals will be stored with s4 symmetry.

Returns
Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*, *ChemistryRestrictedIntegralOperatorCompact*, *ChemistryUnrestrictedIntegralOperatorCompact*], *FermionSpace*, *FermionState*] – Fermion Hamiltonian, Fock space, Fock state.

get_system_ao(*run_hf=True*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Output hamiltonian operator will be of type *PySCFChemistryRestrictedIntegralOperator* or *PySCFChemistryUnrestrictedIntegralOperator*, which store the underlying atomic orbitals.

Parameters
run_hf (bool, default: True) – If True, a Hartree-Fock calculation for the system will be executed, and the Fock space and HF state are returned with the hamiltonian operator. If False, no calculation is performed and the Fock space and HF state are returned as None. If the PySCF mean-field object has already been converged prior to calling get_system_ao(), behaviour will follow run_hf=True.

Returns
Tuple[Union[*PySCFChemistryRestrictedIntegralOperator*, *PySCFChemistryUnrestrictedIntegralOperator*], Optional[*FermionSpace*], Optional[*FermionState*]] – Fermion Hamiltonian, Fock space, Fock state.

property mf_energy: float

Return the total mean-field energy.

Return type
float

property mf_type: str

Return the mean-field type as a string (e.g. "RHF").

Returns
str – Mean-field type name.

property n_electron: int

Number of electrons in the active space.

Returns
int – Number of electrons.

property n_orb: int

Number of spatial orbitals.

Returns
int – Number of spatial orbitals.

print_json_report(**args*, ***kwargs*)

Prints report in json format.

run_casci(***kwargs*)

Calculate the CASCI energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the mcscf.CASCI object.

Returns
float – CASCI energy.

run_ccsd(***kwargs*)

Calculate the CCSD energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the cc.CCSD object.

Returns
float – CCSD energy

run_hf()

Calculate the HF energy.

Returns
float – HF energy

run_mp2(***kwargs*)

Calculate the MP2 energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the mp.MP2 object.

Returns
float – MP2 energy

class ChemistryDriverPySCFMolecularRHFQMMMCOSMO(*geometry=None*, *zmatrix=None*, *basis=None*, *ecp=None*, *charge=0*, *frozen=None*, *transf=None*, *do_qmmm=None*, *mm_charges=None*, *mm_geometry=None*, *do_mm_coulomb=None*, *do_cosmo=None*, *verbose=0*, *output=None*, *point_group_symmetry=False*)

PySCF driver for molecular RHF, with QMMM and COSMO calculations.

QMMM: Quantum Mechanics - Molecular mechanics. COSMO: Conductor-like Screening Model.

Parameters

- **geometry** (Union[List, str, None], default: None) – Molecular geometry.
- **zmatrix** (Optional[str], default: None) – Z matrix representation of molecular geometry (Used only if geometry is not specified).
- **basis** (Optional[Any], default: None) – Atomic basis set valid for Mole class.
- **ecp** (Optional[Any], default: None) – Effective core potentials.
- **charge** (int, default: 0) – Total charge.
- **frozen** (Optional[Any], default: None) – Frozen orbital information.
- **transf** (Optional[Transf], default: None) – Orbital transformer.
- **do_qmmm** (Optional[bool], default: None) – If True, do QMMM embedding. MM geometry and charges are required.
- **mm_charges** (Optional[Any], default: None) – Point charge values for MM region.
- **mm_geometry** (Optional[Any], default: None) – Geometry of MM region.
- **do_mm_coulomb** (Optional[bool], default: None) – If True, calculate MM-MM Coulomb interaction energy. Needs to be added as a constant to the final energy: <\| H \|> + self.e_mm_coulomb
- **do_cosmo** (Optional[bool], default: None) – If True, adds implicit water (COSMO) to the mean-field calculation. get_system_legacy() returns H + v_solvent as fermion operator. The resulting final energy needs to be corrected as <\|H + v_solvet\|> + self.cosmo_correction. Where self.cosmo_correction = E_cosmo - Tr[v_solvent * rdm1]. If do_cosmo=True, self.cosmo_correction is defined after self._run_hf().
- **verbose** (int, default: 0) – Control SCF verbosity.
- **output** (Optional[str], default: None) – Specify log file name. If None, logs are printed to STDOUT.
- **point_group_symmetry** (bool, default: False) – Enable point group symmetry.

static build_mm_charges(*mm_charges*, *mm_geometry*)

Puts InQuanto MM charge dict into PySCF-friendly format.

Orders mm charges according to mm_geometry.

Parameters

- **mm_geometry** (List) – Geometry of MM region in format ['Atom1', [x, y, z]], 'Atom2', [x, y, z]], ...].
- **mm_charges** (Union[Dict, List, float]) – Dict of charges, or float if all charges are the same. If float, all output charges will all have this value. If it is already a list, do nothing.

Returns
List – List of mm charges ordered the same as mm_geometry.

compute_nuclear_dipole()
Compute the nuclear electric dipole.

Returns
Tuple[float, float, float] – x, y, and z components of nuclear electric dipole.

compute_one_electron_operator(*oper*, *origin=(0, 0, 0)*)
Compute a one-electron fermionic operator in atomic units.

For operators with origin-dependent expectation values, origin can be specified. oper values:

- 'kin' - kinetic energy.
- 'nuc' - nucleus-electron attraction energy.
- 'hcore' - one-electron hamiltonian.
- 'ovlp' - one-electron overlap.
- 'r' - electronic first moment (x, y, z).
- 'rr' - electronic second moment (xx, xy, xz, yx, yy, yz, zx, zy, zz).
- 'dm' - electronic dipole moment (x, y, z).

Parameters
- **oper** (str) – Key to specify the operator.
- **origin** (tuple, default: (0, 0, 0)) – Coordinate position of the origin.

Returns
Union[*FermionOperator*, List[*FermionOperator*]] – One electron operators.

extract_point_group_information(*reduce_infinite_point_groups=True*)
Returns point group information.

Parameters
reduce_infinite_point_groups – Reduce infinite point groups, e.g., Coov -> C2v

Returns
Tuple[str, List[str]] – Point group symmetry, list of orbital irreps.

Examples

```
>>> from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF
>>> driver = ChemistryDriverPySCFMolecularRHF(
...     geometry='H 0 0 0; H 0 0 0.75',
...     basis='sto3g',
...     point_group_symmetry=True,
... )
>>> ham, space, state = driver.get_system()
>>> driver.extract_point_group_information()
('D2h', ['Ag', 'Ag', 'B1u', 'B1u'])
```

---

classmethod from_mf(*mf*, *frozen=None*, *transf=None*)

Initialize driver from a PySCF mean-field object.

Parameters

- **mf** (SCF) – PySCF mean-field object.
- **frozen** (Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]], None], default: None) – Frozen core specified as either list or callable.
- **transf** (Union[Callable[[ndarray], ndarray], Transf, None], default: None) – Orbital transformation function.

Returns

ChemistryDriverPySCFMolecularRHFQMMMCOSMO – PySCF driver.

property frozen: List[int] | List[List[int]]

Return the frozen orbital information.

Return type

Union[List[int], List[List[int]]]

generate_report()

Generate report in a hierarchical dictionary format.

PySCF attributes such as mo_coeff are exported if the SCF is converged.

Returns

Dict[str, Any] – Attributes of the internal PySCF mean-field object.

get_cube_density(*density_matrix*, *cube_resolution=0.25*)

Output Gaussian Cube contents for density.

Parameters

- **density_matrix** (ndarray) – One-body reduced density matrix. 2D array for RHF/ROHF, 3D array for UHF.
- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.

Returns

str – Cube file formatted string.

get_cube_orbitals(*cube_resolution=0.25*, *mo_coeff=None*, *orbital_indices=None*)

Output Gaussian Cube file contents for orbitals.

Parameters

- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.
- **mo_coeff** (Optional[array], default: None) – Molecular orbital coefficients for the orbitals to be visualised.
- **orbital_indices** (Optional[List], default: None) – Indices of the molecular orbitals of interest. If None, all orbitals are returned.

Returns

List[str] – List of cube file formatted strings. For UHF, the spinorbitals are returned as [mo1_alpha, mo1_beta, mo2_alpha, mo2_beta...].

get_excitation_amplitudes (*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*)

Get the (guess) parameters as excitation amplitudes from a classical calculation.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.

Returns

SymbolDict – Excitation parameters.

get_excitation_operators (*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*, *antihermitian=True*)

Get the excitation operators and (guess) parameters.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.
- **antihermitian** (bool, default: True) – Returns anti-Hermitian operators T - T^ if specified.

Returns

FermionOperatorList – Excitation operator list.

get_lowdin_system (*method='lowdin'*)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

method (str, default: "lowdin") – Method passed to PySCF's orth.orth_ao().

Returns

Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*], *FermionSpace*, Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*]] – Fermion Hamiltonian, Fock space, Fock state.

static get_mm_coulomb (*mm_charges_pyscf*, *mm_geometry*, *unit='Ha'*)

Get the Coulomb interaction energy between MM particles.

Parameters

- **mm_charges_pyscf** (Any) – Charges of MM region, in same order as mm_geometry.

- **mm_geometry** (Any) – Geometry of MM region in format ['Atom1', [x, y, z]], 'Atom2', [x, y, z]], ...].
- **unit** (str, default: "Ha") – Energy units of Coulomb interaction energy. Default is Hartree.

Returns
Electrostatic Coulomb energy.

get_mulliken_pop()

Interface for Mulliken population analysis of PySCF.

Returns
Tuple[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]] – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction(*rdms*)

Compute the Strongly contracted NEVPT2 correction to the energy from the provided density matrices.

Parameters
rdms (Tuple) – A tuple of reduced density matrices as PDMs in PySCF's un-reordered format as (pdm1, …, rdm4).

Returns
The NEVPT2 correction to the energy.

get_rdm1_ccsd()

Reduced one-body density matrix in the AO basis from CCSD.

Returns
Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*] – One-body reduced density matrix.

get_rdm2_ccsd()

Reduced two-body density matrix in the AO basis with CCSD.

Returns
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Two-body reduced density matrix.

Note: This object will be replaced with an RDM2 class not to return a raw 4D tensor.

---

get_system(*symmetry=1*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Parameters
symmetry (Union[str, int], default: 1) – Code to specify target symmetry for storage of two-body integrals. Uses the same convention as PySCF. Currently, supports s1, s4 and s8 index permutation symmetries.

Warning: For unrestricted integral operators, the aabb and bbaa two-body integrals cannot be compacted with s8 symmetry. If s8 symmetry is requested, the aaaa and bbbb will be stored with s8 symmetry, while the aabb and bbaa integrals will be stored with s4 symmetry.

Returns
Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*, *ChemistryRestrictedIntegralOperatorCompact*, *ChemistryUnrestrictedIntegralOperatorCompact*], *FermionSpace*, *FermionState*] – Fermion Hamiltonian, Fock space, Fock state.

get_system_ao(*run_hf=True*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Output hamiltonian operator will be of type *PySCFChemistryRestrictedIntegralOperator* or *PySCFChemistryUnrestrictedIntegralOperator*, which store the underlying atomic orbitals.

Parameters
run_hf (bool, default: True) – If True, a Hartree-Fock calculation for the system will be executed, and the Fock space and HF state are returned with the hamiltonian operator. If False, no calculation is performed and the Fock space and HF state are returned as None. If the PySCF mean-field object has already been converged prior to calling `get_system_ao()`, behaviour will follow `run_hf=True`.

Returns
Tuple[Union[*PySCFChemistryRestrictedIntegralOperator*, *PySCFChemistryUnrestrictedIntegralOperator*], Optional[*FermionSpace*], Optional[*FermionState*]] – Fermion Hamiltonian, Fock space, Fock state.

property mf_energy: float

Return the total mean-field energy.

Return type
float

property mf_type: str

Return the mean-field type as a string (e.g. “RHF”).

Returns
str – Mean-field type name.

property n_electron: int

Number of electrons in the active space.

Returns
int – Number of electrons.

property n_orb: int

Number of spatial orbitals.

Returns
int – Number of spatial orbitals.

print_json_report(**args*, ***kwargs*)

Prints report in json format.

run_casci(***kwargs*)

Calculate the CASCI energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the `mcscf.CASCI` object.

Returns
float – CASCI energy.

run_ccsd(***kwargs*)

Calculate the CCSD energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the `cc.CCSD` object.

Returns

`float` – CCSD energy

run_hf()

Calculate the HF energy.

Returns

`float` – HF energy

run_mp2(***kwargs*)

Calculate the MP2 energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the `mp.MP2` object.

Returns

`float` – MP2 energy

class ChemistryDriverPySCFMolecularROHF(*geometry=None*, *zmatrix=None*, *basis=None*, *ecp=None*, *charge=0*, *multiplicity=1*, *frozen=None*, *transf=None*, *verbose=0*, *output=None*, *point_group_symmetry=False*)

PySCF driver for molecular ROHF calculations.

Parameters

- **geometry** (`Union[List, str, Geometry, None]`, default: `None`) – Molecular geometry.
- **zmatrix** (`Optional[str]`, default: `None`) – Z matrix representation of molecular geometry (Used only if `geometry` is not specified).
- **basis** (`Optional[Any]`, default: `None`) – Atomic basis set valid for `Mole` class.
- **ecp** (`Optional[Any]`, default: `None`) – Effective core potentials.
- **charge** (`int`, default: `0`) – Total charge.
- **multiplicity** (`int`, default: `1`) – Spin multiplicity, 2S+1.
- **frozen** (`Optional[Any]`, default: `None`) – Frozen orbital information.
- **transf** (`Optional[Transf]`, default: `None`) – Orbital transformer.
- **verbose** (`int`, default: `0`) – Control PySCF verbosity.
- **output** (`Optional[str]`, default: `None`) – Specify log file name. If None, logs are printed to STDOUT.
- **point_group_symmetry** (`bool`, default: `False`) – Enable point group symmetry.

compute_nuclear_dipole()

Compute the nuclear electric dipole.

Returns

`Tuple[float, float, float]` – x, y, and z components of nuclear electric dipole.

compute_one_electron_operator(*oper*, *origin=(0, 0, 0)*)

Compute a one-electron fermionic operator in atomic units.

For operators with origin-dependent expectation values, origin can be specified. oper values:

- 'kin' - kinetic energy.
- 'nuc' - nucleus-electron attraction energy.
- 'hcore' - one-electron hamiltonian.
- 'ovlp' - one-electron overlap.
- 'r' - electronic first moment (x, y, z).
- 'rr' - electronic second moment (xx, xy, xz, yx, yy, yz, zx, zy, zz).
- 'dm' - electronic dipole moment (x, y, z).

Parameters

- **oper** (`str`) – Key to specify the operator.
- **origin** (`tuple`, default: `(0, 0, 0)`) – Coordinate position of the origin.

Returns

`Union`[*FermionOperator*, `List`[*FermionOperator*]] – One electron operators.

extract_point_group_information(*reduce_infinite_point_groups=True*)

Returns point group information.

Parameters

reduce_infinite_point_groups – Reduce infinite point groups, e.g., Coov -> C2v

Returns

`Tuple`[`str`, `List`[`str`]] – Point group symmetry, list of orbital irreps.

Examples

```
>>> from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF
>>> driver = ChemistryDriverPySCFMolecularRHF(
...     geometry='H 0 0 0; H 0 0 0.75',
...     basis='sto3g',
...     point_group_symmetry=True,
... )
>>> ham, space, state = driver.get_system()
>>> driver.extract_point_group_information()
('D2h', ['Ag', 'Ag', 'B1u', 'B1u'])
```

---

classmethod from_mf(*mf*, *frozen=None*, *transf=None*)

Initialize driver from a PySCF mean-field object.

