
InQuanto™

Release 3.7.0

Quantinuum™

Oct 30, 2024

INTRODUCTION

1	What is InQuanto?	2
1.1	Why use InQuanto?	2
1.2	How it works	3
1.3	Powered by TKET™	3
2	Installing InQuanto	5
2.1	System Requirements	5
2.2	Troubleshooting	6
3	Quick-start guide	7
3.1	Express database	7
3.2	VQE wrapper function	7
4	How to use InQuanto	10
4.1	Chemistry Workflows	10
4.2	Preparing Mean-Field Systems	12
4.3	Spaces, Operators, States and Mappings	12
4.4	Running Computables and Algorithms	12
4.5	Expert use of InQuanto	12
5	Algorithms	14
5.1	Variational Algorithms	15
5.2	Non-variational and Phase Estimation algorithms	25
5.3	Time evolution algorithms	37
6	Computables	42
6.1	Basic Usage and Composability	43
6.2	Evaluating Computables with Protocols	44
6.3	Composite Computables	50
6.4	Primitive Computables	58
6.5	Custom Computables & Partial Evaluations	60
7	Protocols	63
7.1	Protocols for expectation values	63
7.2	Protocols for overlap squared	67
7.3	Protocols for overlaps	71
7.4	Protocols for derivatives	78
7.5	Other averaging protocols	82
7.6	Protocols for Phase Estimation	83
7.7	Resource estimation	90

8	Noise mitigation	96
8.1	Using Qermit	97
8.2	Using InQuanto's PMSV and SPAM	98
9	Spaces, Operators, and States	100
9.1	Fermionic Spaces	101
9.2	Qubit spaces, operators and states	107
9.3	Fermion-to-Qubit Mapping	111
9.4	Integral Operators	114
9.5	Orbital Transformation and Optimization	116
9.6	Double Factorization	118
10	Ansatzes	124
10.1	Trotter based	124
10.2	Basic ansatz	131
10.3	Other Ansatz	133
11	Symmetry	142
11.1	Finding symmetries	143
11.2	Point Group Symmetry	144
11.3	Z2 Tapering	145
12	Geometry	147
12.1	Molecular Systems	147
12.2	Periodic systems	151
13	Classical Minimizers	153
13.1	MinimizerScipy	153
13.2	MinimizerRotosolve	155
13.3	MinimizerSGD	156
13.4	Minimizer SPSA	158
14	Density Matrix Embedding Theory	159
14.1	Impurity DMET	159
14.2	One-shot DMET	161
14.3	Full DMET with correlation matrix	162
14.4	Custom fragments	163
14.5	DMET for model systems and other Hamiltonians	164
15	Express	165
15.1	Express methods	165
15.2	Model hamiltonians	167
15.3	List of Express files	168
16	Tutorials	172
17	Core Tutorials	173
17.1	A basic VQE simulation	173
17.2	Extended VQE	181
17.3	Variational Quantum Deflation for excited states	189
17.4	Visualization with inquanto-nglview	193
18	Backend Tutorials	196
18.1	Running on the Aer simulator	196
18.2	Running on Quantinuum H-Series	203
18.3	Quantinuum H-Series - Launching circuits and retrieving results	208

18.4	Quantinuum H-Series - Quantum Subspace Expansion	213
18.5	Quantinuum H-Series	220
18.6	IBMQ Setup	222
19	Case Study Tutorials - Fe4N2	228
19.1	Fe4N2 - 1 - system construction with AVAS and CASSCF	228
19.2	Fe4N2 - 2 - circuit construction with ADAPT-VQE	235
19.3	Fe4N2 - 3 - calculations on the H-series	239
20	Fragmentation Tutorials	248
20.1	Tackling larger systems with fragmentation	248
20.2	Projection-based embedding	252
20.3	NEVPT2 and AC0 energy corrections	255
20.4	Projection-based embedding with energy corrections	263
21	Overview of examples	269
21.1	algorithms/adapt	269
21.2	algorithms/qse	269
21.3	algorithms/time_evolution	269
21.4	algorithms/vqd	270
21.5	algorithms/vqe	270
21.6	ansatzes	270
21.7	computables	270
21.8	computables/atomic	271
21.9	computables/composite	271
21.10	computables/composite/gf	271
21.11	computables/composite/rdm	272
21.12	computables/primitive	272
21.13	core	273
21.14	embeddings	273
21.15	express	273
21.16	mappings	273
21.17	minimizers	273
21.18	operators	274
21.19	protocols	274
21.20	protocols/phase_estimation	275
21.21	spaces	275
21.22	symmetry	276
22	InQuanto-Extensions	277
23	InQuanto-PySCF	278
23.1	Basic usage	278
23.2	Generating a driver from a PySCF object	279
23.3	Periodic systems	280
23.4	Classical post-HF calculations	281
23.5	InQuanto-PySCF driver classes	282
23.6	Active space specification and AVAS	282
23.7	Energy correction with NEVPT2 and AC0	286
23.8	Hamiltonians for Embedding methods	288
23.9	DMET with PySCF fragment solvers	288
23.10	DMET with a custom solver	292
23.11	Other PySCF Hamiltonians for DMET	293
23.12	FMO with a custom solver	294
23.13	QM/MM	296

23.14 COSMO	297
24 InQuanto-NGLView	299
24.1 Visualizing Structures	299
24.2 Visualizing Fragments	300
24.3 Visualizing Orbitals	302
25 InQuanto-Phayes	303
26 Overview of InQuanto extensions examples	310
26.1 inquanto-pyscf/drivers	310
26.2 inquanto-pyscf/embeddings	310
26.3 inquanto-pyscf/fmo	310
26.4 inquanto-pyscf/projection_embedding	311
26.5 inquanto-pyscf/symmetry	311
26.6 inquanto-phayes/algorithm	311
26.7 inquanto-nglview/nglview	312
27 InQuanto API Reference	313
27.1 inquanto.algorithms	313
27.2 inquanto.ansatzes	329
27.3 inquanto.computables	462
27.4 inquanto.core	572
27.5 inquanto.embeddings	585
27.6 inquanto.express	596
27.7 inquanto.geometries	603
27.8 inquanto.mappings	626
27.9 inquanto.minimizers	644
27.10 inquanto.operators	652
27.11 inquanto.protocols	905
27.12 inquanto.spaces	1017
27.13 inquanto.states	1055
27.14 inquanto.symmetry	1099
28 InQuanto-Extensions API Reference	1104
28.1 inquanto-pyscf	1104
28.2 inquanto-nglview	1264
28.3 inquanto-phayes	1266
29 Changelog	1268
29.1 InQuanto 3.7.0	1268
29.2 InQuanto 3.6.1	1268
29.3 InQuanto 3.5.8	1269
29.4 InQuanto 3.5.7	1269
29.5 InQuanto 3.5.6	1269
29.6 InQuanto 3.5.5	1269
29.7 InQuanto 3.5.4	1269
29.8 InQuanto 3.5.3	1269
29.9 InQuanto 3.5.2	1269
29.10 InQuanto 3.5.1	1269
29.11 InQuanto 3.5.0	1270
29.12 InQuanto 3.4.2	1270
29.13 InQuanto 3.4.1	1270
29.14 InQuanto 3.4.0	1271
29.15 InQuanto 3.3.1	1271