Parameters

- **mf** (`SCF`) – PySCF mean-field object.
- **frozen** (`Union`[`List`[`int`], `List`[`List`[`int`]], `Callable`[[`SCF`], `Union`[`List`[`int`], `List`[`List`[`int`]]]], `None`], default: `None`) – Frozen core specified as either list or callable.

- **transf** (Union[Callable[[ndarray], ndarray], Transf, None], default: None) – Orbital transformation function.

Returns
ChemistryDriverPySCFMolecularROHF – PySCF driver.

property frozen: List[int] | List[List[int]]

Return the frozen orbital information.

Return type
Union[List[int], List[List[int]]]

generate_report()

Generate report in a hierarchical dictionary format.

PySCF attributes such as mo_coeff are exported if the SCF is converged.

Returns
Dict[str, Any] – Attributes of the internal PySCF mean-field object.

get_cube_density(*density_matrix*, *cube_resolution=0.25*)

Output Gaussian Cube contents for density.

Parameters

- **density_matrix** (ndarray) – One-body reduced density matrix. 2D array for RHF/ROHF, 3D array for UHF.
- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.

Returns
str – Cube file formatted string.

get_cube_orbitals(*cube_resolution=0.25*, *mo_coeff=None*, *orbital_indices=None*)

Output Gaussian Cube file contents for orbitals.

Parameters

- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.
- **mo_coeff** (Optional[array], default: None) – Molecular orbital coefficients for the orbitals to be visualised.
- **orbital_indices** (Optional[List], default: None) – Indices of the molecular orbitals of interest. If None, all orbitals are returned.

Returns
List[str] – List of cube file formatted strings. For UHF, the spinorbitals are returned as [mo1_alpha, mo1_beta, mo2_alpha, mo2_beta...].

get_excitation_amplitudes(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*)

Get the (guess) parameters as excitation amplitudes from a classical calculation.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.

- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restricted wavefunctions.

Returns
SymbolDict – Excitation parameters.

get_excitation_operators (*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*, *antihermitian=True*)

Get the excitation operators and (guess) parameters.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restricted wavefunctions.
- **antihermitian** (bool, default: True) – Returns anti-Hermitian operators T - T^ if specified.

Returns
FermionOperatorList – Excitation operator list.

get_lowdin_system (*method='lowdin'*)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters
method (str, default: "lowdin") – Method passed to PySCF's orth.orth_ao().

Returns
Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*], *FermionSpace*, Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*]] – Fermion Hamiltonian, Fock space, Fock state.

get_mulliken_pop ()

Interface for Mulliken population analysis of PySCF.

Returns
Tuple[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]] – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction (*rdms*)

Compute the Strongly contracted NEVPT2 correction to the energy from the provided density matrices.

Parameters
rdms (Tuple) – A tuple of reduced density matrices as PDMs in PySCF's un-reordered format as (pdm1, …, rdm4).

Returns
The NEVPT2 correction to the energy.

get_rdm1_ccsd()

Reduced one-body density matrix in the AO basis from CCSD.

Returns

Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*] – One-body reduced density matrix.

get_rdm2_ccsd()

Reduced two-body density matrix in the AO basis with CCSD.

Returns

ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Two-body reduced density matrix.

Note: This object will be replaced with an RDM2 class not to return a raw 4D tensor.

---

get_system(*symmetry=1*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Parameters

symmetry (Union[str, int], default: 1) – Code to specify target symmetry for storage of two-body integrals. Uses the same convention as PySCF. Currently, supports s1, s4 and s8 index permutation symmetries.

Warning: For unrestricted integral operators, the aabb and bbaa two-body integrals cannot be compacted with s8 symmetry. If s8 symmetry is requested, the aaaa and bbbb will be stored with s8 symmetry, while the aabb and bbaa integrals will be stored with s4 symmetry.

Returns

Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*, *ChemistryRestrictedIntegralOperatorCompact*, *ChemistryUnrestrictedIntegralOperatorCompact*], *FermionSpace*, *FermionState*] – Fermion Hamiltonian, Fock space, Fock state.

get_system_ao(*run_hf=True*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Output hamiltonian operator will be of type *PySCFChemistryRestrictedIntegralOperator* or *PySCFChemistryUnrestrictedIntegralOperator*, which store the underlying atomic orbitals.

Parameters

run_hf (bool, default: True) – If True, a Hartree-Fock calculation for the system will be executed, and the Fock space and HF state are returned with the hamiltonian operator. If False, no calculation is performed and the Fock space and HF state are returned as None. If the PySCF mean-field object has already been converged prior to calling get_system_ao(), behaviour will follow run_hf=True.

Returns

Tuple[Union[*PySCFChemistryRestrictedIntegralOperator*, *PySCFChemistryUnrestrictedIntegralOperator*], Optional[*FermionSpace*], Optional[*FermionState*]] – Fermion Hamiltonian, Fock space, Fock state.

property mf_energy: float

Return the total mean-field energy.

Return type
float

property mf_type: str

Return the mean-field type as a string (e.g. "RHF").

Returns
str – Mean-field type name.

property n_electron: int

Number of electrons in the active space.

Returns
int – Number of electrons.

property n_orb: int

Number of spatial orbitals.

Returns
int – Number of spatial orbitals.

print_json_report(**args*, ***kwargs*)

Prints report in json format.

run_casci(***kwargs*)

Calculate the CASCI energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the mcscf.CASCI object.

Returns
float – CASCI energy.

run_ccsd(***kwargs*)

Calculate the CCSD energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the cc.CCSD object.

Returns
float – CCSD energy

run_hf()

Calculate the HF energy.

Returns
float – HF energy

run_mp2(***kwargs*)

Calculate the MP2 energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the mp.MP2 object.

Returns
float – MP2 energy

class ChemistryDriverPySCFMolecularROHFQMMMCOSMO(*geometry=None*, *zmatrix=None*, *basis=None*, *ecp=None*, *charge=0*, *multiplicity=1*, *frozen=None*, *do_qmmm=None*, *mm_charges=None*, *mm_geometry=None*, *do_mm_coulomb=None*, *do_cosmo=None*, *verbose=0*, *output=None*, *point_group_symmetry=False*)

PySCF driver for molecular ROHF, with QMMM and COSMO calculations.

QMMM: Quantum Mechanics - Molecular mechanics. COSMO: Conductor-like Screening Model.

Parameters

- **geometry** (Union[List, str, None], default: None) – Molecular geometry.
- **zmatrix** (Optional[str], default: None) – Z matrix representation of molecular geometry (Used only if geometry is not specified).
- **basis** (Optional[Any], default: None) – Atomic basis set valid for Mole class.
- **ecp** (Optional[Any], default: None) – Effective core potentials.
- **charge** (int, default: 0) – Total charge.
- **multiplicity** (int, default: 1) – Spin multiplicity, 2S+1.
- **frozen** (Optional[Any], default: None) – Frozen orbital information.
- **do_qmmm** (Optional[bool], default: None) – If True, do QMMM embedding. MM geometry and charges are required.
- **mm_charges** (Optional[Any], default: None) – Point charge values for MM region.
- **mm_geometry** (Optional[Any], default: None) – Geometry of MM region.
- **do_mm_coulomb** (Optional[bool], default: None) – If True, calculate MM-MM Coulomb interaction energy. Needs to be added as a constant to the final energy: <\| H \|> + self.e_mm_coulomb
- **do_cosmo** (Optional[bool], default: None) – If True, adds implicit water (COSMO) to the mean-field calculation. get_system_legacy() returns H + v_solvent as fermion operator. The resulting final energy needs to be corrected as <\|H + v_solvet\|> + self.cosmo_correction. Where self.cosmo_correction = E_cosmo - Tr[v_solvent * rdm1]. If do_cosmo=True, self.cosmo_correction is defined after self._run_hf().
- **verbose** (int, default: 0) – Control SCF verbosity.
- **output** (Optional[str], default: None) – Specify log file name. If None, logs are printed to STDOUT.
- **point_group_symmetry** (bool, default: False) – Enable point group symmetry.

static build_mm_charges(*mm_charges*, *mm_geometry*)

Puts InQuanto MM charge dict into PySCF-friendly format.

Orders mm charges according to mm_geometry.

Parameters

- **mm_geometry** (List) – Geometry of MM region in format ['Atom1', [x, y, z]], 'Atom2', [x, y, z]], ...].
- **mm_charges** (Union[Dict, List, float]) – Dict of charges, or float if all charges are the same. If float, all output charges will all have this value. If it is already a list, do nothing.

Returns
List – List of mm charges ordered the same as mm_geometry.

compute_nuclear_dipole()

Compute the nuclear electric dipole.

Returns
Tuple[float, float, float] – x, y, and z components of nuclear electric dipole.

compute_one_electron_operator(*oper*, *origin=(0, 0, 0)*)

Compute a one-electron fermionic operator in atomic units.

For operators with origin-dependent expectation values, origin can be specified. oper values:

- 'kin' - kinetic energy.
- 'nuc' - nucleus-electron attraction energy.
- 'hcore' - one-electron hamiltonian.
- 'ovlp' - one-electron overlap.
- 'r' - electronic first moment (x, y, z).
- 'rr' - electronic second moment (xx, xy, xz, yx, yy, yz, zx, zy, zz).
- 'dm' - electronic dipole moment (x, y, z).

Parameters

- **oper** (str) – Key to specify the operator.
- **origin** (tuple, default: (0, 0, 0)) – Coordinate position of the origin.

Returns
Union[*FermionOperator*, List[*FermionOperator*]] – One electron operators.

extract_point_group_information(*reduce_infinite_point_groups=True*)

Returns point group information.

Parameters
reduce_infinite_point_groups – Reduce infinite point groups, e.g., Coov -> C2v

Returns
Tuple[str, List[str]] – Point group symmetry, list of orbital irreps.

Examples

```
>>> from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF
>>> driver = ChemistryDriverPySCFMolecularRHF(
...     geometry='H 0 0 0; H 0 0 0.75',
...     basis='sto3g',
...     point_group_symmetry=True,
... )
>>> ham, space, state = driver.get_system()
>>> driver.extract_point_group_information()
('D2h', ['Ag', 'Ag', 'B1u', 'B1u'])
```

---

classmethod from_mf(*mf*, *frozen=None*, *transf=None*)

Initialize driver from a PySCF mean-field object.

Parameters

- **mf** (SCF) – PySCF mean-field object.
- **frozen** (Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]], None], default: None) – Frozen core specified as either list or callable.
- **transf** (Union[Callable[[ndarray], ndarray], Transf, None], default: None) – Orbital transformation function.

Returns

ChemistryDriverPySCFMolecularROHFQMMMCOSMO – PySCF driver.

property frozen: List[int] | List[List[int]]

Return the frozen orbital information.

Return type

Union[List[int], List[List[int]]]

generate_report()

Generate report in a hierarchical dictionary format.

PySCF attributes such as mo_coeff are exported if the SCF is converged.

Returns

Dict[str, Any] – Attributes of the internal PySCF mean-field object.

get_cube_density(*density_matrix*, *cube_resolution=0.25*)

Output Gaussian Cube contents for density.

Parameters

- **density_matrix** (ndarray) – One-body reduced density matrix. 2D array for RHF/ROHF, 3D array for UHF.
- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.

Returns

str – Cube file formatted string.

get_cube_orbitals(*cube_resolution=0.25*, *mo_coeff=None*, *orbital_indices=None*)

Output Gaussian Cube file contents for orbitals.

Parameters

- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.
- **mo_coeff** (Optional[array], default: None) – Molecular orbital coefficients for the orbitals to be visualised.
- **orbital_indices** (Optional[List], default: None) – Indices of the molecular orbitals of interest. If None, all orbitals are returned.

Returns

List[str] – List of cube file formatted strings. For UHF, the spinorbitals are returned as [mo1_alpha, mo1_beta, mo2_alpha, mo2_beta...].

get_excitation_amplitudes (*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*)

Get the (guess) parameters as excitation amplitudes from a classical calculation.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.

Returns

SymbolDict – Excitation parameters.

get_excitation_operators (*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*, *antihermitian=True*)

Get the excitation operators and (guess) parameters.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.
- **antihermitian** (bool, default: True) – Returns anti-Hermitian operators T - T^ if specified.

Returns

FermionOperatorList – Excitation operator list.

get_lowdin_system (*method='lowdin'*)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

method (str, default: "lowdin") – Method passed to PySCF's orth.orth_ao().

Returns

Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*], *FermionSpace*, Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*]] – Fermion Hamiltonian, Fock space, Fock state.

static get_mm_coulomb (*mm_charges_pyscf*, *mm_geometry*, *unit='Ha'*)

Get the Coulomb interaction energy between MM particles.

Parameters

- **mm_charges_pyscf** (Any) – Charges of MM region, in same order as mm_geometry.

- **mm_geometry** (Any) – Geometry of MM region in format `['Atom1', [x, y, z]], 'Atom2', [x, y, z]], ...]`.
- **unit** (str, default: "Ha") – Energy units of Coulomb interaction energy. Default is Hartree.

Returns
Electrostatic Coulomb energy.

get_mulliken_pop()

Interface for Mulliken population analysis of PySCF.

Returns
Tuple[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]] – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction(*rdms*)

Compute the Strongly contracted NEVPT2 correction to the energy from the provided density matrices.

Parameters
rdms (Tuple) – A tuple of reduced density matrices as PDMs in PySCF's un-reordered format as (pdm1, …, rdm4).

Returns
The NEVPT2 correction to the energy.

get_rdm1_ccsd()

Reduced one-body density matrix in the AO basis from CCSD.

Returns
Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*] – One-body reduced density matrix.

get_rdm2_ccsd()

Reduced two-body density matrix in the AO basis with CCSD.

Returns
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Two-body reduced density matrix.

Note: This object will be replaced with an RDM2 class not to return a raw 4D tensor.

---

get_system(*symmetry=1*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Parameters
symmetry (Union[str, int], default: 1) – Code to specify target symmetry for storage of two-body integrals. Uses the same convention as PySCF. Currently, supports s1, s4 and s8 index permutation symmetries.

Warning: For unrestricted integral operators, the aabb and bbaa two-body integrals cannot be compacted with s8 symmetry. If s8 symmetry is requested, the aaaa and bbbb will be stored with s8 symmetry, while the aabb and bbaa integrals will be stored with s4 symmetry.

Returns
Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*, *ChemistryRestrictedIntegralOperatorCompact*, *ChemistryUnrestrictedIntegralOperatorCompact*], *FermionSpace*, *FermionState*] – Fermion Hamiltonian, Fock space, Fock state.

get_system_ao(*run_hf=True*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Output hamiltonian operator will be of type *PySCFChemistryRestrictedIntegralOperator* or *PySCFChemistryUnrestrictedIntegralOperator*, which store the underlying atomic orbitals.

Parameters
run_hf (bool, default: True) – If True, a Hartree-Fock calculation for the system will be executed, and the Fock space and HF state are returned with the hamiltonian operator. If False, no calculation is performed and the Fock space and HF state are returned as None. If the PySCF mean-field object has already been converged prior to calling `get_system_ao()`, behaviour will follow `run_hf=True`.

Returns
Tuple[Union[*PySCFChemistryRestrictedIntegralOperator*, *PySCFChemistryUnrestrictedIntegralOperator*], Optional[*FermionSpace*], Optional[*FermionState*]] – Fermion Hamiltonian, Fock space, Fock state.

property mf_energy: float

Return the total mean-field energy.

Return type
float

property mf_type: str

Return the mean-field type as a string (e.g. “RHF”).

Returns
str – Mean-field type name.

property n_electron: int

Number of electrons in the active space.

Returns
int – Number of electrons.

property n_orb: int

Number of spatial orbitals.

Returns
int – Number of spatial orbitals.

print_json_report(**args*, ***kwargs*)

Prints report in json format.

run_casci(***kwargs*)

Calculate the CASCI energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the `mcscf.CASCI` object.

Returns
float – CASCI energy.

run_ccsd(***kwargs*)

Calculate the CCSD energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the `cc.CCSD` object.

Returns

`float` – CCSD energy

run_hf()

Calculate the HF energy.

Returns

`float` – HF energy

run_mp2(***kwargs*)

Calculate the MP2 energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the `mp.MP2` object.

Returns

`float` – MP2 energy

class ChemistryDriverPySCFMolecularUHF(*geometry=None*, *zmatrix=None*, *basis=None*, *ecp=None*, *charge=0*, *multiplicity=1*, *frozen=None*, *verbose=0*, *output=None*, *point_group_symmetry=False*)

PySCF driver for molecular UHF calculations.

Parameters

- **geometry** (`Union[List, str, Geometry, None]`, default: `None`) – Molecular geometry.
- **zmatrix** (`Optional[str]`, default: `None`) – Z matrix representation of molecular geometry (Used only if `geometry` is not specified).
- **basis** (`Optional[Any]`, default: `None`) – Atomic basis set valid for `Mole` class.
- **ecp** (`Optional[Any]`, default: `None`) – Effective core potentials.
- **charge** (`int`, default: `0`) – Total charge.
- **multiplicity** (`int`, default: `1`) – Spin multiplicity, 2S+1.
- **frozen** (`Optional[Any]`, default: `None`) – Frozen orbital information.
- **verbose** (`int`, default: `0`) – Control PySCF verbosity.
- **output** (`Optional[str]`, default: `None`) – Specify log file name. If None, logs are printed to STDOUT.
- **point_group_symmetry** (`bool`, default: `False`) – Enable point group symmetry.

compute_nuclear_dipole()

Compute the nuclear electric dipole.

Returns

`Tuple[float, float, float]` – x, y, and z components of nuclear electric dipole.

compute_one_electron_operator(*oper*, *origin=(0, 0, 0)*)

Compute a one-electron fermionic operator in atomic units.

For operators with origin-dependent expectation values, origin can be specified. oper values:

- 'kin' - kinetic energy.
- 'nuc' - nucleus-electron attraction energy.
- 'hcore' - one-electron hamiltonian.
- 'ovlp' - one-electron overlap.
- 'r' - electronic first moment (x, y, z).
- 'rr' - electronic second moment (xx, xy, xz, yx, yy, yz, zx, zy, zz).
- 'dm' - electronic dipole moment (x, y, z).

Parameters

- **oper** (`str`) – Key to specify the operator.
- **origin** (`tuple`, default: `(0, 0, 0)`) – Coordinate position of the origin.

Returns

`Union`[*FermionOperator*, `List`[*FermionOperator*]] – One electron operators.

extract_point_group_information (*reduce_infinite_point_groups=True*)

Returns point group information.

Parameters

reduce_infinite_point_groups – Reduce infinite point groups, e.g., Coov -> C2v

Returns

`Tuple`[`str`, `List`[`str`]] – Point group symmetry, list of orbital irreps.

Examples

```
>>> from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF
>>> driver = ChemistryDriverPySCFMolecularRHF(
...     geometry='H 0 0 0; H 0 0 0.75',
...     basis='sto3g',
...     point_group_symmetry=True,
... )
>>> ham, space, state = driver.get_system()
>>> driver.extract_point_group_information()
('D2h', ['Ag', 'Ag', 'B1u', 'B1u'])
```

---

classmethod from_mf (*mf*, *frozen=None*)

Initialize driver from a PySCF mean-field object.

Parameters

- **mf** (`SCF`) – PySCF mean-field object.
- **frozen** (`Union`[`List`[`int`], `List`[`List`[`int`]], `Callable`[[`SCF`], `Union`[`List`[`int`], `List`[`List`[`int`]]]], `None`], default: `None`) – Frozen core specified as either list or callable.

Returns

ChemistryDriverPySCFMolecularUHF – PySCF driver.

property frozen: List[int] | List[List[int]]

Return the frozen orbital information.

Return type
Union[List[int], List[List[int]]]

generate_report()

Generate report in a hierarchical dictionary format.

PySCF attributes such as `mo_coeff` are exported if the SCF is converged.

Returns
Dict[str, Any] – Attributes of the internal PySCF mean-field object.

get_cube_density(*density_matrix*, *cube_resolution=0.25*)

Output Gaussian Cube contents for density.

Parameters

- **density_matrix** (ndarray) – One-body reduced density matrix. 2D array for RHF/ROHF, 3D array for UHF.
- **cube_resolution** (float, default: 0.25) – Resolution to be passed to `cubegen.orbital`.

Returns
str – Cube file formatted string.

get_cube_orbitals(*cube_resolution=0.25*, *mo_coeff=None*, *orbital_indices=None*)

Output Gaussian Cube file contents for orbitals.

Parameters

- **cube_resolution** (float, default: 0.25) – Resolution to be passed to `cubegen.orbital`.
- **mo_coeff** (Optional[array], default: None) – Molecular orbital coefficients for the orbitals to be visualised.
- **orbital_indices** (Optional[List], default: None) – Indices of the molecular orbitals of interest. If None, all orbitals are returned.

Returns
List[str] – List of cube file formatted strings. For UHF, the spinorbitals are returned as `[mo1_alpha, mo1_beta, mo2_alpha, mo2_beta...]`.

get_excitation_amplitudes(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*)

Get the (guess) parameters as excitation amplitudes from a classical calculation.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.

Returns
SymbolDict – Excitation parameters.

get_excitation_operators(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*, *antihermitian=True*)

Get the excitation operators and (guess) parameters.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.
- **antihermitian** (bool, default: True) – Returns anti-Hermitian operators T - T^ if specified.

Returns
FermionOperatorList – Excitation operator list.

get_lowdin_system(*method='lowdin'*)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters
method (str, default: "lowdin") – Method passed to PySCF's orth.orth_ao().

Returns
Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*], *FermionSpace*, Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*]] – Fermion Hamiltonian, Fock space, Fock state.

get_mulliken_pop()

Interface for Mulliken population analysis of PySCF.

Returns
Tuple[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]] – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction(*rdms*)

Compute the Strongly contracted NEVPT2 correction to the energy from the provided density matrices.

Parameters
rdms (Tuple) – A tuple of reduced density matrices as PDMs in PySCF's un-reordered format as (pdm1, …, rdm4).

Returns
The NEVPT2 correction to the energy.

get_rdm1_ccsd()

Reduced one-body density matrix in the AO basis from CCSD.

Returns
Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*] – One-body reduced density matrix.

get_rdm2_ccsd()

Reduced two-body density matrix in the AO basis with CCSD.

Returns
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Two-body reduced density matrix.

Note: This object will be replaced with an RDM2 class not to return a raw 4D tensor.

get_system(*symmetry=1*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Parameters
symmetry (Union[str, int], default: 1) – Code to specify target symmetry for storage of two-body integrals. Uses the same convention as PySCF. Currently, supports s1, s4 and s8 index permutation symmetries.

Warning: For unrestricted integral operators, the aabb and bbaa two-body integrals cannot be compacted with s8 symmetry. If s8 symmetry is requested, the aaaa and bbbb will be stored with s8 symmetry, while the aabb and bbaa integrals will be stored with s4 symmetry.

Returns
Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*, *ChemistryRestrictedIntegralOperatorCompact*, *ChemistryUnrestrictedIntegralOperatorCompact*], *FermionSpace*, *FermionState*] – Fermion Hamiltonian, Fock space, Fock state.

get_system_ao(*run_hf=True*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Output hamiltonian operator will be of type *PySCFChemistryRestrictedIntegralOperator* or *PySCFChemistryUnrestrictedIntegralOperator*, which store the underlying atomic orbitals.

Parameters
run_hf (bool, default: True) – If True, a Hartree-Fock calculation for the system will be executed, and the Fock space and HF state are returned with the hamiltonian operator. If False, no calculation is performed and the Fock space and HF state are returned as None. If the PySCF mean-field object has already been converged prior to calling get_system_ao(), behaviour will follow run_hf=True.

Returns
Tuple[Union[*PySCFChemistryRestrictedIntegralOperator*, *PySCFChemistryUnrestrictedIntegralOperator*], Optional[*FermionSpace*], Optional[*FermionState*]] – Fermion Hamiltonian, Fock space, Fock state.

property mf_energy: float

Return the total mean-field energy.

Return type
float

property mf_type: str

Return the mean-field type as a string (e.g. "RHF").

Returns

str – Mean-field type name.

property n_electron: int

Number of electrons in the active space.

Returns

int – Number of electrons.

property n_orb: int

Number of spatial orbitals.

Returns

int – Number of spatial orbitals.

print_json_report(**args*, ***kwargs*)

Prints report in json format.

run_casci(***kwargs*)

Calculate the CASCI energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the `mcscf.CASCI` object.

Returns

float – CASCI energy.

run_ccsd(***kwargs*)

Calculate the CCSD energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the `cc.CCSD` object.

Returns

float – CCSD energy

run_hf()

Calculate the HF energy.

Returns

float – HF energy

run_mp2(***kwargs*)

Calculate the MP2 energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the `mp.MP2` object.

Returns

float – MP2 energy

class ChemistryDriverPySCFMolecularUHFQMMMCOSMO(*geometry=None*, *zmatrix=None*, *basis=None*, *ecp=None*, *charge=0*, *multiplicity=1*, *frozen=None*, *do_qmmm=None*, *mm_charges=None*, *mm_geometry=None*, *do_mm_coulomb=None*, *do_cosmo=None*, *verbose=0*, *output=None*, *point_group_symmetry=False*)

PySCF driver for molecular UHF, with QMMM and COSMO calculations.

QMMM: Quantum Mechanics - Molecular mechanics. COSMO: Conductor-like Screening Model.

Parameters

- **geometry** (Union[List, str, None], default: None) – Molecular geometry.
- **zmatrix** (Optional[str], default: None) – Z matrix representation of molecular geometry (Used only if geometry is not specified).
- **basis** (Optional[Any], default: None) – Atomic basis set valid for Mole class.
- **ecp** (Optional[Any], default: None) – Effective core potentials.
- **charge** (int, default: 0) – Total charge.
- **multiplicity** (int, default: 1) – Spin multiplicity, 2S+1.
- **frozen** (Optional[Any], default: None) – Frozen orbital information.
- **do_qmmm** (Optional[bool], default: None) – If True, do QMMM embedding. MM geometry and charges are required.
- **mm_charges** (Optional[Any], default: None) – Point charge values for MM region.
- **mm_geometry** (Optional[Any], default: None) – Geometry of MM region.
- **do_mm_coulomb** (Optional[bool], default: None) – If True, calculate MM-MM Coulomb interaction energy. Needs to be added as a constant to the final energy: <\| H \|> + self.e_mm_coulomb
- **do_cosmo** (Optional[bool], default: None) – If True, adds implicit water (COSMO) to the mean-field calculation. get_system_legacy() returns H + v_solvent as fermion operator. The resulting final energy needs to be corrected as <\|H + v_solvet\|> + self.cosmo_correction. Where self.cosmo_correction = E_cosmo - Tr[v_solvent * rdm1]. If do_cosmo=True, self.cosmo_correction is defined after self._run_hf().
- **verbose** (int, default: 0) – Control SCF verbosity.
- **output** (Optional[str], default: None) – Specify log file name. If None, logs are printed to STDOUT.
- **point_group_symmetry** (bool, default: False) – Enable point group symmetry.

static build_mm_charges(*mm_charges*, *mm_geometry*)

Puts InQuanto MM charge dict into PySCF-friendly format.

Orders mm charges according to mm_geometry.

Parameters

- **mm_geometry** (List) – Geometry of MM region in format ['Atom1', [x, y, z]], 'Atom2', [x, y, z]], ...].
- **mm_charges** (Union[Dict, List, float]) – Dict of charges, or float if all charges are the same. If float, all output charges will all have this value. If it is already a list, do nothing.

Returns

List – List of mm charges ordered the same as mm_geometry.

compute_nuclear_dipole()

Compute the nuclear electric dipole.

Returns
Tuple[float, float, float] – x, y, and z components of nuclear electric dipole.

compute_one_electron_operator(*oper*, *origin=(0, 0, 0)*)

Compute a one-electron fermionic operator in atomic units.

For operators with origin-dependent expectation values, origin can be specified. oper values:

- 'kin' - kinetic energy.
- 'nuc' - nucleus-electron attraction energy.
- 'hcore' - one-electron hamiltonian.
- 'ovlp' - one-electron overlap.
- 'r' - electronic first moment (x, y, z).
- 'rr' - electronic second moment (xx, xy, xz, yx, yy, yz, zx, zy, zz).
- 'dm' - electronic dipole moment (x, y, z).

Parameters

- **oper** (str) – Key to specify the operator.
- **origin** (tuple, default: (0, 0, 0)) – Coordinate position of the origin.

Returns
Union[*FermionOperator*, List[*FermionOperator*]] – One electron operators.

extract_point_group_information(*reduce_infinite_point_groups=True*)

Returns point group information.

Parameters
reduce_infinite_point_groups – Reduce infinite point groups, e.g., Coov -> C2v

Returns
Tuple[str, List[str]] – Point group symmetry, list of orbital irreps.

Examples

```
>>> from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF
>>> driver = ChemistryDriverPySCFMolecularRHF(
...     geometry='H 0 0 0; H 0 0 0.75',
...     basis='sto3g',
...     point_group_symmetry=True,
... )
>>> ham, space, state = driver.get_system()
>>> driver.extract_point_group_information()
('D2h', ['Ag', 'Ag', 'B1u', 'B1u'])
```

---

classmethod from_mf(*mf*, *frozen=None*)

Initialize driver from a PySCF mean-field object.

Parameters

- **mf** (SCF) – PySCF mean-field object.

- **frozen** (Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]], None], default: None) – Frozen core specified as either list or callable.

Returns
ChemistryDriverPySCFMolecularUHFQMMMCOSMO – PySCF driver.

property frozen: List[int] | List[List[int]]

Return the frozen orbital information.

Return type
Union[List[int], List[List[int]]]

generate_report()

Generate report in a hierarchical dictionary format.

PySCF attributes such as mo_coeff are exported if the SCF is converged.

Returns
Dict[str, Any] – Attributes of the internal PySCF mean-field object.

get_cube_density(*density_matrix*, *cube_resolution=0.25*)

Output Gaussian Cube contents for density.

Parameters

- **density_matrix** (ndarray) – One-body reduced density matrix. 2D array for RHF/ROHF, 3D array for UHF.
- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.

Returns
str – Cube file formatted string.

get_cube_orbitals(*cube_resolution=0.25*, *mo_coeff=None*, *orbital_indices=None*)

Output Gaussian Cube file contents for orbitals.

Parameters

- **cube_resolution** (float, default: 0.25) – Resolution to be passed to cubegen.orbital.
- **mo_coeff** (Optional[array], default: None) – Molecular orbital coefficients for the orbitals to be visualised.
- **orbital_indices** (Optional[List], default: None) – Indices of the molecular orbitals of interest. If None, all orbitals are returned.

Returns
List[str] – List of cube file formatted strings. For UHF, the spinorbitals are returned as [mo1_alpha, mo1_beta, mo2_alpha, mo2_beta...].

get_excitation_amplitudes(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*)

Get the (guess) parameters as excitation amplitudes from a classical calculation.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.

- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.

Returns
SymbolDict – Excitation parameters.

get_excitation_operators(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*, *antihermitian=True*)

Get the excitation operators and (guess) parameters.