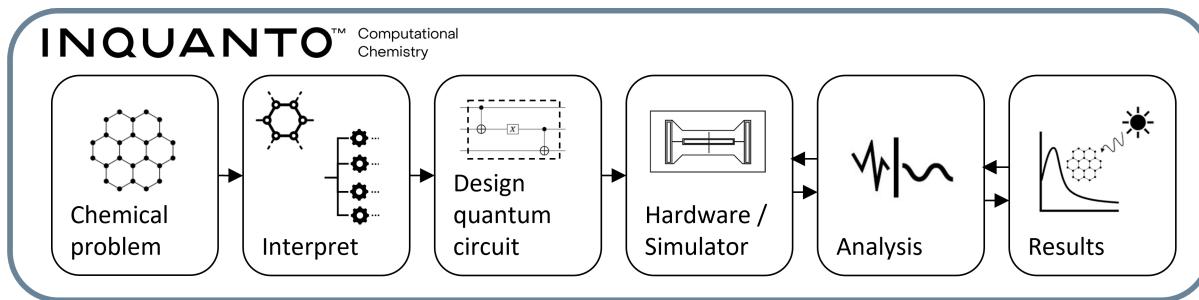
29.16 InQuanto 3.3.0	1271
29.17 InQuanto-PySCF 1.5.0	1271
29.18 InQuanto-NGLView v0.7.1	1272
29.19 InQuanto-Phayes v0.2.0	1272
29.20 InQuanto 3.2.1	1272
29.21 InQuanto 3.2.0	1272
29.22 InQuanto-PySCF 1.4.0	1273
29.23 InQuanto 3.1.2	1273
29.24 InQuanto 3.1.1	1273
29.25 InQuanto 3.1.0	1273
29.26 InQuanto 3.0.2	1273
29.27 InQuanto 3.0.1	1273
29.28 InQuanto 3.0.0	1274
29.29 2.1.1	1277
29.30 2.1.0	1277
29.31 2.0.0	1277
29.32 1.3.0	1278
29.33 1.2.2	1278
29.34 1.2.1	1278
29.35 1.2.0	1278
29.36 1.1.0	1279
29.37 1.0.5	1279
29.38 1.0.4	1279
29.39 1.0.3	1280
29.40 1.0.2	1280
29.41 1.0.1	1280
30 Bibliography	1281
31 Support	1282
32 How to cite InQuanto	1283
33 Software Licence	1284
33.1 Notices	1284
34 Open-source Attribution	1285
Bibliography	1286
Python Module Index	1291
Index	1292



This user guide 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 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 electrons and nuclei in 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 example 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](#).

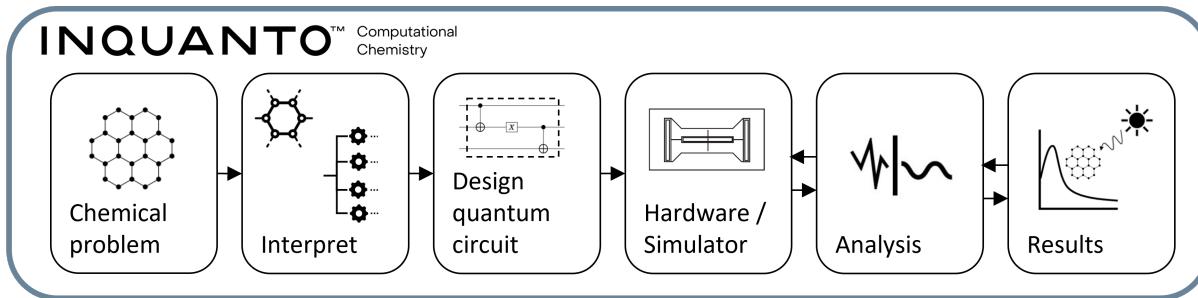


Fig. 1.1: Schematic overview of the InQuanto platform.

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.

InQuanto utilizes pytket, a python interface to TKET. 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.

INSTALLING INQUANTO

InQuanto and the associated extensions are distributed as a set of Python packages. To install InQuanto you will require access to a private Python repository as well as an **InQuanto API-KEY**. For all enquiries about accessing the package repo and licensing, please [contact the InQuanto team](#).

InQuanto can be used as part of most Python package managers (venv, Conda, etc). Please ensure you meet the [system requirements](#).

If you encounter any problems during installation, check out the [troubleshooting guide](#) before reaching to support at inquanto@quantinuum.com. Otherwise, try the [Quick-start guide](#).

2.1 System Requirements

2.1.1 inquanto

Python	3.10, 3.11, 3.12
OS	Linux, macOS>=11, WSL, Windows 10, Windows 11

2.1.2 inquanto-pyscf

Python	3.10, 3.11, 3.12
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	3.10, 3.11, 3.12
OS	Linux, macOS>=11, WSL, Windows 10, Windows 11

2.1.4 inquanto-phayes

Python	3.10, 3.11, 3.12
OS	Linux, macOS>=11, WSL, Windows 10, Windows 11
Runtime libraries	jax (see jax webpage for the inherent dependencies)

2.2 Troubleshooting

2.2.1 Licensing

InQuanto uses the Python keyring module to access the system keyring for storage and use of the license. The keyring module requires a valid backend. If using InQuanto on a managed platform, such as a high performance computing (HPC) service, keyring and its backends may be managed or password protected. We recommend checking with your system administrator about how best to store your InQuanto credentials.

No ‘keyring’ backend was found

Some Unix distributions do not have a default keyring backend. The `keyring` package InQuanto uses to interact with the system keyring has a list of alternative available backends that can be installed. 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.

A more extensive traceback can be obtained using:

```
python -c "from inquanto import activate_inquanto_license; activate_inquanto_
    _license(verbose=True)"
```

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 to install these packages on Mac at the user level is using Homebrew and removing any other installations including virtual environments (conda); this makes lapack and libomp available to any Python installations and virtual environments.

QUICK-START GUIDE

On this page we give a quick example of running a quantum computational chemistry calculation using InQuanto. This example evaluates the H₂ 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 *inquito-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 exploring InQuanto's functionality.

In the cell below, we import the *load_h5()* function which we then use to load the H₂ 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 inquito.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, ), (-1.270292724390438, 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.
-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 )
-1.1368465754747636
```

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](#). Then we prepare a 4 qubit ansatz circuit (corresponding to the 4 spin-orbitals of H₂ STO-3G) by defining the *FermionSpace* and *FermionState objects* and feeding them into the *FermionSpaceAnsatzUCCSD* class (more [here](#)). 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 2024-10-30 10:07:06.319174
```

```
# TIMER BLOCK-0 ENDS - DURATION (s): 0.3207506 [0:00:00.320751]
-1.13684658
{d0: -0.10723347230091601, s0: 0.0, s1: 0.0}
```

After the VQE algorithm has converged we can inquire the total energy of the system usng `final_value()`. In 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 singly excited terms (s) have no contribution due to being symmetry forbidden but the doubly (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 a different system with more electrons or orbitals from `express`, 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 only recommended for testing purposes, not for production purposes. 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 `SparseStatevectorProtocol`, and the Computable is `ExpectationValue`. Again, under the hood, the computables are built (using `build()`) and then ran (using `run()`). For non-state vector calculations, one must utilize a proper instance of `AlgorithmVQE`, which is much more flexible than `run_vqe()`.

HOW TO USE INQUANTO

InQuanto is a modular Python library with components that fit together to allow bespoke quantum computational chemistry calculations. Whilst InQuanto can be used in a reasonably black-box manner, the user is recommended to have some familiarity with modern computational chemistry and quantum computational chemistry.

In the figure below we show how the key components in InQuanto can be brought together. More in-depth guides on each of these components can be found in this user guide.

4.1 Chemistry Workflows

In InQuanto, `algorithms` solve quantum computational problems representing chemical systems and quantities of interest. The classes in this module, such as `AlgorithmVQE`, include quantum and hybrid quantum-classical methods which we may apply to evaluate meaningful chemical quantities, such as ground and excited states. The choice of algorithm may be determined by the problem of interest. See [Algorithms](#).

Algorithms use `computables`. Computables are expressions representing physical or chemical quantities that one may want to compute with a quantum circuit. 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 symbolic, contain the ingredients needed to give the quantity of interest, and their evaluation is performed using `protocols`. Protocols build and contain the quantum circuits required to evaluate a computable, as well as instructions for measuring and interpreting quantum measurements. See [Computables](#) and [Protocols](#) for more details.

These algorithms also 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.

 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:

$$\hat{H} = \sum_{i,j=0}^N h_{ij} a_i^\dagger a_j + \frac{1}{2} \sum_{i,j,k,l=0}^N h_{ijkl} a_i^\dagger a_k^\dagger a_l a_j \quad (4.1)$$

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]

In order to build a computable, 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*) preparing mean-field quantities, and *ii*) selecting and representing the electronic system.

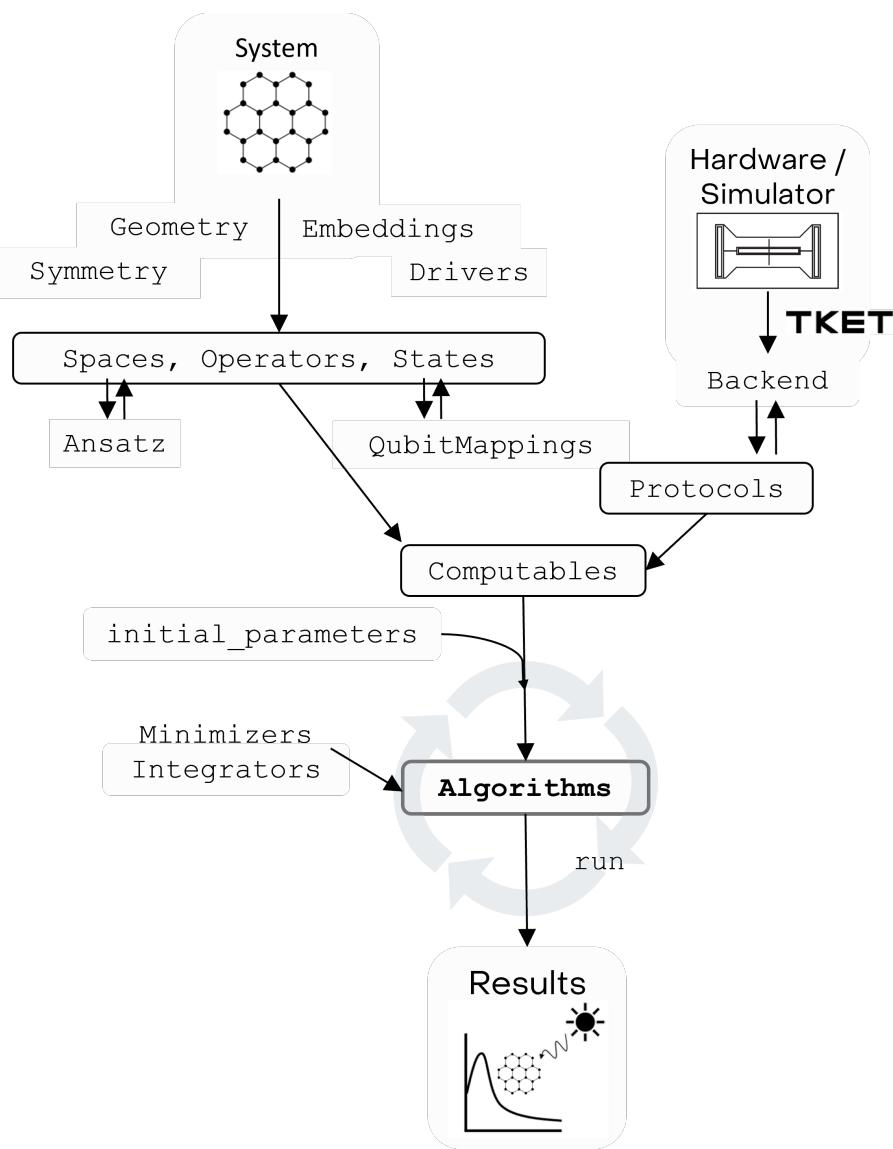


Fig. 4.1: Schematic overview of the InQuanto workflow.

Doing so will allow us to construct and run computables and algorithms.

4.2 Preparing Mean-Field Systems

One must first process the molecule/material from a specification of molecular geometry or an atomic structure file (e.g. *mol.xyz*). Then, drivers run mean-field calculations with a small classical computational overhead, such as Hartree-Fock.

Geometry 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. Drivers are used to run classical computational chemistry calculations to construct components and are generally provided by *InQuanto Extensions* but there are also model Hamiltonians and data in *Express*.

4.3 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 parameterized. 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.

4.4 Running Computables and Algorithms

When the qubit based ansatz and operators have been prepared, they are fed into *Computables* and we are ready to focus on exactly how the computable is evaluated. This is done using *Protocols*. Specifically, Protocols add instructions for how to measure the necessary state and operators for the computable of interest. They are also where noise mitigation capabilities may be provided.

Computables and Protocols 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 quantum computational hardware and simulators accessed via *pytket-extensions*. Pytket optimizes the underlying 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(s). 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.

4.5 Expert use of InQuanto

For expert quantum computational chemistry users there are a couple of useful tips.

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.

Expert users can also manipulate circuits using the `pytket` stack. Ultimately, InQuanto constructs, runs, and interprets `pytket` circuits. Thus, one can obtain the circuit objects from InQuanto using functions such as `get_circuit()` to change the underlying circuit e.g. by manually appending gates. It is also possible to inject `pytket` native objects into InQuanto, such as passing a custom built `compiler pass` objects to protocols. Due to its modular nature, these modifications can be made and become part of an InQuanto workflow.

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 behavior 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 in the following pages, 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. 5.1.