Parameters

- **fock_space** (*FermionSpace*) – Fermionic Fock space information.
- **threshold** (float, default: 0.0) – Threshold of the amplitude to include the excitation operator.
- **t1** (Optional[ndarray], default: None) – Guess 1e excitation amplitudes.
- **t2** (Optional[ndarray], default: None) – Guess 2e excitation amplitudes.
- **compact** (bool, default: False) – Enable compact form of excitation for the restrected wavefunctions.
- **antihermitian** (bool, default: True) – Returns anti-Hermitian operators T - T^ if specified.

Returns
FermionOperatorList – Excitation operator list.

get_lowdin_system(*method='lowdin'*)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters
method (str, default: "lowdin") – Method passed to PySCF's orth.orth_ao().

Returns
Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*], *FermionSpace*, Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*]] – Fermion Hamiltonian, Fock space, Fock state.

static get_mm_coulomb(*mm_charges_pyscf*, *mm_geometry*, *unit='Ha'*)

Get the Coulomb interaction energy between MM particles.

Parameters

- **mm_charges_pyscf** (Any) – Charges of MM region, in same order as mm_geometry.
- **mm_geometry** (Any) – Geometry of MM region in format ['Atom1', [x, y, z]], 'Atom2', [x, y, z]], ...].
- **unit** (str, default: "Ha") – Energy units of Coulomb interaction energy. Default is Hartree.

Returns
Electrostatic Coulomb energy.

get_mulliken_pop()

Interface for Mulliken population analysis of PySCF.

Returns
Tuple[ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]] – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction(*rdms*)

Compute the Strongly contracted NEVPT2 correction to the energy from the provided density matrices.

Parameters
rdms (Tuple) – A tuple of reduced density matrices as PDMs in PySCF's un-reordered format as (pdm1, …, rdm4).

Returns
The NEVPT2 correction to the energy.

get_rdm1_ccsd()

Reduced one-body density matrix in the AO basis from CCSD.

Returns
Union[*RestrictedOneBodyRDM*, *UnrestrictedOneBodyRDM*] – One-body reduced density matrix.

get_rdm2_ccsd()

Reduced two-body density matrix in the AO basis with CCSD.

Returns
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Two-body reduced density matrix.

Note: This object will be replaced with an RDM2 class not to return a raw 4D tensor.

---

get_system(*symmetry=1*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Parameters
symmetry (Union[str, int], default: 1) – Code to specify target symmetry for storage of two-body integrals. Uses the same convention as PySCF. Currently, supports s1, s4 and s8 index permutation symmetries.

Warning: For unrestricted integral operators, the aabb and bbaa two-body integrals cannot be compacted with s8 symmetry. If s8 symmetry is requested, the aaaa and bbbb will be stored with s8 symmetry, while the aabb and bbaa integrals will be stored with s4 symmetry.

Returns
Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*, *ChemistryRestrictedIntegralOperatorCompact*, *ChemistryUnrestrictedIntegralOperatorCompact*], *FermionSpace*, *FermionState*] – Fermion Hamiltonian, Fock space, Fock state.

get_system_ao(*run_hf=True*)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Output hamiltonian operator will be of type *PySCFChemistryRestrictedIntegralOperator* or *PySCFChemistryUnrestrictedIntegralOperator*, which store the underlying atomic orbitals.

Parameters

run_hf (bool, default: True) – If True, a Hartree-Fock calculation for the system will be executed, and the Fock space and HF state are returned with the hamiltonian operator. If False, no calculation is performed and the Fock space and HF state are returned as None. If the PySCF mean-field object has already been converged prior to calling get_system_ao(), behaviour will follow run_hf=True.

Returns

Tuple[Union[*PySCFChemistryRestrictedIntegralOperator*, *PySCFChemistryUnrestrictedIntegralOperator*], Optional[*FermionSpace*], Optional[*FermionState*]] – Fermion Hamiltonian, Fock space, Fock state.

property mf_energy: float

Return the total mean-field energy.

Return type

float

property mf_type: str

Return the mean-field type as a string (e.g. "RHF").

Returns

str – Mean-field type name.

property n_electron: int

Number of electrons in the active space.

Returns

int – Number of electrons.

property n_orb: int

Number of spatial orbitals.

Returns

int – Number of spatial orbitals.

print_json_report(**args*, ***kwargs*)

Prints report in json format.

run_casci(***kwargs*)

Calculate the CASCI energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the mcscf.CASCI object.

Returns

float – CASCI energy.

run_ccsd(***kwargs*)

Calculate the CCSD energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the cc.CCSD object.

Returns

float – CCSD energy

run_hf()

Calculate the HF energy.

Returns
`float` – HF energy

run_mp2 (***kwargs*)

Calculate the MP2 energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the `mp.MP2` object.

Returns
`float` – MP2 energy

class ChemistryDriverPySCFMomentumRHF (*geometry=None*, *zmatrix=None*, *cell=None*, *nks=None*, *basis=None*, *ecp=None*, *pseudo=None*, *charge=0*, *exp_to_discard=None*, *df='GDF'*, *dimension=3*, *verbose=0*, *output=None*, *frozen=None*, *precision=None*)

PySCF driver for momentum-space RHF calculations.

Parameters

- **geometry** (`Union[List, str, GeometryPeriodic, None]`, default: `None`) – Molecular geometry.
- **zmatrix** (`Optional[str]`, default: `None`) – Z matrix representation of molecular geometry (Used only if `geometry` is not specified).
- **cell** (`Optional[List[List]]`, default: `None`) – Unit cell parameter. If provided, overrides `geometry.unit_cell`.
- **nks** (`Optional[List[int]]`, default: `None`) – Number of k-points for each direction.
- **basis** (`Optional[Any]`, default: `None`) – Atomic basis set valid for `Mole` class.
- **ecp** (`Optional[Any]`, default: `None`) – Effective core potentials.
- **pseudo** (`Optional[Any]`, default: `None`) – Pseudo potentials.
- **charge** (`int`, default: `0`) – Total charge.
- **exp_to_discard** (`Optional[float]`, default: `None`) – Exponent to discard a primitive Gaussian.
- **df** (`str`, default: `"GDF"`) – Density fitting function name.
- **dimension** (`int`, default: `3`) – Number of spatial dimensions.
- **frozen** (`Optional[Any]`, default: `None`) – Frozen orbital information.
- **verbose** (`int`, default: `0`) – Control SCF verbosity.
- **output** (`Optional[str]`, default: `None`) – Specify log file name. If None, logs are printed to STDOUT.
- **precision** (`Optional[float]`, default: `None`) – Precision passed to PySCF Cell.

classmethod from_mf (*mf*, *frozen=None*)

Initialize driver from a PySCF mean-field object.

Parameters

- **mf** (`SCF`) – PySCF mean-field object.
- **frozen** (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]], None]`, default: `None`) – Frozen core specified as either list or callable.

Returns
ChemistryDriverPySCFMomentumRHF – PySCF driver.

property frozen: List[int] | List[List[int]]
Return the frozen orbital information.

Return type
Union[List[int], List[List[int]]]

generate_report()
Generate report in a hierarchical dictionary format.

PySCF attributes such as `mo_coeff` are exported if the SCF is converged.

Returns
Dict[str, Any] – Attributes of the internal PySCF mean-field object.

get_excitation_amplitudes(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*)
Get the (guess) parameters as excitation amplitudes from a classical calculation.

Return type
SymbolDict

get_excitation_operators(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*, *antihermitian=True*)
Get the excitation operators.

Return type
FermionOperatorList

get_madelung_constant()
Return Madelung constant for momentum-space calculations.

Returns
float – Madelung constant contribution to the energy.

get_system()
Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Returns
Tuple[Union[*ChemistryRestrictedIntegralOperator*, *ChemistryUnrestrictedIntegralOperator*], *FermionSpace*, *FermionState*] – Fermion Hamiltonian, Fock space, Fock state.

property mf_energy: float
Return the total mean-field energy.

Return type
float

property mf_type: str
Return the mean-field type as a string (e.g. “RHF”).

Returns
str – Mean-field type name.

property n_electron: int
Number of electrons in the active space.

Returns
int – Number of electrons in the active space.

property n_kp: int

Number of k points.

Returns

int – Number of k points.

property n_orb: int

Number of spatial orbitals in a unit cell.

Returns

int – Number of spatial orbitals in a unit cell

print_json_report(**args*, ***kwargs*)

Prints report in json format.

run_casci(***kwargs*)

Calculate the CASCI energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the mcscf.CASCI object.

Returns

float – CASCI energy.

run_ccsd(***kwargs*)

Calculate the CCSD energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the cc.CCSD object.

Returns

float – CCSD energy

run_hf()

Calculate the HF energy.

Returns

float – HF energy

run_mp2(***kwargs*)

Calculate the MP2 energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the mp.MP2 object.

Returns

float – MP2 energy

class ChemistryDriverPySCFMomentumROHF(*geometry=None*, *zmatrix=None*, *cell=None*, *nks=None*, *basis=None*, *ecp=None*, *pseudo=None*, *multiplicity=1*, *charge=0*, *exp_to_discard=None*, *df='GDF'*, *dimension=3*, *verbose=0*, *output=None*, *frozen=None*, *precision=None*)

PySCF driver for momentum-space ROHF calculations.

Parameters

- **geometry** (Union[List, str, *GeometryPeriodic*, None], default: None) – Molecular geometry.
- **zmatrix** (Optional[str], default: None) – Z matrix representation of molecular geometry (Used only if geometry is not specified).

- **cell** (Optional[List[List]], default: None) – Unit cell parameter. If provided, overrides geometry.unit_cell.
- **nks** (Optional[List[int]], default: None) – Number of k-points for each direction.
- **basis** (Optional[Any], default: None) – Atomic basis set valid for Mole class.
- **ecp** (Optional[Any], default: None) – Effective core potentials.
- **pseudo** (Optional[Any], default: None) – Pseudo potentials.
- **multiplicity** (int, default: 1) – Spin multiplicity, 2S+1.
- **charge** (int, default: 0) – Total charge.
- **exp_to_discard** (Optional[float], default: None) – Exponent to discard a primitive Gaussian.
- **df** (str, default: "GDF") – Density fitting function name.
- **dimension** (int, default: 3) – Number of spatial dimensions.
- **frozen** (Optional[Any], default: None) – Frozen orbital information.
- **verbose** (int, default: 0) – Control SCF verbosity.
- **output** (Optional[str], default: None) – Specify log file name. If None, logs are printed to STDOUT.
- **precision** (Optional[float], default: None) – Precision passed to PySCF Cell.

classmethod from_mf(*mf*, *frozen=None*)

Initialize driver from a PySCF mean-field object.

Parameters

- **mf** (SCF) – PySCF mean-field object.
- **frozen** (Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]], None], default: None) – Frozen core specified as either list or callable.

Returns

ChemistryDriverPySCFMomentumROHF – PySCF driver.

property frozen: List[int] | List[List[int]]

Return the frozen orbital information.

Return type

Union[List[int], List[List[int]]]

generate_report()

Generate report in a hierarchical dictionary format.

PySCF attributes such as mo_coeff are exported if the SCF is converged.

Returns

Dict[str, Any] – Attributes of the internal PySCF mean-field object.

get_excitation_amplitudes(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*)

Get the (guess) parameters as excitation amplitudes from a classical calculation.

Return type

SymbolDict

get_excitation_operators(*fock_space*, *threshold=0.0*, *t1=None*, *t2=None*, *compact=False*, *antihermitian=True*)

Get the excitation operators.

Return type
FermionOperatorList

get_madelung_constant()

Return Madelung constant for momentum-space calculations.

Returns
float – Madelung constant contribution to the energy.

get_system()

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Returns
Tuple[*ChemistryRestrictedIntegralOperator*, *FermionSpace*, *FermionState*] – Fermion Hamiltonian, Fock space, Fock state.

property mf_energy: float

Return the total mean-field energy.

Return type
float

property mf_type: str

Return the mean-field type as a string (e.g. "RHF").

Returns
str – Mean-field type name.

property n_electron: int

Number of electrons in the active space.

Returns
int – Number of electrons in the active space.

property n_kp: int

Number of k points.

Returns
int – Number of k points.

property n_orb: int

Number of spatial orbitals in a unit cell.

Returns
int – Number of spatial orbitals in a unit cell

print_json_report(**args*, ***kwargs*)

Prints report in json format.

run_casci(***kwargs*)

Calculate the CASCI energy.

Parameters
****kwargs** – Keyword arguments to set attributes of the mcscf.CASCI object.

Returns
float – CASCI energy.

run_ccsd(***kwargs*)

Calculate the CCSD energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the `cc.CCSD` object.

Returns

`float` – CCSD energy

run_hf()

Calculate the HF energy.

Returns

`float` – HF energy

run_mp2(***kwargs*)

Calculate the MP2 energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the `mp.MP2` object.

Returns

`float` – MP2 energy

class DMETRHFFragmentPySCFActive(*dmet*, *mask*, *name=None*, *frozen=None*)

Custome active space fragment solver for DMETRHF method based on PySCF RHF method.

This class implements the solve(…) method, which uses PySCF to perform a RHF calculation to select an active space, then calls the solve_active(…) method to calculate the ground state quantities for the active space.

Note: The class does not implement the active_space(…) method, it is the user's responsibility to create a subclass and provide an implementation.

Parameters

- **dmet** (*DMETRHF*) – DMETRHF instance that uses this fragment.
- **mask** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – Array of booleans, indices correspond to the spatial orbitals of the total system. If the corresponding orbital is in the fragment, then the boolean should be True - else it should be False.
- **name** (Optional[str], default: None) – Optional name of the fragment.
- **frozen** (Union[List[int], Callable[[SCF], List[int]], None], default: None) – Frozen orbital information.

static construct_fragment_energy_operator(*one_body*, *two_body*, *veff*, *mask*)

Computes the fragment energy with symmetrization.

Parameters

- **mask** (ndarray) – Fragment orbitals boolean mask.
- **one_body** (ndarray) – One-body integrals in the fragment basis.
- **two_body** (ndarray) – Two-body integrals in the fragment basis.
- **veff** (ndarray) – Fock matrix in the fragment basis.

Returns
ChemistryRestrictedIntegralOperator – Fragment energy operator based on democrating mixing defined by DMET.

solve (*hamiltonian_operator*, *fragment_energy_operator*, *n_electron*)

Solves the fragment system using PySCF's RHF and solve_active(…).

First it performs an RHF calculation and the active space problem will be passed on to solve_active(…) method.

It returns the ground state energy, the fragment energy and the one-body RDM for the whole fragment+bath system.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator of the fragment system.
- **fragment_energy_operator** (*ChemistryRestrictedIntegralOperator*) – Fragment energy operator of the fragment system.
- **n_electron** (int) – Number of electrons in the fragment system.

Returns
Tuple[float, float, ndarray[Any, dtype[TypeVar(ScalarType, bound=generic, covariant=True)]]] – Energy, fragment energy, 1-RDM of the embedding system (fragment and bath).

solve_active (*hamiltonian_operator*, *fragment_energy_operator*, *fermion_space*, *fermion_state*)

This is an abstract method.

The subclass implementation must return the energy, fragment energy, and the one-particle reduced density matrix (1-RDM) for the active space problem.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator for the active space of the fragment system.
- **fragment_energy_operator** (*ChemistryRestrictedIntegralOperator*) – Fragment energy operator for the active space of the fragment system.
- **fermion_space** (*FermionSpace*) – Fermion space object for the active space of the fragment system.
- **fermion_state** (*FermionState*) – Fock state for the active space of the fragment system.

Returns
Tuple[float, float, *RestrictedOneBodyRDM*] – Energy, fragment energy, rdm1 of the active space of the embedding system (fragment and bath).

class DMETRHFFragmentPySCFCCSD (*dmet*, *mask*, *name=None*, *frozen=None*)

PySCF CCSD fragment solver for the DMETRHF method.

This class implements the solve(…) method, which uses PySCF to calculate CCSD ground state quantities of a fragment.

Parameters

- **dmet** (*DMETRHF*) – DMETRHF instance that uses this fragment.

- **mask** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – Array of booleans, indices correspond to the spatial orbitals of the total system. If the corresponding orbital is in the fragment, then the boolean should be True - else it should be False.
- **name** (Optional[str], default: None) – Optional name of the fragment.
- **frozen** (Union[List[int], Callable[[SCF], List[int]], None], default: None) – Frozen orbital information.

static compute_fragment_energy(*rdm1*, *rdm2*, *one_body*, *two_body*, *veff*, *mask*)

Computes the fragment energy with symmetrization.

Parameters

- **mask** (ndarray) – Fragment orbitals boolean mask.
- **rdm1** (ndarray) – One-body density matrix in the fragment basis.
- **rdm2** (ndarray) – Two-body density matrix in the fragment basis.
- **one_body** (ndarray) – One-body integrals in the fragment basis.
- **two_body** (ndarray) – Two-body integrals in the fragment basis.
- **veff** (ndarray) – Fock matrix in the fragment basis.

Returns

float – Fragment energy.

solve(*hamiltonian_operator*, *n_electron*)

Solves the fragment system using PySCF's CCSD method.

It returns the CCSD ground state energy, and one- and two-body RDM-s.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator of the fragment system.
- **n_electron** (int) – Number of electrons in the fragment system.

Returns

Tuple[float, ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]] – Energy, 1-RDM, 2-RDM of the embedding system (fragment and bath).

class DMETRHFFragmentPySCFFCI(*dmet*, *mask*, *name=None*)

PySCF FCI fragment solver for the DMETRHF method.

This class implements the solve(…) method, which uses PySCF to calculate FCI ground state quantities of a fragment.

Parameters

- **dmet** (DMETRHF) – DMETRHF instance that uses this fragment.
- **mask** (ndarray) – Array of booleans, indices correspond to the spatial orbitals of the total system. If the corresponding orbital is in the fragment, then the boolean should be True - else it should be False.
- **name** (str, default: None) – Optional name of the fragment.
- **frozen** – Frozen orbital information.

static compute_fragment_energy(*rdm1*, *rdm2*, *one_body*, *two_body*, *veff*, *mask*)

Computes the fragment energy with symmetrization.

Parameters

- **mask** (ndarray) – Fragment orbitals boolean mask.
- **rdm1** (ndarray) – One-body density matrix in the fragment basis.
- **rdm2** (ndarray) – Two-body density matrix in the fragment basis.
- **one_body** (ndarray) – One-body integrals in the fragment basis.
- **two_body** (ndarray) – Two-body integrals in the fragment basis.
- **veff** (ndarray) – Fock matrix in the fragment basis.

Returns

float – Fragment energy.

solve(*hamiltonian_operator*, *n_electron*)

Solves the fragment system using PySCF's FCI method.

It returns the FCI ground state energy, and one- and two-body RDM-s.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator of the fragment system.
- **n_electron** (int) – Number of electrons in the fragment system.

Returns

Tuple[float, ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]] – Energy, 1-RDM, 2-RDM of the embedding system (fragment and bath).

class DMETRHFFragmentPySCFMP2(*dmet*, *mask*, *name=None*, *frozen=None*)

PySCF MP2 fragment solver for the DMETRHF method.

This class implements the solve(…) method, which uses PySCF to calculate MP2 ground state quantities of a fragment.

Parameters

- **dmet** (*DMETRHF*) – DMETRHF instance that uses this fragment.
- **mask** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – Array of booleans, indices correspond to the spatial orbitals of the total system. If the corresponding orbital is in the fragment, then the boolean should be True - else it should be False.
- **name** (Optional[str], default: None) – Optional name of the fragment.
- **frozen** (Union[List[int], Callable[[SCF], List[int]], None], default: None) – Frozen orbital information.

static compute_fragment_energy(*rdm1*, *rdm2*, *one_body*, *two_body*, *veff*, *mask*)

Computes the fragment energy with symmetrization.

Parameters

- **mask** (ndarray) – Fragment orbitals boolean mask.
- **rdm1** (ndarray) – One-body density matrix in the fragment basis.

- **rdm2** (ndarray) – Two-body density matrix in the fragment basis.
- **one_body** (ndarray) – One-body integrals in the fragment basis.
- **two_body** (ndarray) – Two-body integrals in the fragment basis.
- **veff** (ndarray) – Fock matrix in the fragment basis.

Returns
float – Fragment energy.

solve (*hamiltonian_operator*, *n_electron*)

Solves the fragment system using PySCF's MP2 method.

It returns the MP2 ground state energy, and one- and two-body RDM-s.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator of the fragment system.
- **n_electron** – Number of electrons in the fragment system.

Returns
Tuple[float, ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]] – Energy, 1-RDM, 2-RDM of the embedding system (fragment and bath).

class DMETRHFFragmentPySCFRHF (*dmet*, *mask*, *name=None*)

PySCF RHF fragment solver for DMETRHF method.

This class implements the solve(…) method, which uses PySCF to calculate RHF ground state quantities of a fragment.

Parameters

- **dmet** (DMETRHF) – DMETRHF instance that uses this fragment.
- **mask** (ndarray) – Array of booleans, indices correspond to the spatial orbitals of the total system. If the corresponding orbital is in the fragment, then the boolean should be True - else it should be False.
- **name** (str, default: None) – Optional name of the fragment.

static compute_fragment_energy (*rdm1*, *rdm2*, *one_body*, *two_body*, *veff*, *mask*)

Computes the fragment energy with symmetrization.

Parameters

- **mask** (ndarray) – Fragment orbitals boolean mask.
- **rdm1** (ndarray) – One-body density matrix in the fragment basis.
- **rdm2** (ndarray) – Two-body density matrix in the fragment basis.
- **one_body** (ndarray) – One-body integrals in the fragment basis.
- **two_body** (ndarray) – Two-body integrals in the fragment basis.
- **veff** (ndarray) – Fock matrix in the fragment basis.

Returns
float – Fragment energy.

solve (*hamiltonian_operator*, *n_electron*)

Solves the fragment system using PySCF's RHF method.

It returns the RHF ground state energy, and one- and two-body RDM-s.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator of the fragment system.
- **n_electron** (int) – Number of electrons in the fragment system.

Returns

Tuple[float, ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]] – Energy, 1-RDM, 2-RDM of the embedding system (fragment and bath).

class FromActiveSpace (*ncas*, *nelecas*)

Complete Active Space for molecular RHF/ROHF.

This class helps to specify the list of frozen orbitals from information about the active space. See below for a HOMO-LUMO CI calculation for minimal basis LiH, where the use of this class with `FromActiveSpace(nelecas=2, ncas=2)` is equivalent to the explicit list `[0, 3, 4, 5, 6]`.

Parameters

- **ncas** (int) – Number of active spatial orbitals
- **nelecas** (int) – Number of active electrons

Examples

```
>>> from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF,␣
↪FromActiveSpace
>>> driver = ChemistryDriverPySCFMolecularRHF(
...     geometry='Li 0 0 0; H 0 0 1.75',
...     basis='sto3g',
...     frozen=FromActiveSpace(ncas=5, nelecas=2),
... )
```

class FrozenCore (*n_core*)

Frozen core orbitals for molecular RHF/ROHF/UHF.

An alternative way of specifying the frozen spatial orbitals.

Parameters

n_core (int) – Number of frozen core spatial orbitals.

Examples

```
>>> from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF,␣
↪FrozenCore
>>> driver = ChemistryDriverPySCFMolecularRHF(
...     geometry='Li 0 0 0; H 0 0 1.75', basis='sto3g',
...     frozen=FrozenCore(n_core=1),
... )
```

class ImpurityDMETROHFFragmentPySCFActive(*dmet*, *mask*, *name=None*, *multiplicity=1*, *frozen=None*)

Custome active space fragment solver for ImpurityDMETROHF method based on PySCF ROHF method.

This class implements the solve(…) method, which uses PySCF to perform a ROHF calculation to select an active space, then calls the solve_active(…) method to calculate the ground state quantities for the active space.

Note: The class does not implement the active_space(…) method, it is the user's responsibility to create a subclass and provide an implementation.

Parameters

- **dmet** (*ImpurityDMETROHF*) – DMETRHF instance that uses this fragment.
- **mask** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – Array of booleans, indices correspond to the spatial orbitals of the total system. If the corresponding orbital is in the fragment, then the boolean should be True - else it should be False.
- **name** (Optional[str], default: None) – Optional name of the fragment.
- **multiplicity** (int, default: 1) – Multiplicity for the fragment ROHF.
- **frozen** (Union[List[int], Callable[[SCF], List[int]], None], default: None) – Frozen orbital information.

solve(*hamiltonian_operator*, *n_electron*)

Solves the fragment system using PySCF's ROHF and solve_active(…).

This method first runs an PySCF ROHF for the fragment system, then for the active space it calls the solve_active(…) method.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator of the fragment system.
- **n_electron** (int) – Number of electrons in the fragment system.

Returns

float – Energy of the embedding system (fragment and bath).

solve_active(*hamiltonian_operator*, *fermion_space*, *fermion_state*)

This is an abstract method.

The subclass implementation must return an accurate ground state energy of the active space of the embedding system (fragment+bath).

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator for the active space of the fragment system.
- **fermion_space** (*FermionSpace*) – Fermion space object for the active space of the fragment system.
- **fermion_state** (*FermionState*) – Fock state for the active space of the fragment system.

Returns

float – Energy of the embedding system (fragment and bath).

class ImpurityDMETROHFFragmentPySCFCCSD(*dmet*, *mask*, *name=None*, *multiplicity=1*, *frozen=None*)

PySCF CCSD fragment solver for Impurity DMETROHF.

This class implements the solve(…) method, which uses PySCF to calculate the CCSD ground state energy of a fragment.

Parameters

- **dmet** (*ImpurityDMETROHF*) – ImpurityDMET ROHF that uses this fragment.
- **mask** (ndarray) – Array of booleans, indices correspond to the spatial orbitals of the total system, the fragment is masked with True.
- **name** (Optional[str], default: None) – Name of the fragment.
- **multiplicity** (int, default: 1) – Multiplicity for the fragment ROHF.
- **frozen** (Union[List[int], Callable[[SCF], List[int]]], None], default: None) – Frozen orbital information.

solve(*hamiltonian_operator*, *n_electron*)

Solves the fragment system using PySCF's CCSD method.

It returns the CCSD ground state energy.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator of the fragment system.
- **n_electron** (int) – Number of electrons in the fragment system.

Returns

float – Energy of the embedding system (fragment and bath).

class ImpurityDMETROHFFragmentPySCFFCI(*dmet*, *mask*, *name=None*, *multiplicity=1*)

PySCF FCI fragment solver for Impurity DMETROHF.

This class implements the solve(…) method, which uses PySCF to calculate the FCI ground state energy of a fragment.

Parameters

- **dmet** (ImpurityDMETROHF) – ImpurityDMET ROHF that uses this fragment.
- **mask** (ndarray) – Array of booleans, indices correspond to the spatial orbitals of the total system, the fragment is masked with True.
- **name** (str, default: None) – Name of the fragment.
- **multiplicity** (int, default: 1) – Multiplicity for the fragment ROHF.

solve(*hamiltonian_operator*, *n_electron*)

Solves the fragment system using PySCF's FCI method.

It returns the FCI ground state energy.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator of the fragment system.
- **n_electron** (int) – Number of electrons in the fragment system.

Returns

float – Energy of the embedding system (fragment and bath).

class ImpurityDMETROHFFragmentPySCFMP2(*dmet*, *mask*, *name=None*, *multiplicity=1*, *frozen=None*)

PySCF MP2 fragment solver for Impurity DMETROHF.

This class implements the solve(…) method, which uses PySCF to calculate the MP2 ground state energy of a fragment.

Parameters

- **dmet** (*ImpurityDMETROHF*) – ImpurityDMET ROHF that uses this fragment.
- **mask** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – Array of booleans, indices correspond to the spatial orbitals of the total system, the fragment is masked with True.
- **name** (Optional[str], default: None) – Name of the fragment.
- **multiplicity** (int, default: 1) – Multiplicity for the fragment ROHF.
- **frozen** (Union[List[int], Callable[[SCF], List[int]], None], default: None) – Frozen orbital information.

solve(*hamiltonian_operator*, *n_electron*)

Solves the fragment system using PySCF's MP2 method.

It returns the MP2 ground state energy.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator of the fragment system.
- **n_electron** (int) – Number of electrons in the fragment system.

Returns

float – Energy of the embedding system (fragment and bath).

class ImpurityDMETROHFFragmentPySCFROHF(*dmet*, *mask*, *name=None*, *multiplicity=1*)

PySCF ROHF fragment solver for Impurity DMETROHF.

This class implements the solve(…) method, which uses PySCF to calculate the ROHF ground state energy of a fragment.

Parameters

- **dmet** (ImpurityDMETROHF) – ImpurityDMETROHF instance that uses this fragment.
- **mask** (ndarray) – Array of booleans, indices correspond to the spatial orbitals of the total system, the fragment is masked with True.
- **name** (str, default: None) – Name of the fragment.
- **multiplicity** (int, default: 1) – Multiplicity for the fragment ROHF.

solve(*hamiltonian_operator*, *n_electron*)

Solves the fragment system using PySCF's ROHF method.

It returns the ROHF ground state energy.

Parameters

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator of the fragment system.
- **n_electron** (int) – Number of electrons in the fragment system.

Returns
float – Energy of the embedding system (fragment and bath).

class PySCFChemistryRestrictedIntegralOperator(*mol_or_mf*, *run_hf=True*)

Handles a (restricted-orbital) chemistry integral operator.

Stores a PySCF `pyscf.scf.hf.RHF` object, and uses it to generate the constant, one- and two-body spatial integrals. All indices are in chemistry notation, `two_body[p, q, r, s]` = (pq|rs).

Raises

- **ValueError** – If the input `mol_or_mf` is not of type `pyscf.gto.Mole` or `pyscf.scf.hf.RHF`.
- **RuntimeError** – If the PySCF self-consistent Hartree-Fock calculation does not converge.

Parameters

- **mol_or_mf** (Union[Mole, RHF]) – Input PySCF object from which the molecular orbital integrals are computed.
- **run_hf** (bool, default: True) – Whether to run a restricted Hartree-Fock calculation on the input system.

TOLERANCE = 1e-08

Internal tolerance used when iterating over terms. Terms with magnitude smaller than this are not returned by *items()*.

approx_equal(*other*, *rtol=1.0e-5*, *atol=1.0e-8*, *equal_nan=False*)

Checks if the two objects are numerically equal.

Input parameters propagated to `numpy.allclose()`.

Parameters

- **other** (*PySCFChemistryRestrictedIntegralOperator*) – Integrals for comparison to (must be of the same type).
- **rtol** (float, default: 1.0e-5) – Relative tolerance.
- **atol** (float, default: 1.0e-8) – Absolute tolerance.
- **equal_nan** (bool, default: False) – Whether to compare NaN's as equal.

Returns
bool – True if operators are numerically equal within tolerances, False otherwise.

copy()

Performs a deep copy of object.

Return type
BaseChemistryIntegralOperator

df()

Returns a `pandas.DataFrame` object showing all terms in the operator.

Return type
DataFrame

double_factorize(*tol1=-1*, *tol2=None*, *diagonalize_one_body=True*, *diagonalize_one_body_offset=True*, *combine_one_body_terms=True*)

Double factorizes the two-electron integrals and returns the hamiltonian in diagonal form.