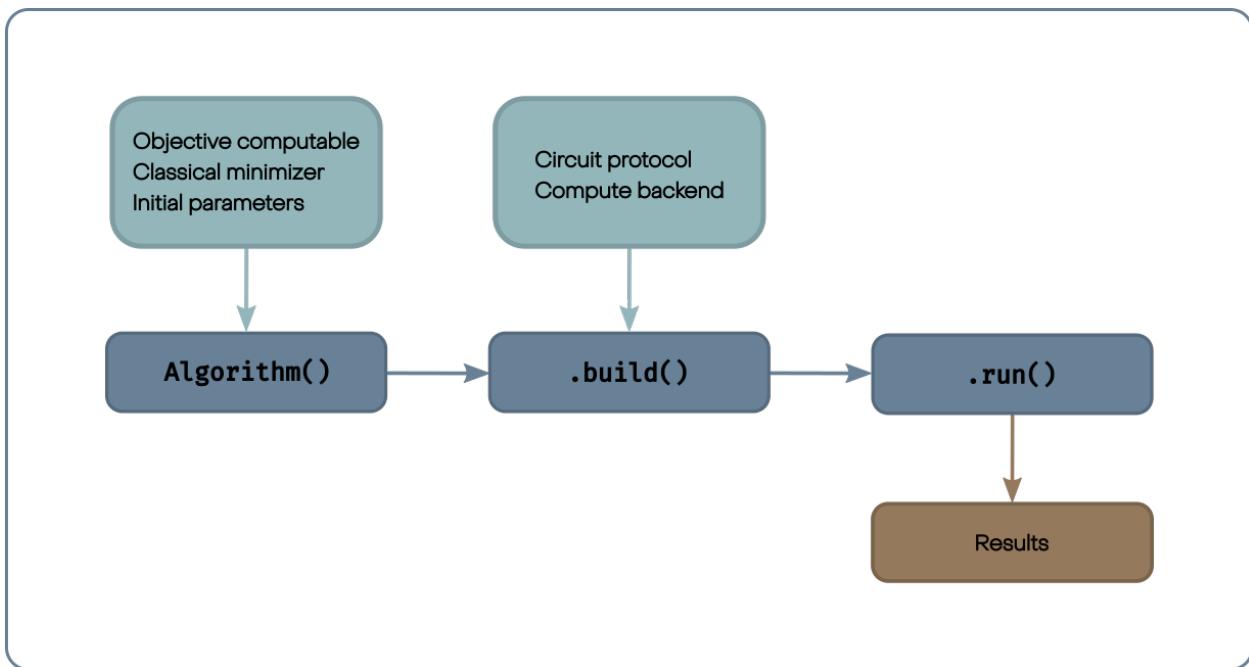


Fig. 5.1: An example of the usage of a high-level algorithm class in InQuanto. System and algorithmic details are provided to the `Algorithm` instance at construction, normally using `computable`, then compilation and simulation details are provided when calling the `.build()` method. The `.run()` method is used to perform the computation.

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 input at the `.build()` stage. 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 following pages.

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. The `build` process also will require the specification of one or more `Protocols`. These objects instruct InQuanto in how to compile the quantum circuits themselves, and require defining a pytket backend. Pytket backends are used to access different simulation strategies - whether using classical simulation, an emulator of quantum hardware, or a quantum device. 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, for example using the `generate_report()` method. Note that `generate_report()` is designed for inspection and should not be used for data processing.

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

5.1 Variational Algorithms

Variational algorithms optimize cost functions to produce meaningful quantum states. For example the ground state wave function is known to be the one with lowest total energy, and therefore for a parametric circuit, the symbol values can be tuned until the lowest possible energy is found. This is the idea behind the `Variational Quantum Eigensolver`, which is a popular method for quantum computational chemistry on current hardware due to its low circuit depth. These are hybrid algorithms which perform the parameter optimization steps on a classical computer using `minimizers`. Other variational algorithms, such as `Variational Quantum Deflation` optimize different cost functions to resolve excited states. Finally, the various flavors of `ADAPT-VQE` iteratively improve the circuit as well as the parameters values.

5.1.1 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.

$$\min E(\theta) = \langle \Psi(\vec{\theta}) | \hat{H} | \Psi(\vec{\theta}) \rangle \quad (5.1)$$

In InQuanto, the `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 can be found later in this manual.

Firstly, we generate the system of interest. Here we choose the hydrogen molecule in a minimal basis. The H_2 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 get_system
fermion_hamiltonian, fermion_fock_space, fermion_state = get_system("h2_sto3g.h5")
```

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 parameterized wave function ansatz, which will be optimized 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 of the system and its 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 `ExpectationValue` `Computable` object.

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

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

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

We can then define the initial parameters for our ansatz. Here, we use random coefficients, but these values may be user-constructed as specified in the [Ansatzes](#) section.

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

We can then finally initialize 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, `ComputableTuple`) 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. On instantiating a protocol object, one must 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 inquanto.protocols import SparseStatevectorProtocol
from pytket.extensions.qiskit import AerStateBackend
backend = AerStateBackend()
protocol_objective = SparseStatevectorProtocol(backend)
```

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(protocol_objective=protocol_objective)
```

```
<inquanto.algorithms.vqe._algorithm_vqe.AlgorithmVQE at 0x7f67283a76d0>
```

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 optimization cycle an expectation value is calculated and the parameters changed to minimize the expectation value at each step.

```
vqe.run()
```

```
# TIMER BLOCK-0 BEGINS AT 2024-10-30 10:09:09.877426
```

```
# TIMER BLOCK-0 ENDS - DURATION (s): 1.2941799 [0:00:01.294180]
```

```
<inquanto.algorithms.vqe._algorithm_vqe.AlgorithmVQE at 0x7f67283a76d0>
```

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': -1.136846575472054, 'final_parameters': array([-0.107, -0.    , 0.    ]), 'initial_parameters': [{  
    'ordering': 0, 'symbol': 'd0', 'value': 0.0}, {'ordering': 1, 'symbol': 's0', 'value': 0.0}, {'ordering': 2, 'symbol': 's1', 'value': 0.0}], 'final_parameters': [{  
    'ordering': 0, 'symbol': 'd0', 'value': -0.1072334945073921}, {'ordering': 1, 'symbol': 's0', 'value': 0.0}, {'ordering': 2, 'symbol': 's1', 'value': 0.0}]}
```

A modified initialization 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, but the same state-vector protocol object can be used for the two expressions for efficiency reasons (this is not the case for the shot-based protocols).

```
from inquanto.computables import ExpectationValueDerivative

protocol = protocol_objective

gradient_expression = ExpectationValueDerivative(ansatz, qubit_hamiltonian, ansatz.  
    ↪free_symbols_ordered())
vqe_with_gradient = (
    AlgorithmVQE(
        objective_expression=expectation_value,
        minimizer=minimizer,
        initial_parameters=initial_parameters,
        gradient_expression=gradient_expression,
    )
    .build(
        protocol_objective=protocol,
        protocol_gradient=protocol
    )
    .run()
)
```

(continues on next page)

(continued from previous page)

```

results = vqe_with_gradient.generate_report()

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 2024-10-30 10:09:11.187847
```

```

# TIMER BLOCK-1 ENDS - DURATION (s): 0.2863350 [0:00:00.286335]
Minimum Energy: -1.1368465754720527
d0: -0.10723347230091601
s0: 0.0
s1: 0.0
```

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

5.1.2 Variational Quantum Deflation AlgorithmVQD

The Variational Quantum Deflation (VQD) algorithm is a variational minimization algorithm that sequentially finds excited states by *minimizing* an objective function (shown below) which penalizes 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.

$$E(\theta_k) = \langle \Psi(\theta_k) | \hat{H} | \Psi(\theta_k) \rangle + \sum_i^{k-1} \lambda_i |\langle \Psi(\theta_k) | \Psi(\theta_i) \rangle|^2 \quad (5.2)$$

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.express import get_system
from inquanto.ansatzes import FermionSpaceAnsatzkUpCCGSD
from inquanto.computables import ExpectationValue
from inquanto.mappings import QubitMappingJordanWigner
from inquanto.minimizers import MinimizerScipy
from inquanto.algorithms import AlgorithmVQE
from pytket.extensions.qiskit import AerStateBackend
from inquanto.protocols import SparseStatevectorProtocol

fermion_hamiltonian, fermion_fock_space, fermion_state = get_system("h2_sto3g.h5")
mapping = QubitMappingJordanWigner()
qubit_hamiltonian = mapping.operator_map(fermion_hamiltonian)
ansatz = FermionSpaceAnsatzkUpCCGSD(fermion_fock_space, fermion_state, k_input=2)
expectation_value = ExpectationValue(ansatz, qubit_hamiltonian)
```

(continues on next page)

(continued from previous page)

```

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

vqe = (
    AlgorithmVQE(
        minimizer,
        expectation_value,
        initial_parameters=ansatz.state_symbols.construct_zeros(),
    )
    .build(
        protocol_objective=SparseStatevectorProtocol(AerStateBackend())
    )
    .run()
)

```

```
# TIMER BLOCK-0 BEGINS AT 2024-10-30 10:07:32.193900
```

```
# TIMER BLOCK-0 ENDS - DURATION (s): 4.2118584 [0:00:04.211858]
```

We then insist that any excited state optimized 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. (5.2). We can define the weight arbitrarily. For this example we follow the recipe in the original paper, [7] and use 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. In this case, for efficiency we use the same instance of the state-vector protocol class.

```

from inquanto.algorithms import AlgorithmVQD

protocol = SparseStatevectorProtocol(AerStateBackend())

vqd = (
    AlgorithmVQD(
        expectation_value,
        overlap_expression,
        weight_expression,
)

```

(continues on next page)

(continued from previous page)

```

        minimizer,
        ansatz_2.state_symbols.construct_random(seed=0),
        vqe._final_value,
        vqe._final_parameters,
        3,
    )
    .build(
        objective_protocol=protocol,
        overlap_protocol=protocol,
        weight_protocol=protocol,
    )
    .run()
)

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

```

```
# TIMER BLOCK-1 BEGINS AT 2024-10-30 10:07:36.468008
```

```
# TIMER BLOCK-1 ENDS - DURATION (s): 15.2261552 [0:00:15.226155]
# TIMER BLOCK-2 BEGINS AT 2024-10-30 10:07:51.694338
```

```
# TIMER BLOCK-2 ENDS - DURATION (s): 38.0713250 [0:00:38.071325]
# TIMER BLOCK-3 BEGINS AT 2024-10-30 10:08:29.765866
```

```
# TIMER BLOCK-3 ENDS - DURATION (s): 35.6532515 [0:00:35.653251]
state_energies: [-1.1368465754720403, -0.49517377025680587, -0.1358364111289325, 0.
˓→5515572309171315]
state_parameters: [SymbolDict({gd0k0: -0.05361669853564202, gd0k1: -0.
˓→05361671023590805, gs0k0: 0.0, gs0k1: 3.258370963209165e-09, gs1k0: 0.0, gs1k1: 0.0}),
˓→SymbolDict({gd0k0_2: 1.1747773670840467, gd0k1_2: -1.3965779650942605, gs0k0_2: 0.
˓→5697170315809282, gs0k1_2: -0.09511930520252851, gs1k0_2: -0.5697171505203064,
˓→gs1k1_2: -0.9683766481898197}), SymbolDict({gd0k0_2: 1.145323598261563, gd0k1_2: -1.
˓→3965782540153477, gs0k0_2: -1.0205476971781753, gs0k1_2: 0.6657162706744146, gs1k0_
˓→2: -1.0205493369719645, gs1k1_2: 0.22309277766935418}), SymbolDict({gd0k0_2: 1.
˓→1329762158576213, gd0k1_2: -1.3705336613795591, gs0k0_2: -1.193016681375957, gs0k1_
˓→2: 0.24699113243933868, gs1k0_2: -1.644291455278248, gs1k1_2: -0.4879017794522087})]
```

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

5.1.3 AlgorithmAdaptVQE and AlgorithmIQEB

AlgorithmAdaptVQE and *AlgorithmIQEB* are algorithms that construct an ansatz iteratively from a collection or pool of excitation operators. [8] [9] This differs from algorithms like *AlgorithmVQE* in which the ansatz is fixed, i.e. parameters of a fixed set of operators are optimized. At each ADAPT step, 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

$$\hat{U}_{\text{ADAPT/IQEB}}^{(N)} = \left(e^{\theta_N \hat{A}_N} \right) \left(e^{\theta_{N-1} \hat{A}_{N-1}} \right) \dots \left(e^{\theta_\lambda \hat{A}_\lambda} \right) \dots \left(e^{\theta_1 \hat{A}_1} \right) \quad (5.3)$$

$$|\Psi_{\text{ADAPT/IQEB}}^{(N)}\rangle = \hat{U}_{\text{ADAPT/IQEB}} |\text{HF}\rangle \quad (5.4)$$

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 [8], the operators \hat{A}_λ in the pool consist of all possible spin complemented anti-hermitian operators within unitary coupled cluster ansatz ([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

$$\frac{\partial E^{(n)}}{\partial \theta_\lambda} = \langle \Psi_{\text{ADAPT}}^{(n)} | [\hat{H}, \hat{A}_\lambda] | \Psi_{\text{ADAPT}}^{(n)} \rangle \quad (5.5)$$

for each excitation operator. The \hat{A}_λ 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 H₂ Hamiltonian, and qubit mapping of the [AlgorithmVQE](#) example).