Writes the hamiltonian as $H = H_0 + H_1 + S + V$ where $S + V$ is the coulomb interaction. $V = (1/2)\sum_{ijkl}(ij|kl)a_i^\dagger a_j a_k^\dagger a_l$ is a reordered two-body operator and S is a one-body-like energy offset given by $S = -(1/2)\sum_{ij} s_{ij} a_i^\dagger a_j$ where $s_{ij} = \sum_k (ik|kj)$. H_0 and H_1 are the constant and one-electron terms respectively.

First level of factorization is an eigenvalue decomposition of the ERI tensor: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. The second factorization is diagonalization of the V_{pq}^t matrix for each t: $V_{pq}^t = \sum_u^{N_\lambda^t} U_{pu}^t \lambda_u^t U_{qu}^t$. At each diagonalization truncation is performed by discarding eigenvalues, starting from the smallest, until the sum of those discarded exceeds the threshold.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} - s_{pq}/2 = \sum_r W_{pr}\omega_r W_{qr}$. One-body diagonalization is not truncated.

Warning: Not intended for reduction of classical memory usage, only for truncating the two-body terms of the hamiltonian for quantum simulation.

Parameters

- **tol1** (float, default: -1) – Truncation threshold for first diagonalization of ERI matrix. If negative, no truncation is performed.
- **tol2** (Optional[float], default: None) – Truncation threshold for second diagonalization of ERI matrix. If None, same as tol1. If negative, no truncation is performed.
- **diagonalize_one_body** (Optional[bool], default: True) – Whether to diagonalize the physical one-body integrals h_pq.
- **diagonalize_one_body_offset** (Optional[bool], default: True) – Whether to diagonalize the one-body offset integrals s_{pq}
- **combine_one_body_terms** (Optional[bool], default: True) – Whether to consolidate the one-body and one-body offset integrals into effective one-body integrals. Requires diagonalize_one_body == diagonalize_one_body_offset.

Returns

DoubleFactorizedHamiltonian –:

Hamiltonian operator storing two-body integrals in double factorized form and, optionally, diagonalised
one-body integrals.

effective_potential(*rdm1*)

Calculates the effective Coulombic potential for a given 1-RDM.

Calculates only the potential due to the total density, RDM1a + RDM1b.

Parameters
rdm1 (*RestrictedOneBodyRDM*) – Restricted, one-body reduced density matrix object.

Returns
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Effective potential matrix.

effective_potential_spin(*rdm1*)

Calculates the contribution to the effective Coulomb potential due to a spin imbalance.

RDM1a - RDM1b contribution to the effective potential matrix.

Parameters
rdm1 (*RestrictedOneBodyRDM*) – Restricted, one-body reduced density matrix object.

Returns
ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]] – Effective potential matrix.

energy(*rdm1*, *rdm2=None*)

Calculate total energy based on the one- and two-body reduced density matrices.

If rdm2 is not given, this method returns the mean-field energy.

Warning: This method computes the full, rank-4 two-body integrals tensor and stores in memory to calculate the electron interaction energy.

Parameters
- **rdm1** (*RestrictedOneBodyRDM*) – Restricted, one-body reduced density matrix object.
- **rdm2** (Optional[*RestrictedTwoBodyRDM*], default: None) – Restricted, two-body reduced density matrix object.

Returns
float – Total energy.

energy_electron_mean_field(*rdm1*)

Calculates the electronic energy in the mean-field approximation.

Parameters
rdm1 (*RestrictedOneBodyRDM*) – Restricted, one-body reduced density matrix object.

Returns
Tuple[float, float] – Mean-field electronic energy (1e + 2e), and Mean-field Coulomb contribution (2e).

is_openshell()

Returns True if the PySCF mean-field object is of type ROHF, False otherwise.

Return type
bool

items(*yield_constant=True*, *yield_one_body=True*, *yield_two_body=True*)

Generates the constant, one- and two-body operator terms contained in the operator object.

Parameters
- **yield_constant** (bool, default: True) – Whether to generate a constant term.
- **yield_one_body** (bool, default: True) – Whether to generate one-body terms.
- **yield_two_body** (bool, default: True) – Whether to generate two-body terms.

Yields
Next requested FermionOperatorString and constant/integral value.

Return type
Generator[Tuple[*FermionOperatorString*, float], None, None]

classmethod load_h5(*name*)

Loads operator object from .h5 file.

Parameters
name (Union[str, Group]) – Name of .h5 file to be loaded.

Returns
BaseChemistryIntegralOperator – Loaded integral operator object.

print_table()

Prints operator terms in a table format.

Return type
None

qubit_encode(*mapping=None*)

Performs qubit encoding (mapping) using the provided mapping class of the current integral operator.

Parameters
mapping (Optional[QubitMapping], default: None) – Mapping function. Default mapping procedure is Jordan-Wigner.

Returns
Union[QubitOperator, QubitOperatorList, List, ndarray] – Mapped QubitOperator object.

rotate(*rotation*)

Performs an in-place unitary rotation of the chemistry integrals.

Rotation must be real-valued (orthogonal). In practice, this rotates the MO coefficient matrix.

Raises
ValueError – If dimensions of rotation matrix are not compatible with integrals.

Parameters
rotation (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – Orthogonal rotation matrix.

Returns
PySCFChemistryRestrictedIntegralOperator – self after molecular orbital rotation.

run_rhf()

Run a restricted Hartree-Fock calculation.

Raises
RuntimeError – If the RHF calculation does not converge.

Returns
ndarray[Any, dtype[float]] – MO coefficient matrix.

save_h5(*name*)

Dumps operator object to .h5 file.

Parameters
name (Union[str, Group]) – Destination filename of .h5 file.

Return type
None

to_ChemistryRestrictedIntegralOperator()

Convert to a core InQuanto `ChemistryRestrictedIntegralOperator` object.

Output object stores integrals in the MO basis explicitly.

Returns

ChemistryRestrictedIntegralOperator – Core InQuanto integral operator.

to_FermionOperator(*yield_constant=True*, *yield_one_body=True*, *yield_two_body=True*)

Converts chemistry integral operator to `FermionOperator`.

Returns

FermionOperator – Fermion operator form of integral operator.

Parameters

- **yield_constant** (`bool`, default: `True`) –
- **yield_one_body** (`bool`, default: `True`) –
- **yield_two_body** (`bool`, default: `True`) –

class PySCFChemistryUnrestrictedIntegralOperator(*mol_or_mf*)

Handles a (unrestricted-orbital) chemistry integral operator.

Stores a PySCF `pyscf.scf.uhf.UHF` object, and uses it to generate the constant, one- and two-body spatial integrals. All indices are in chemistry notation, `two_body[p, q, r, s]` = (pq|rs).

Raises

- **ValueError** – If the input `mol_or_mf` is not of type `pyscf.gto.Mole` or `pyscf.scf.uhf.UHF`.
- **RuntimeError** – If the PySCF self-consistent Hartree-Fock calculation does not converge.

Parameters

mol_or_mf (`Union`[`Mole`, `UHF`]) – Input PySCF object from which the molecular orbital integrals are computed.

TOLERANCE = 1e-08

Internal tolerance used when iterating over terms. Terms with magnitude smaller than this are not returned by *items()*.

approx_equal(*other*, *rtol=1.0e-5*, *atol=1.0e-8*, *equal_nan=False*)

Checks if the two objects are numerically equal.

Input parameters propagated to `numpy.allclose()`.

Parameters

- **other** (*PySCFChemistryUnrestrictedIntegralOperator*) – Integrals for comparison to (must be of the same type).
- **rtol** (`float`, default: `1.0e-5`) – Relative tolerance.
- **atol** (`float`, default: `1.0e-8`) – Absolute tolerance.
- **equal_nan** (`bool`, default: `False`) – Whether to compare NaN's as equal.

Returns

`bool` – True if operators are numerically equal within tolerances, False otherwise.

copy()

Performs a deep copy of object.

Return type
BaseChemistryIntegralOperator

df()

Returns a pandas.DataFrame object showing all terms in the operator.

Return type
DataFrame

double_factorize(*tol1=-1*, *tol2=None*, *diagonalize_one_body=True*, *diagonalize_one_body_offset=True*, *combine_one_body_terms=True*)

Double factorizes the two-electron integrals and returns the hamiltonian in diagonal form.

Writes the hamiltonian as $H = H_0 + H_1 + S + V$ where $S + V$ is the coulomb interaction. $V = (1/2)\sum_{ijkl}(ij|kl)a_i^\dagger a_j a_k^\dagger a_l$ is a reordered two-body operator and S is a one-body-like energy offset given by $S = -(1/2)\sum_{ij} s_{ij} a_i^\dagger a_j$ where $s_{ij} = \sum_k (ik|kj)$. H_0 and H_1 are the constant and one-electron terms respectively.

First level of factorization is an eigenvalue decomposition of the ERI tensor: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. The second factorization is diagonalization of the V_{pq}^t matrix for each t: $V_{pq}^t = \sum_u^{N_\lambda^t} U_{pu}^t \lambda_u^t U_{qu}^t$. At each diagonalization truncation is performed by discarding eigenvalues, starting from the smallest, until the sum of those discarded exceeds the threshold.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} - s_{pq}/2 = \sum_r W_{pr}\omega_r W_{qr}$. One-body diagonalization is not truncated.

Warning: Not intended for reduction of classical memory usage, only for truncating the two-body terms of the hamiltonian for quantum simulation.

Parameters

- **tol1** (float, default: -1) – Truncation threshold for first diagonalization of ERI matrix. If negative, no truncation is performed.
- **tol2** (Optional[float], default: None) – Truncation threshold for second diagonalization of ERI matrix. If None, same as tol1. If negative, no truncation is performed.
- **diagonalize_one_body** (Optional[bool], default: True) – Whether to diagonalize the physical one-body integrals h_pq.
- **diagonalize_one_body_offset** (Optional[bool], default: True) – Whether to diagonalize the one-body offset integrals s_{pq}
- **combine_one_body_terms** (Optional[bool], default: True) – Whether to consolidate the one-body and one-body offset integrals into effective one-body integrals. Requires diagonalize_one_body == diagonalize_one_body_offset.

Returns

DoubleFactorizedHamiltonian –:

Hamiltonian operator storing two-body integrals in double factorized form and, optionally, diagonalised
one-body integrals.

effective_potential(*rdm1*)

Calculates the effective Coulombic potential for a given 1-RDM.

Parameters
rdm1 (*UnrestrictedOneBodyRDM*) – Unrestricted, one-body reduced density matrix.

Returns
List[ndarray] – Effective potentials for the alpha and beta spin channels.

energy(*rdm1*, *rdm2=None*)

Calculate total energy based on the one- and two-body reduced density matrices.

If rdm2 is not given, this method returns the mean-field energy.

Warning: This method computes the full, rank-4 two-body integrals tensor and stores in memory to calculate the electron interaction energy.

Parameters

- **rdm1** (*UnrestrictedOneBodyRDM*) – Unrestricted, one-body reduced density matrix object.
- **rdm2** (Optional[*UnrestrictedTwoBodyRDM*], default: None) – Unrestricted, two-body reduced density matrix object.

Returns
float – Total energy.

energy_electron_mean_field(*rdm1*)

Calculates the electronic energy in the mean-field approximation.

Parameters
rdm1 (*UnrestrictedOneBodyRDM*) – Unrestricted, one-body reduced density matrix object.

Returns
Tuple[float, float] – Mean-field electronic energy (1e + 2e), and Mean-field Coulomb contribution (2e).

items(*yield_constant=True*, *yield_one_body=True*, *yield_two_body=True*)

Generates the constant, one- and two-body operator terms contained in the operator object.

Parameters

- **yield_constant** (bool, default: True) – Whether to generate a constant term.
- **yield_one_body** (bool, default: True) – Whether to generate one-body terms.
- **yield_two_body** (bool, default: True) – Whether to generate two-body terms.

Yields
Next requested FermionOperatorString and constant/integral value.

Return type
Generator[Tuple[*FermionOperatorString*, float], None, None]

classmethod load_h5(*name*)

Loads operator object from .h5 file.

Parameters
name (Union[str, Group]) – Name of .h5 file to be loaded.

Returns
BaseChemistryIntegralOperator – Loaded integral operator object.

print_table()

Prints operator terms in a table format.

Return type
None

qubit_encode(*mapping=None*)

Performs qubit encoding (mapping) using the provided mapping class of the current integral operator.

Parameters
mapping (Optional[QubitMapping], default: None) – Mapping function. Default mapping procedure is Jordan-Wigner.

Returns
Union[QubitOperator, QubitOperatorList, List, ndarray] – Mapped QubitOperator object.

rotate(*rotation_aa*, *rotation_bb*)

Performs an in-place rotation of the chemistry integrals.

Each spin block is rotated separately. Rotation matrices must be real-valued (orthogonal). In practice, this rotates the MO coefficient matrices.

Raises
ValueError – If dimensions of rotation matrices are not compatible with integrals.

Parameters

- **rotation_aa** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – Orthogonal rotation matrix for the alpha spin block.
- **rotation_bb** (ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]) – Orthogonal rotation matrix for the beta spin block.

Returns
PySCFChemistryUnrestrictedIntegralOperator – self after molecular orbital rotation.

save_h5(*name*)

Dumps operator object to .h5 file.

Parameters
name (Union[str, Group]) – Destination filename of .h5 file.

Return type
None

to_ChemistryUnrestrictedIntegralOperator()

Convert to a core InQuanto ChemistryUnrestrictedIntegralOperator object.

Output object stores integrals in the MO basis explicitly.

Returns
ChemistryUnrestrictedIntegralOperator – Core InQuanto integral operator.

to_FermionOperator(*yield_constant=True*, *yield_one_body=True*, *yield_two_body=True*)

Converts chemistry integral operator to `FermionOperator`.

Returns

FermionOperator – Fermion operator form of integral operator.

Parameters

- **yield_constant** (`bool`, default: `True`) –
- **yield_one_body** (`bool`, default: `True`) –
- **yield_two_body** (`bool`, default: `True`) –

get_correlation_potential_pattern(*driver*, **atoms_list*)

Get correlation potential pattern.

Parameters

- **driver** (`BasePySCFDriverMolecular`) – Molecular PySCF driver.
- ***atoms_list** (`List[int]`) – List of atom indices.

Returns

`ndarray[Any, dtype[TypeVar(ScalarType, bound= generic, covariant=True)]]` – Pattern matrix.

get_fragment_orbital_masks(*driver*, **atoms_list*)

Get fragment orbital masks.

Parameters

- **driver** (`BasePySCFDriverMolecular`) – Molecular PySCF driver.
- ***atoms_list** (`List[int]`) – List of atom indices.

Returns

`Tuple[ndarray[Any, dtype[bool]], ...]` – Fragment orbital mask.

get_fragment_orbitals(*driver*, **atoms_list*)

Get fragment orbitals.

Parameters

- **driver** (`BasePySCFDriverMolecular`) – Molecular PySCF driver
- ***atoms_list** (`List[int]`) – List of atom indices.

Returns

`DataFrame` – Orbitals table in a DataFrame.

23.1.1 inquanto.extensions.pyscf.fmo

Module for Fragment Molecular Orbital (FMO) calculations of orthogonalized second quantized systems.

class FMO(*hamiltonian_operator*, *scf_max_iteration=20*, *scf_tolerance=1e-5*)

Bases: `object`

BasicFMO driver class.

- Kitaura, Kazuo, Eiji Ikeo, Toshio Asada, Tatsuya Nakano, and Masami Uebayasi. "Fragment molecular orbital method: an approximate computational method for large molecules." Chemical Physics Letters 313, no. 3-4 (1999): 701-706.

- Yamazaki, Takeshi, Shunji Matsuura, Ali Narimani, Anushervon Saidmuradov, and Arman Zaribafiyan. "Towards the practical application of near-term quantum computers in quantum chemistry simulations: A problem decomposition approach." arXiv preprint arXiv:1806.01305 (2018).

Parameters

- **hamiltonian_operator** (Union[*ChemistryRestrictedIntegralOperator*, *PySCFChemistryRestrictedIntegralOperator*]) – Stores integrals for full system in orthogonal localized basis.
- **scf_max_iteration** (int, default: 20) – Max iteration for the outer scf loop.
- **scf_tolerance** (float, default: 1e-5) – Convergence tolerance for the outer loop for $|energy - energy'| < scf_tolerance$.

static ao_mask_2_atom_mask(*mol*, *ao_mask*)

Convert a mask covering the list of atomic orbitals (AOs) to a mask covering all atoms in the system.

Parameters

- **mol** (Mole) – PySCF gto.Mole object describing the molecular system.
- **ao_mask** (List[bool]) – List of bool indicating if each AO is included or not.

Raises

ValueError – If the ao_mask fragments a single atom.

Returns

List[bool] – Atomic mask corresponding to input AO mask.

static atom_mask_2_ao_mask(*mol*, *atom_mask*)

Convert a mask covering the list of atoms to a mask covering all atomic orbitals (AOs) in the system.

Parameters

- **mol** (Mole) – PySCF gto.Mole object describing the molecular system.
- **atom_mask** (List[bool]) – List of bool indicating if each atom is included or not.

Returns

List[bool] – AO mask corresponding to input atomic mask.

energy(*fragments*, *dimer_fragments*)

Computes the total FMO2 energies.

Parameters

- **fragments** (List[*FMOFragment*]) – list of FMO fragment solver
- **dimer_fragments** (Dict[Tuple, *FMOFragment*]) – dic of FMO fragment solver for the dimers

Returns

energy

run(*fragments*)

Running the FMO self-consistent cycles.

Parameters

fragments (List[*FMOFragment*]) – List of FMO fragment solvers.

Returns
float – Total energy.

class FMOFragment (*fmo*, *mask*, *n_electron*, *name=None*)

Bases: object

Base solver for an FMO fragment.

Default solver is Restricted Hartree-Fock.

Parameters

- **fmo** (FMO) – Fragment Molecular Orbital calculator.
- **mask** (Union[List[bool], ndarray[Any, dtype[bool]]]) – Orbital fragment mask.
- **n_electron** (int) – Number of electrons in fragment.
- **name** (str, default: None) – Reference name for fragment.

classmethod compose_fragments (*fragment*, **fragments*)

Combines two or more fragments into a single fragment.

Parameters

- **fragment** (*FMOFragment*) – A fragment solver.
- ***fragments** (*FMOFragment*) – Other fragment solvers to merge.

Returns
FMOFragment – A new fragment solver.

solve (*hamiltonian_operator*)

Compute the energy and 1-RDM of the fragment.

Used in the self-consistent FMO cycle.

Note: Default solver is RHF (does not use PySCF).

Parameters
hamiltonian_operator (*ChemistryRestrictedIntegralOperator*) – Integral operator for the fragment.

Returns
Tuple[float, *RestrictedOneBodyRDM*] – Electronic energy and 1-RDM of the fragment.

solve_final (*hamiltonian_operator*)

Compute the final energy of the fragment.

Used after FMO self-consistency is reached, to compute final energies.

Note: Unless overwritten it does the same as solve(…).

Parameters
hamiltonian_operator (*ChemistryRestrictedIntegralOperator*) – Integral operator for the fragment.

Returns
`float` – Electronic energy.

class FMOFragmentPySCFActive(*fmo*, *mask*, *n_electron*, *name=None*, *frozen=None*)

Bases: *FMOFragmentPySCFRHF*

PySCF custom active space solver for FMO fragments.

Uses RHF as the solver in the FMO self-consistent cycle, and a user defined solver for the final energy calculation.

Parameters

- **fmo** (`FMO`) – Fragment Molecular Orbital calculator.
- **mask** (`Union[List[bool], ndarray[Any, dtype[bool]]]`) – Orbital fragment mask.
- **n_electron** (`int`) – Number of electrons in fragment.
- **name** (`str`, default: `None`) – Reference name for fragment.
- **frozen** (`Union[List[int], Callable[[SCF], List[int]]]`, default: `None`) – Frozen orbital information passed to PySCf driver.

classmethod compose_fragments(*fragment*, **fragments*)

Combines two or more fragments into a single fragment.

Parameters

- **fragment** (*FMOFragment*) – A fragment solver.
- ***fragments** (*FMOFragment*) – Other fragment solvers to merge.

Returns
FMOFragment – A new fragment solver.

solve(*hamiltonian_operator*)

Compute the energy and 1-RDM of the fragment using the PySCF RHF solver.

Used in the self-consistent FMO cycle.

Parameters
hamiltonian_operator (`Union[ChemistryRestrictedIntegralOperator, PySCFChemistryRestrictedIntegralOperator]`) – Integral operator for the fragment.

Returns
`Tuple[float, RestrictedOneBodyRDM]` – Electronic energy and 1-RDM of the fragment.

solve_final(*hamiltonian_operator*)

Compute the final energy of the fragment using the custom active space solver.

Used after FMO self-consistency is reached, to compute final energies.

Parameters
hamiltonian_operator (`Union[ChemistryRestrictedIntegralOperator, PySCFChemistryRestrictedIntegralOperator]`) – Integral operator for the fragment.

Returns
Electronic energy of the fragment.

solve_final_active(*hamiltonian_operator*, *fermion_space*, *fermion_state*)

Compute the final energy of an active space in the fragment.

Used by *FMOFragmentPySCFActive.solve_final()*.

Note: Must be implemented by user.

Parameters

- **fermion_space** (*FermionSpace*) – Fermion active space.
- **fermion_state** (*FermionState*) – Fermion state.
- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) –

Returns

Tuple[float, *RestrictedOneBodyRDM*] – Electronic energy.

class FMOFragmentPySCFCCSD(*fmo*, *mask*, *n_electron*, *name=None*, *frozen=None*)

Bases: *FMOFragmentPySCFRHF*

PySCF RHF-CCSD solver for FMO fragments.

Uses RHF as the solver in the FMO self-consistent cycle, and CCSD for the final energy calculation.

Parameters

- **fmo** (FMO) – Fragment Molecular Orbital calculator.
- **mask** (Union[List[bool], ndarray[Any, dtype[bool]]]) – Orbital fragment mask.
- **n_electron** (int) – Number of electrons in fragment.
- **name** (str, default: None) – Reference name for fragment.

classmethod compose_fragments(*fragment*, **fragments*)

Combines two or more fragments into a single fragment.

Parameters

- **fragment** (*FMOFragment*) – A fragment solver.
- ***fragments** (*FMOFragment*) – Other fragment solvers to merge.

Returns

FMOFragment – A new fragment solver.

solve(*hamiltonian_operator*)

Compute the energy and 1-RDM of the fragment using the PySCF RHF solver.

Used in the self-consistent FMO cycle.

Parameters

hamiltonian_operator (Union[*ChemistryRestrictedIntegralOperator*, *PySCFChemistryRestrictedIntegralOperator*]) – Integral operator for the fragment.

Returns

Tuple[float, *RestrictedOneBodyRDM*] – Electronic energy and 1-RDM of the fragment.

solve_final (*hamiltonian_operator*)

Compute the final energy of the fragment using the PySCF CCSD solver.

Used after FMO self-consistency is reached, to compute final energies.

Parameters

hamiltonian_operator (Union[*ChemistryRestrictedIntegralOperator*, *PySCFChemistryRestrictedIntegralOperator*]) – Integral operator for the fragment.

Returns

float – Electronic energy of the fragment.

class FMOFragmentPySCFMP2 (*fmo*, *mask*, *n_electron*, *name=None*, *frozen=None*)

Bases: *FMOFragmentPySCFRHF*

PySCF RHF-MP2 solver for FMO fragments.

Uses RHF as the solver in the FMO self-consistent cycle, and MP2 for the final energy calculation.

Parameters

- **fmo** (FMO) – Fragment Molecular Orbital calculator.
- **mask** (Union[List[bool], ndarray[Any, dtype[bool]]]) – Orbital fragment mask.
- **n_electron** (int) – Number of electrons in fragment.
- **name** (str, default: None) – Reference name for fragment.

classmethod compose_fragments (*fragment*, **fragments*)

Combines two or more fragments into a single fragment.

Parameters

- **fragment** (*FMOFragment*) – A fragment solver.
- ***fragments** (*FMOFragment*) – Other fragment solvers to merge.

Returns

FMOFragment – A new fragment solver.

solve (*hamiltonian_operator*)

Compute the energy and 1-RDM of the fragment using the PySCF RHF solver.

Used in the self-consistent FMO cycle.

Parameters

hamiltonian_operator (Union[*ChemistryRestrictedIntegralOperator*, *PySCFChemistryRestrictedIntegralOperator*]) – Integral operator for the fragment.

Returns

Tuple[float, *RestrictedOneBodyRDM*] – Electronic energy and 1-RDM of the fragment.

solve_final (*hamiltonian_operator*)

Compute the final energy of the fragment using the PySCF MP2 solver.

Used after FMO self-consistency is reached, to compute final energies.

Parameters

hamiltonian_operator (Union[*ChemistryRestrictedIntegralOperator*,

PySCFChemistryRestrictedIntegralOperator]) – Integral operator for the fragment.

Returns
Electronic energy of the fragment.

class FMOFragmentPySCFRHF (*fmo*, *mask*, *n_electron*, *name=None*)

Bases: *FMOFragment*

PySCF RHF solver for FMO fragments.

Parameters

- **fmo** (FMO) – Fragment Molecular Orbital calculator.
- **mask** (Union[List[bool], ndarray[Any, dtype[bool]]]) – Orbital fragment mask.
- **n_electron** (int) – Number of electrons in fragment.
- **name** (str, default: None) – Reference name for fragment.

classmethod compose_fragments (*fragment*, **fragments*)

Combines two or more fragments into a single fragment.

Parameters

- **fragment** (*FMOFragment*) – A fragment solver.
- ***fragments** (*FMOFragment*) – Other fragment solvers to merge.

Returns
FMOFragment – A new fragment solver.

solve (*hamiltonian_operator*)

Compute the energy and 1-RDM of the fragment using the PySCF RHF solver.

Used in the self-consistent FMO cycle.

Parameters
hamiltonian_operator (Union[*ChemistryRestrictedIntegralOperator*, *PySCFChemistryRestrictedIntegralOperator*]) – Integral operator for the fragment.

Returns
Tuple[float, *RestrictedOneBodyRDM*] – Electronic energy and 1-RDM of the fragment.

solve_final (*hamiltonian_operator*)

Compute the final energy of the fragment.

Used after FMO self-consistency is reached, to compute final energies.

Note: Unless overwritten it does the same as solve(…).

Parameters
hamiltonian_operator (*ChemistryRestrictedIntegralOperator*) – Integral operator for the fragment.

Returns
float – Electronic energy.

23.2 inquanto-nglview

InQuanto NGLView extension.

class VisualizerNGL(*geometry*, *background_color='#FFFFFF'*)

NGLView visualizer for inquanto.

Parameters

- **geometry** (Union[List[Tuple[str, Tuple[float, ...]]], *GeometryPeriodic*, *GeometryMolecular*]) – Molecular or unit cell geometry.
- **background_color** (str, default: "#FFFFFF") – Background color of NGLView stage. Used as the backgroundColor NGL stage parameter, so may be a preset color name i.e. "red", "white" etc. or sRGB code.

visualize_fragmentation(*source*, *atom_labels=None*, *fragment_colors=None*, *fragment_color_seed=6*)

Visualize a DMET fragmentation scheme of the molecule.

Fragmentation groups should be defined in the molecular geometry with the Geometry.set_groups() method.

Note: Not compatible with periodic geometries.

Parameters

- **source** (str) – The heading of the column defining the fragmentation in the geometry dataframe, Geometry.df().
- **atom_labels** (Optional[str], default: None) – Label type for atoms, can be "index", "symbol" or None.
- **fragment_colors** (Optional[List[str]], default: None) – Color of each fragment. Formatted for the NGL add_ball_and_stick() method, so may be preset color names i.e. ["red", "blue"...] or sRGB code. Order corresponds to ordering of fragments in geometry dataframe geometry.df[source].
- **fragment_color_seed** (Optional[int], default: 6) – RNG seed for generating fragment colors. Used only if fragment_colors is None.

Returns

NGLWidget – An ngl widget showing the molecule with fragments highlighted.

visualize_molecule(*atom_labels=None*)

Generate a ball-and-stick visualization of the molecule.

Parameters

atom_labels (Optional[str], default: None) – Label type for atoms, can be "index", "symbol" or None.

Returns

NGLWidget – An ngl widget showing the molecule.

visualize_orbitals(*cube_orbitals*, *atom_labels=None*, *red_isolevel=-0.55*, *blue_isolevel=0.55*)

Visualize the input orbital atop the molecular or periodic structure.

Parameters

- **cube_orbitals** (str) – A string containing the contents of a .cube file, which details the shape of an orbital.
- **atom_labels** (Optional[str], default: None) – Label type for atoms, can be "index", "symbol" or None.
- **red_isolevel** (float, default: -0.55) – The isolevel at which to color the orbital isosurface red.
- **blue_isolevel** (float, default: 0.55) – The isolevel at which to color the orbital isosurface blue.

Returns

NGLWidget – An ngl widget showing the visualized orbital, and the molecular or unit cell structure.

visualize_unit_cell(*atom_labels=None*)

Generate a ball-and-stick visualisation of the unit cell.

Draws the cell edges, and atoms inside.

Parameters

atom_labels – Label type for atoms, can be "index", "symbol" or None.

Returns

NGLWidget – An ngl widget showing the unit cell.

CHAPTER

TWENTYFOUR

CHANGELOG

24.1 2.1.1

27 April 2023

- Some performance improvements
- Fixed networkx dependence
- Fixed numpy errors due to _typing
- Other small fixes

24.2 2.1.0

27 March 2023

- Added double factorization `inquanto.operators.DoubleFactorizedHamiltonian`
- Added Real Basis Rotation Ansatz *inquanto.ansatzes.RealBasisRotationAnsatz*
- Addition of some C++ components and functionality (speed up compact integrals)
- Adding various tools for manipulating *inquanto.operators.QubitOperatorList*
- Qubitwise commutativity + improved tools for general commutativity
- Added `inquanto.operators.OperatorList.sublist()` and some *inquanto.algorithms.adapt.AlgorithmFermionicAdaptVQE* convenience methods.
- Hotfix to *inquanto.geometries.GeometryMolecular.save_xyz()*
- Extra *inquanto.algorithms.time_evolution.AlgorithmVQS* examples
- Upgrading Python support (up to 3.11) in line with pytket
- Remove restriction on pytket-qiskit dependency requirement
- *inquanto.embeddings.dmet* docstrings
- Improved Impurity DMET exact diagonalization and VQE solver
- Improved API documentation
- **inquanto-pyscf** : *inquanto.extensions.pyscf.fmo.FMO*, NEVPT2 methods e.g. *get_nevpt2_correction()*, and fixes.
- **inquanto-nglview**: improved API docs and fixes

24.3 2.0.0

5 December 2022

- Re-designed ansatzes to provide more uniform interface, allow multireference calculations and easier custom ansatz development.
- New ansatz classes added. `CircuitAnsatz`, `TrotterAnsatz`, `GivensAnsatz`, `MultiReferenceState`, `LayeredAnsatz`..
- `ProtocolStateVectorSparse` now caches results.
- New `ProtocolSymbolic` class allows for evaluation of expressions with symbolic ansatzes.
- Implementation of the QRDM-NEVPT2 method. Several new computables for RDM calculations.
- Add support for real-time evolution in AlgorithmVQS.
- New `CompactTwoBodyIntegralsS8` for eight-fold symmetric compact integrals.
- `Computable.cost_estimate` and `cost_estimate_elementwise` for circuit cost estimation on Quantinuum hardware.
- `kupCCGSpD` refactored to `kUpCCGSDSinglet`.
- `Pyscf` refactored to `PySCF`.