```
from inquanto.algorithms import AlgorithmAdaptVQE
from inquanto.spaces import FermionSpace
from inquanto.states import QubitState, FermionState
from inquanto.mappings import QubitMappingJordanWigner
from inquanto.ansatzes import FermionSpaceAnsatzUCCSD
from inquanto.minimizers import MinimizerScipy
from inquanto.express import get_system
from inquanto.protocols import SparseStatevectorProtocol
from pytket.extensions.qiskit import AerStateBackend

fermion_hamiltonian, fermion_space, fermion_state = get_system("h2_sto3g.h5")

jw = QubitMappingJordanWigner()
qubit_hamiltonian = jw.operator_map(fermion_hamiltonian)

space = FermionSpace(4)
fermion_state = FermionState([1, 1, 0, 0])
qubit_state = QubitState([1, 1, 0, 0])
jw_map = QubitMappingJordanWigner()

pool = space.construct_single_ucc_operators(fermion_state)
pool += space.construct_double_ucc_operators(fermion_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,
    qubit_state,
    qubit_hamiltonian,
    scipy_minimizer,
    tolerance=1.0e-3
)

protocol = SparseStatevectorProtocol(AerStateBackend())

adapt.build(
    protocol,
    protocol,
    protocol
)

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-0 BEGINS AT 2024-10-30 10:07:10.453304
```

```
# TIMER BLOCK-0 ENDS - DURATION (s): 0.3099910 [0:00:00.309991]
Minimum Energy: -1.1368465754720527
d0 : -0.10723347230091601
```

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 `SparseStatevectorProtocol` 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 H₂ 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)

```

(continues on next page)

(continued from previous page)

```

fermionic_adapt = AlgorithmFermionicAdaptVQE(
    pool,
    state,
    fermion_hamiltonian,
    scipy_minimizer,
    tolerance=1.0e-3
)

protocol = SparseStatevectorProtocol(AerStateBackend())

fermionic_adapt.build(
    protocol,
    protocol,
    protocol
)

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-1 BEGINS AT 2024-10-30 10:07:10.868912
```

```
# TIMER BLOCK-1 ENDS - DURATION (s): 0.2134677 [0:00:00.213468]
Minimum Energy: -1.1368465754720527
d0 : -0.10723347230091601
```

Note that in this case the reference state and pool have not been qubit encoded before passing into [Algorithm-FermionicAdaptVQE](#). Jordan-Wigner encoding is performed by default.

In the IQEB (Iterative Qubit-Excitation Based)-VQE algorithm [9], 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 [10] commutation relations

$$\{\hat{Q}_i, \hat{Q}_i^\dagger\} = I, \quad (5.6)$$

$$[\hat{Q}_i, \hat{Q}_j^\dagger] = 0 \quad (i \neq j), \quad (5.7)$$

$$[\hat{Q}_i, \hat{Q}_j] = [\hat{Q}_i^\dagger, \hat{Q}_j^\dagger] = 0 \quad \forall i, j \quad (5.8)$$

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

$$\hat{Q}_i = \frac{1}{2} (X_i + iY_i) \quad (5.9)$$

$$\hat{Q}_i^\dagger = \frac{1}{2} (X_i - iY_i) \quad (5.10)$$

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).

$$\hat{A}_\lambda^{(ik)} = \hat{Q}_{i_\lambda}^\dagger \hat{Q}_{k_\lambda} - \hat{Q}_{k_\lambda}^\dagger \hat{Q}_{i_\lambda} = \frac{1}{2} (X_{i_\lambda} Y_{k_\lambda} - Y_{i_\lambda} X_{k_\lambda}) \quad (5.11)$$

$$\hat{A}_\lambda^{(ijkl)} = \hat{Q}_{i_\lambda}^\dagger \hat{Q}_{j_\lambda}^\dagger \hat{Q}_{k_\lambda} \hat{Q}_{l_\lambda} - \hat{Q}_{k_\lambda}^\dagger \hat{Q}_{l_\lambda}^\dagger \hat{Q}_{i_\lambda} \hat{Q}_{j_\lambda} \quad (5.12)$$

$$\begin{aligned} &= \frac{1}{8} (X_{i_\lambda} Y_{j_\lambda} X_{k_\lambda} X_{l_\lambda} + Y_{i_\lambda} X_{j_\lambda} X_{k_\lambda} X_{l_\lambda} + Y_{i_\lambda} Y_{j_\lambda} Y_{k_\lambda} X_{l_\lambda} + Y_{i_\lambda} Y_{j_\lambda} X_{k_\lambda} Y_{l_\lambda} \\ &\quad - X_{i_\lambda} X_{j_\lambda} Y_{k_\lambda} X_{l_\lambda} - X_{i_\lambda} X_{j_\lambda} X_{k_\lambda} Y_{l_\lambda} - Y_{i_\lambda} X_{j_\lambda} Y_{k_\lambda} Y_{l_\lambda} - X_{i_\lambda} Y_{j_\lambda} Y_{k_\lambda} Y_{l_\lambda}) \end{aligned} \quad (5.13)$$

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 H₂ qubit Hamiltonian and `scipy_minimizer` (see [here](#) for minimizers).

```
from inquanto.algorithms import AlgorithmIQEB
from inquanto.spaces import ParaFermionSpace

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
)

protocol = SparseStatevectorProtocol(AerStateBackend())

iqeb.build(
    protocol,
    protocol,
    protocol,
)
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-2 BEGINS AT 2024-10-30 10:07:11.193243

# TIMER BLOCK-2 ENDS - DURATION (s): 0.2487216 [0:00:00.248722]
# TIMER BLOCK-3 BEGINS AT 2024-10-30 10:07:11.442464
# TIMER BLOCK-3 ENDS - DURATION (s): 0.0167675 [0:00:00.016767]
# TIMER BLOCK-4 BEGINS AT 2024-10-30 10:07:11.459620
# TIMER BLOCK-4 ENDS - DURATION (s): 0.0160099 [0:00:00.016010]

# TIMER BLOCK-5 BEGINS AT 2024-10-30 10:07:11.673349
# TIMER BLOCK-5 ENDS - DURATION (s): 0.0704615 [0:00:00.070461]
# TIMER BLOCK-6 BEGINS AT 2024-10-30 10:07:11.744369
# TIMER BLOCK-6 ENDS - DURATION (s): 0.0618142 [0:00:00.061814]
# TIMER BLOCK-7 BEGINS AT 2024-10-30 10:07:11.806715
# TIMER BLOCK-7 ENDS - DURATION (s): 0.0522258 [0:00:00.052226]

CONVERGED!!!
Final ansatz elements after 2 iteration(s):
r_1_1      [(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: -1.1368465754720527
r_1_1 : -0.10723347230091601

```

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. Note that the internal VQE initial parameter coefficients are set to be all zeros.

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` [9]. 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.

5.2 Non-variational and Phase Estimation algorithms

Some InQuanto algorithms solve, for example, subspace or projective problems non-variationally. Two examples are the `Quantum Subspace Expansion`, and `Quantum Self Consistent Equation of Motion` methods. However, it should be noted that these methods often require accurate ground states, which may be obtained through variational approaches. Here we group these algorithms with the quantum phase estimation algorithms (although they are significantly different), which can also be considered a projection based approach.

Quantum phase estimation (QPE) [11, 12, 13, 14] is a quantum algorithm used to estimate the phase $\phi \in [0, 1]$ of a

given unitary operator U and eigenstate $|\phi\rangle$ satisfying

$$U|\phi\rangle = e^{i2\pi\phi}|\phi\rangle. \quad (5.14)$$

There are two main approaches to QPE algorithms, i.e., QPE based on the quantum Fourier transform (QFT) and QPE based on classical post-processing. The former is referred to as canonical QPE. It requires as many ancilla qubits as is necessary for representing the phase to the desired precision. The latter is referred to in several ways, including iterative QPE, stochastic QPE, and statistical QPE, depending on the method of the classical post-processing. Generally, the phase value is inferred by analyzing the samples obtained with the basic measurement operation with one ancilla qubit. In InQuanto, we refer to such a classical-post-processing-based QPE as iterative QPE for convenience, although some algorithms do not necessarily involve iterative feedback loops. InQuanto supports both canonical and iterative QPE algorithms, as explained in the following sections.