24.4 1.3.0

22 November 2022

- Fix `FermionSpace.construct_number_alpha_operator` and `FermionSpace.construct_number_beta_operator`.
- Add `FermionOperator.to_latex` and `QubitOperator.to_latex`.
- Add descriptions to files listed by `inquanto.express.list_h5`.
- Fix bug where the PMSV supporter was sometimes not applied.
- Add `QubitOperatorList.all_qubits` and `QubitOperatorList.to_sparse_matrices`.
- Allow `ChemistryRestrictedIntegralOperator` and `FermionOperator` to be constructed from FCI files.
- Add `QubitState.from_ndarray`.
- Remove `QubitSpace.generate_qubit_trotter_operator`.
- Add exponentiation and Trotterization functionality to `QubitOperator` and `QubitOperatorList`.

24.5 1.2.2

22 September 2022

- Fixed a typo in auxiliary expressions in ADAPT and VQS.

24.6 1.2.1

21 September 2022

- Added import for QubitMappingParity at module level.

24.7 1.2.0

16 September 2022

- Added optional compiler_passes argument to `Computable.run()`.
- Logger configured.
- Workflow to synchronise branches develop and research-develop improved.
- Compact 2-body integrals implemented.
- `ChemistryUnrestrictedIntegralOperator._freeze()` implemented.
- Ansatzes deep copy implemented.
- HVA fixed.
- Convenience method to convert `FermionOperator` to integral operators implemented.
- Imaginary gradients sign fixed in protocols.
- Various `FermionOperator` helper methods implemented (`is_hermitian`, `is_antihermitian`, `is_self_inverse` etc.).
- Disable `QubitOperatorString` to be constructed with two Paulis on the same qubit.
- `save_h5` and `load_h5` implemented for spaces classes.
- Added `custom_prefix` for circuit names to `Computable.run()`, `.launch()` and `.generate_circuits()`.
- Improvements to API documentation and type hints: Ansatzes, Algorithms, Mappings, Core, Geometries, Operators, Symmetry, Embeddings.
- Fix exponent order issue in `FermionSpaceStateExpChemicallyAware`.

24.8 1.1.0

29 July 2022

- `toeplitz_decomposition` added for QubitOperator.
- `ProtocolSymbolic` added for symbolic evaluation of expectation values and computables.
- Fixed a bug in `Gradients` (State vector protocols, `ProtocolPhaseShift()` and `ProtocolHadamard-DirectPauliY()`)
- Fixed triplet excitations generator (can be used in QSE now)
- General improvements in express module
  - `MetricTensor` can now be used as part of `Computables` objects
- Padding methods added to `QubitOperator`
- Improved documentation

24.9 1.0.5

04 July 2022

- Added ability for authentication by password.
- Enabled support for any pytket v1.x release.

24.10 1.0.4

15 June 2022

- Various minor bugfixes.
- Improved validation output.
- Updated sourcedefender dependency version.

24.11 1.0.3

10 June 2022

- Added additional tests for parameter classes.
- Fix for type safety in trigonometric methods.
- Allowed api-key to be fetched from keyring or environment variable.

24.12 1.0.2

01 June 2022

- Fixed bug in contracted systems symmetries.
- Added finite difference metric tensor tests.

24.13 1.0.1

25 May 2022

- Fixed bug in contracted systems symmetries.
- Fixed bug in AlgorithmVQE property.
- Fixed time-limit issue on decryption.

CHAPTER

TWENTYFIVE

BIBLIOGRAPHY

CHAPTER

TWENTYSIX

SUPPORT

Having issues using InQuanto or found a bug? Inquanto-support@quantinuum.com

CHAPTER

TWENTYSEVEN

HOW TO CITE INQUANTO

If you use InQuanto in your work, please cite with

```
@misc{
    inquanto,
    year={2022},
    author={Tranter, Andrew  and Di Paola, Cono and Mu\~{n}oz Ramo, David and␣
↪Manrique, David Zsolt and Gowland, Duncan and Plekhanov, Evgeny and Greene-Diniz,␣
↪Gabriel  and Christopoulou, Georgia and Prokopiou, Georgia and Keen, Harry D J and␣
↪Polyak, Iakov  and Khan, Irfan T and Pilipczuk, Jerzy and Kirsopp, Josh J M and␣
↪Yamamoto, Kentaro and Tudorovskaya, Maria and Krompiec, Michal and Sze, Michelle ␣
↪and Fitzpatrick, Nathan},
    title={InQuanto: Quantum Computational Chemistry},
    url={https://www.quantinuum.com/products/inquanto},
    journal={Quantinuum}
}
```

CHAPTER

TWENTYEIGHT

SOFTWARE LICENCE

For enquiries regarding access to InQuanto, please contact inquanto@quantinuum.com.

CHAPTER

TWENTYNINE

OPEN-SOURCE ATTRIBUTION

Name	Version	License	URL	Description
h5py	3.7.0	BSD License	http://www.h5py.org	Read and write HDF5 files from Python
ipywidgets	7.7.3	BSD License	http://ipython.org	IPython HTML widgets for Jupyter
jupyter	1.0.0	BSD License	http://jupyter.org	Jupyter metapackage. Install all the Jupyter components in one go.
keyring	23.11	MIT License	https://github.com/jaraco/keyring	Store and access your passwords safely.
matplotlib	3.6.2	Python Software Foundation License	https://matplotlib.org	Python plotting package
nglview	3.0.3	MIT License	https://github.com/arose/nglview	IPython widget to interactively view molecular structures and trajectories.
numpy	1.23.5	BSD License	https://www.numpy.org	NumPy is the fundamental package for array computing with Python.
openfermion	1.5.5	Apache 2	http://www.openfermion.org	The electronic structure package for quantum computers.
ordered-set	4.1.0	MIT License	https://pypi.org/project/ordered-set/	An OrderedSet is a custom MutableSet that remembers its order
pandas	1.5.2	BSD License	https://pandas.pydata.org	Powerful data structures for data analysis, time series, and statistics
pyscf	2.1.1	Apache Software License	http://www.pyscf.org	PySCF: Python-based Simulations of Chemistry Framework
pytest	7.2.2	MIT License	https://docs.pytest.org/en/latest/	pytest: simple powerful testing with Python
pytket	1.9.0	Apache Software License	https://cqcl.github.io/pytket	Python module for interfacing with the CQC tket library of quantum software
pytket-extensions	X.X.X	Apache Software License	https://github.com/CQCL/pytket-extensions	Extension for pytket, providing access to backends
scipy	1.9.3	BSD License	https://www.scipy.org	SciPy: Scientific Library for Python
sourcedefender	10.0.1	Other/Proprietary License	https://sourcedefender.co.uk/	Advanced encryption protecting your python codebase.
sympy	1.11.1	BSD License	https://sympy.org	Computer algebra system (CAS) in Python

BIBLIOGRAPHY

[1] Greene-Diniz, Gabriel, Manrique, David Zsolt, Sennane, Wassil, Magnin, Yann, Shishenina, Elvira, Cordier, Philippe, Llewellyn, Philip, Krompiec, Michal, Rančić, Marko J., and Muñoz Ramo, David. Modelling carbon capture on metal-organic frameworks with quantum computing. *EPJ Quantum Technol.*, 9(1):37, 2022. doi:10.1140/epjqt/s40507-022-00155-w.

[2] Hans Hon Sang Chan, David Muñoz-Ramo, and Nathan Fitzpatrick. Simulating non-unitary dynamics using quantum signal processing with unitary block encoding. 2023. URL: https://arxiv.org/abs/2303.06161, doi:10.48550/ARXIV.2303.06161.

[3] Yuta Kikuchi, Conor Mc Keever, Luuk Coopmans, Michael Lubasch, and Marcello Benedetti. Realization of quantum signal processing on a noisy quantum computer. 2023. URL: https://arxiv.org/abs/2303.05533, doi:10.48550/ARXIV.2303.05533.

[4] A. Szabo and N. S. Ostlund. *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*. Dover Books on Chemistry. Dover Publications, 2012. ISBN 9780486134598.

[5] Trygve Helgaker, Poul Jørgensen, and Jeppe Olsen. *Molecular Electronic-Structure Theory*. Wiley, Chichester ; New York, 2000. ISBN 978-0-471-96755-2 978-1-118-53147-1.

[6] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. A variational eigenvalue solver on a photonic quantum processor. *Nat. Commun.*, 5:4213, 2014. URL: https://doi.org/10.1038/ncomms5213, doi:10.1038/ncomms5213.

[7] Oscar Higgott, Daochen Wang, and Stephen Brierley. Variational quantum computation of excited states. *Quantum*, 3:1–11, 2019. arXiv:1805.08138, doi:10.22331/q-2019-07-01-156.

[8] J. R. McClean, M. E. Kimchi-Schwartz, J. Carter, and W. A. de Jong. Hybrid quantum-classical hierarchy for mitigation of decoherence and determination of excited states. *Phys. Rev. A*, 95():042308, 2017.

[9] Harper R. Grimsley, Sophia E. Economou, Edwin Barnes, and Nicholas J. Mayhall. An adaptive variational algorithm for exact molecular simulations on a quantum computer. *Nature Communications*, 10(1):3007, 2019. URL: https://doi.org/10.1038/s41467-019-10988-2, doi:10.1038/s41467-019-10988-2.

[10] Yordan S. Yordanov, V. Armaos, Crispin H. W. Barnes, and David R. M. Arvidsson-Shukur. Qubit-excitation-based adaptive variational quantum eigensolver. *Communications Physics*, 4(1):228, 2021. URL: https://doi.org/10.1038/s42005-021-00730-0, doi:10.1038/s42005-021-00730-0.

[11] L.-A. Wu and D. A. Lidar. Qubits as parafermions. *Journal of Mathematical Physics*, 43(9):4506–4525, 2002. URL: https://doi.org/10.1063/1.1499208, arXiv:https://doi.org/10.1063/1.1499208, doi:10.1063/1.1499208.

[12] Xiao Yuan, Suguru Endo, Qi Zhao, Ying Li, and Simon C. Benjamin. Theory of variational quantum simulation. *Quantum*, 3:191, October 2019. URL: https://doi.org/10.22331/q-2019-10-07-191, doi:10.22331/q-2019-10-07-191.

[13] Jacob T. Seeley, Martin J. Richard, and Peter J. Love. The Bravyi-Kitaev transformation for quantum computation of electronic structure. *The Journal of Chemical Physics*, 137(22):224109, December 2012. doi:10.1063/1.4768229.

[14] Rodney J. Bartlett and Monika Musiał. Coupled-cluster theory in quantum chemistry. *Reviews of Modern Physics*, 79(1):291–352, February 2007. doi:10.1103/RevModPhys.79.291.

[15] Abhinav Anand, Philipp Schleich, Sumner Alperin-Lea, Phillip W. K. Jensen, Sukin Sim, Manuel Díaz-Tinoco, Jakob S. Kottmann, Matthias Degroote, Artur F. Izmaylov, and Alán Aspuru-Guzik. A Quantum Computing View on Unitary Coupled Cluster Theory. *arXiv:2109.15176 [physics, physics:quant-ph]*, September 2021. arXiv:2109.15176.

[16] Joonho Lee, William J. Huggins, Martin Head-Gordon, and K. Birgitta Whaley. Generalized Unitary Coupled Cluster Wave functions for Quantum Computation. *Journal of Chemical Theory and Computation*, 15(1):311–324, January 2019. doi:10.1021/acs.jctc.8b01004.

[17] I. T. Khan, M. Tudorovskaya, J. J. M. Kirsopp, D. Muñoz Ramo, P Warrier, D. K. Papanastasiou, and Singh R. Chemically aware unitary coupled cluster with ab initio calculations on system model h1: a refrigerant chemicals application. *arXiv:2210.14834 [physics:quant-ph]*, 2022. arXiv:2210.14834.

[18] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M. Chow, and Jay M. Gambetta. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature*, 549(7671):242–246, September 2017. doi:10.1038/nature23879.

[19] Juan Miguel Arrazola, Olivia Di Matteo, Nicolás Quesada, Soran Jahangiri, Alain Delgado, and Nathan Killoran. Universal quantum circuits for quantum chemistry. *Quantum*, 6:742, June 2022. URL: https://doi.org/10.22331/q-2022-06-20-742, doi:10.22331/q-2022-06-20-742.

[20] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Phys. Rev. A*, 52:3457–3467, Nov 1995. URL: https://link.aps.org/doi/10.1103/PhysRevA.52.3457, doi:10.1103/PhysRevA.52.3457.

[21] Adenilton J. da Silva and Daniel K. Park. Linear-depth quantum circuits for multiqubit controlled gates. *Phys. Rev. A*, 106:042602, Oct 2022. URL: https://link.aps.org/doi/10.1103/PhysRevA.106.042602, doi:10.1103/PhysRevA.106.042602.

[22] Sergey Bravyi, Jay M. Gambetta, Antonio Mezzacapo, and Kristan Temme. Tapering off qubits to simulate fermionic Hamiltonians. *arXiv:1701.08213 [quant-ph]*, January 2017. arXiv:1701.08213.

[23] Mateusz Ostaszewski, Edward Grant, and Marcello Benedetti. Structure optimization for parameterized quantum circuits. *Quantum*, 5:391, January 2021. doi:10.22331/q-2021-01-28-391.

[24] James Stokes, Josh Izaac, Nathan Killoran, and Giuseppe Carleo. Quantum Natural Gradient. *Quantum*, 4:269, May 2020. doi:10.22331/q-2020-05-25-269.

[25] Gerald Knizia and Garnet Kin-Lic Chan. Density matrix embedding: a simple alternative to dynamical mean-field theory. *Physical review letters*, 109(18):186404, 2012.

[26] Gerald Knizia and Garnet Kin-Lic Chan. Density matrix embedding: a strong-coupling quantum embedding theory. *Journal of chemical theory and computation*, 9(3):1428–1432, 2013.

[27] Hans Martin Senn and Walter Thiel. QM/MM Methods for Biomolecular Systems. *Angewandte Chemie International Edition*, 48(7):1198–1229, 2009. doi:10.1002/anie.200802019.

[28] Lili Cao and Ulf Ryde. On the Difference Between Additive and Subtractive QM/MM Calculations. *Frontiers in Chemistry*, 2018.

[29] A. Klamt and G. Schüürmann. COSMO: a new approach to dielectric screening in solvents with explicit expressions for the screening energy and its gradient. *Journal of the Chemical Society, Perkin Transactions 2*, pages 799–805, January 1993. doi:10.1039/P29930000799.

[30] Filippo Lipparini, Giovanni Scalmani, Louis Lagardère, Benjamin Stamm, Eric Cancès, Yvon Maday, Jean-Philip Piquemal, Michael J. Frisch, and Benedetta Mennucci. Quantum, classical, and hybrid QM/MM calculations in solution: General implementation of the ddCOSMO linear scaling strategy. *The Journal of Chemical Physics*, 141(18):184108, November 2014. doi:10.1063/1.4901304.

PYTHON MODULE INDEX

i

INDEX

Non-alphabetical

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z