5.2.1 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 [15]

$$|\Psi_{\text{QSE}}\rangle = \sum_{i>j} c_{ij} |\psi_{ij}\rangle \quad (5.15)$$

where

$$|\psi_{ij}\rangle = a_i^\dagger a_j |\Psi_{\text{VQE}}\rangle \quad (5.16)$$

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

$$HC = SCE \quad (5.17)$$

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

$$H_{ij,kl} = \langle \psi_{ij} | \hat{H} | \psi_{kl} \rangle \quad (5.18)$$

$$S_{ij,kl} = \langle \psi_{ij} | \psi_{kl} \rangle \quad (5.19)$$

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 get_system
hamiltonian, fermion_space, fermion_state = get_system("h2_sto3g.h5")
```

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 diagonalization.

```

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 generalized 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. (5.17), using the `QSEMatricesComputable` 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 (see in `AlgorithmVQE`), so we provide the values as an array.

```

from inquanto.computables.composite import QSEMatricesComputable
from inquanto.algorithms import AlgorithmQSE

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

computable = QSEMatricesComputable(
    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, but we use a shot-based protocol. No solver is required, as it is a non-variational method.

```

from inquanto.protocols import PauliAveraging
from pytket.extensions.qiskit import AerBackend

protocol_expression = PauliAveraging(AerBackend(), shots_per_circuit=2000)

algorithm.build(
    protocol=protocol_expression
)

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

```

```
# TIMER Measure and calculate H and S matrix element BEGINS AT 2024-10-30 10:07:15.
˓→652214
```

```
# TIMER Measure and calculate H and S matrix element ENDS - DURATION (s): 1.3601326...
˓→[0:00:01.360133]
# TIMER Solve HC=CSE BEGINS AT 2024-10-30 10:07:17.012489
# TIMER Solve HC=CSE ENDS - DURATION (s): 0.0026735 [0:00:00.002674]
{'final_value': array([-5.252, -1.117, -0.134]), 'final_states': array([[ 0.168+0.
˓→001j, 0.5 -0.052j, -0.004-0.005j],
 [ 2.51 -0.053j, -0. -0.j , 0.006-0.004j],
 [-1.373-0.364j, 0.021-0.009j, 0.391+0.587j],
 [-25.254-8.7j , -0.269-0.092j, -0.087-0.057j]])}
```

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.

5.2.2 Quantum Self Consistent Equation of Motion AlgorithmSCEOM

The Quantum Self Consistent Equation of Motion (QSCEOM) is an equation-of-motion method to compute excitation energies based on a VQE ground state $|\Psi_{\text{VQE}}\rangle = U(\theta^{\text{opt}})|\Psi_{\text{HF}}\rangle$. [16] It satisfies the vacuum annihilation condition which ensures that the ground state cannot be de-excited. The excitation energies are obtained with classical diagonalization of the matrix M whose elements are evaluated with a quantum computer as:

$$M_{i,j} = \langle \Psi_{\text{VQE}} | \left[\hat{S}_i^\dagger, [\hat{H}, \hat{S}_j] \right] | \Psi_{\text{VQE}} \rangle = \langle \Psi_{\text{HF}} | \hat{G}_i^\dagger U^\dagger(\theta^{\text{opt}}) \hat{H} U(\theta^{\text{opt}}) \hat{G}_j | \Psi_{\text{HF}} \rangle - \delta_{ij} E_{\text{gr}} \quad (5.20)$$

where \hat{H} is the Hamiltonian matrix, $\hat{S} = U(\theta^{\text{opt}}) \hat{G} U^\dagger(\theta^{\text{opt}})$ are the self-consistent excitation operators, and E_{gr} is the VQE ground state energy. \hat{G} are the excitation operators that construct the SCEOM subspace.

Eq. (5.20) can be further simplified for the off-diagonal elements as:

$$\text{Re}[M_{i,j}] = M_{i+j,i+j} - \frac{M_{i,i}}{2} - \frac{M_{j,j}}{2} \quad (5.21)$$

where $M_{i+j,i+j} = \langle \Psi_{\text{HF}} | \frac{1}{\sqrt{2}} (\hat{G}_i + \hat{G}_j)^\dagger U^\dagger(\theta^{\text{opt}}) \hat{H} U(\theta^{\text{opt}}) \frac{1}{\sqrt{2}} (\hat{G}_i + \hat{G}_j) | \Psi_{\text{HF}} \rangle$

`AlgorithmSCEOM` is a high level interface of InQuanto to run the SCEOM algorithm for given excitation operators and ansatz with optimal parameters to evaluate the eigenstates. We take the minimal basis hydrogen molecule as an example to calculate the energy of single and double excited states from the VQE state with the UCCSD ansatz.

The first step is constructing the molecular Hamiltonian and the Fock state in fermion objects. For this, we pull from the `inquanto.express` suite of chemical test systems.

```
from inquanto.express import get_system
hamiltonian, fermion_space, fermion_state = get_system("h2_sto3g.h5")
```

And then map them into the qubit objects. Here we use the simplified expression of the UCCSD ansatz for the hydrogen molecule with four qubits.

```
from inquanto.mappings import QubitMappingJordanWigner
from inquanto.ansatzen import TrotterAnsatz
from inquanto.operators import QubitOperator, QubitOperatorList

jw = QubitMappingJordanWigner()
qubit_hamiltonian = jw.operator_map(hamiltonian)
```

(continues on next page)

(continued from previous page)

```

qubit_state = jw.state_map(fermion_state)
ansatz = TrotterAnsatz(
    QubitOperatorList.from_list([QubitOperator("Y0 X1 X2 X3", 1j)]), qubit_state
)

```

Subsequently, we need to define the excitation operators to be applied to our ground state. Here we use single and double excitation operators (defined as `expansion_operators` below) that conserve the particle number and the azimuthal spin (i.e. \hat{S}_z).

```

expansion_operators = fermion_space.construct_single_excitation_operators(
    fermion_state
)
expansion_operators += fermion_space.construct_double_excitation_operators(
    fermion_state
)

```

We then define a new `Computable` object for the M matrix in Eq. (5.20), using the `SCEOMMatrixComputable` class. This computable is then passed to `AlgorithmSCEOM`. In general we first need to perform a VQE calculation to obtain the optimized value of the parameter of the ansatz. In this specific example, we already know it from the previous VQE example (see in `AlgorithmVQE`), so we provide the optimized value as an array.

```

from inquanto.computables.composite import SCEOMMatrixComputable
from inquanto.algorithms import AlgorithmSCEOM

vqe_parameters = ansatz.state_symbols.construct_from_array([-0.10723347230091537])
vqe_state = ansatz.to_CircuitAnsatz(vqe_parameters)

computable = SCEOMMatrixComputable(
    space=fermion_space,
    fermion_state=fermion_state,
    ground_state=vqe_state,
    mapping=jw,
    hermitian_operator=qubit_hamiltonian,
    expansion_operators=expansion_operators,
)

algorithm = AlgorithmSCEOM(
    computable_sceom_matrix=computable
)

```

Here we perform a statevector calculation and we compare the generated energies of the excited states with the results obtained from the exact diagonalization of the hamiltonian.

```

from inquanto.protocols import SparseStatevectorProtocol
from pytket.extensions.qiskit import AerStateBackend

backend = AerStateBackend()

protocol = SparseStatevectorProtocol(backend)

algorithm.build(protocol=protocol).run()

report = algorithm.generate_report()

```

(continues on next page)

(continued from previous page)

```

print(f'QSCEOM eigenvalues: {report["final_value"]}')

# Compare with the exact diagonalization results
qubit_state = jw.state_map(fermion_state)
e_exact = qubit_hamiltonian.eigenspectrum(qubit_state.single_term.hamming_weight)

print(f'Exact eigvals : {e_exact}')

# TIMER Measure and calculate M matrix elements: BEGINS AT 2024-10-30 10:07:25.665929
# TIMER Measure and calculate M matrix elements: ENDS - DURATION (s): 0.1155100 [0:
˓→00:00.115510]
# TIMER Find eigenvalues of M matrix: BEGINS AT 2024-10-30 10:07:25.781483
# TIMER Find eigenvalues of M matrix: ENDS - DURATION (s): 0.0003635 [0:00:00.000364]
QSCEOM eigenvalues: [ 0.552 -0.136 -0.495]
Exact eigvals : [-1.137 -0.495 -0.495 -0.495 -0.136 0.552]

```

The number of states generated by the QSCEOM method depends on the excitation operators that we use, while the full spectrum of excited states is obtained with the exact diagonalization of the hamiltonian. As discussed above, in this example, QSCEOM only generates excited states that preserve the particle number and the azimuthal spin of the ground state.

The eigenvalues and eigenvectors are stored as `final_value()` and `final_states()`, respectively. Numerical instabilities in the matrix diagonalization are handled by removing near linear dependencies in the basis. We can print the bitstrings and the corresponding coefficients for each excited state that is generated by QSCEOM as shown below:

```
print(algorithm.print_sceom_states())
```

0th excited state:					
	Coefficients	Basis	states	State index	Probabilities
0	0.994256+0.000000j		0011	3	0.988545
1	0.107028+0.000000j		1100	12	0.011455

1th excited state:					
	Coefficients	Basis	states	State index	Probabilities
0	0.707107+0.000000j		0110	6	0.5
1	-0.707107+0.000000j		1001	9	0.5

2th excited state:					
	Coefficients	Basis	states	State index	Probabilities
0	-0.707107+0.000000j		0110	6	0.5
1	-0.707107+0.000000j		1001	9	0.5
None					

The excited states are sorted according to their energies.

We can also print a dataframe with information about the $\langle \hat{S}_z \rangle$, $\langle \hat{S}^2 \rangle$, and the overlap with the VQE ground state for each excited state.

```
print(algorithm.get_dataframe_sceom_analysis())
```

	$\langle S_z \rangle$	$\langle S^2 \rangle$	$\langle GS ES \rangle$
0	-0.0	0.0	-0.0

(continues on next page)

(continued from previous page)

1	-0.0	0.0	-0.0
2	-0.0	2.0	0.0

In this example, we neglected symmetry in the excited states calculation. If we want to utilize symmetries, we follow the same strategy as outlined above with the exception that the space from which we construct the set of expansion_operators should correspond to the asymmetric C_1 group so that we include the complete set of expansion_operators. In addition, we have also implemented a symmetry filter that prevents the calculation of redundant off-diagonal elements of the M matrix based on comparison of the irreducible representation of the correlated excited states i, j . This feature is optional and it can be used by passing an additional string argument pointgroup (which corresponds to the point group of the system) to [AlgorithmSCEOM](#).

Regarding shot-based measurements, please note that we need to use a [ProtocolList](#) instead of a single protocol because the correlated excited state in the [ExpectationValue](#) of Eq. (5.20) varies across the elements of the M matrix. This results to a higher number of measurements compared to QSE (see [AlgorithmQSE](#)). Please note that currently only statevector calculations are supported with [AlgorithmSCEOM](#). Shot-based calculations can still be performed with the [SCEOMMatrixComputable](#) class.

5.2.3 Canonical Quantum Phase Estimation [AlgorithmDeterministicQPE](#)

Quantum phase estimation (QPE) is a quantum algorithm used to estimate the phase $\phi \in [0, 1]$ of a given unitary operator U and eigenstate $|\phi\rangle$ satisfying

$$U|\phi\rangle = e^{i2\pi\phi}|\phi\rangle. \quad (5.22)$$

QPE based on the quantum Fourier transform (QFT) [11, 12, 13, 14] is referred to as canonical QPE. QFT can be considered to be a quantum analogue to the discrete Fourier transform, and is expressed as:

$$|j\rangle \longrightarrow \frac{1}{\sqrt{N}} \sum_k \exp\left(i \frac{2\pi j k}{N}\right) |k\rangle \quad (5.23)$$

where N is the total number of states. This general expression can be rewritten with n qubits as

$$|j_1 \cdots j_n\rangle \longrightarrow \frac{1}{\sqrt{2^n}} (|0\rangle + e^{i2\pi 0.j_n}|1\rangle) \otimes \cdots \otimes (|0\rangle + e^{i2\pi 0.j_1 j_2 \cdots j_n}|1\rangle) \quad (5.24)$$

where the binary fraction is represented as $0.j_1 \cdots j_n = \sum_{k=1}^n j_k 2^{-k}$. The basic idea of the canonical QPE is to compute the right-hand side of QFT using Eq. (5.22) and estimate the phase factor as a bit string by the inverse QFT.

To do this, canonical QPE uses two quantum registers. The first register contains n qubits initially in the state $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. The choice of n depends on the precision of the estimate of ϕ we wish to reach. The second register is initialized in the state $|\phi\rangle$, and contains as many qubits as is necessary to represent $|\phi\rangle$.

We compute the right-hand side of (5.24) using the phase kickback technique:

$$|+\rangle \otimes |\phi\rangle \xrightarrow{\text{ctrl}-U^{2^{k-1}}} \frac{1}{\sqrt{2}} (|0\rangle + e^{i2\pi 0.\phi_{n-k+1}\phi_{n-k} \cdots \phi_n}|1\rangle) \otimes |\phi\rangle \quad (5.25)$$

The example of the canonical QPE circuit is shown in [Fig. 5.2](#).

In chemistry, QPE is most often proposed within the context of calculating molecular energies. For this purpose, we set the unitary to the time evolution operator $U(t) = e^{-iHt}$, where H is the Hamiltonian describing the system. $t \in \mathbb{R}$ is a parameter. Then, the eigenstate energy is obtained as $E = -2\pi\phi/t$. The initial state is chosen to be some trial electronic wavefunction $|\Phi\rangle$, such as the Hartree-Fock state. The “quality” of the initial state is an important factor to obtain the target phase value efficiently, as the probability of obtaining ϕ is dependent on the overlap $\langle\phi|\Phi\rangle$. One can also use an ansatz that has been optimized, such as by VQE, which may lead to a reduction in the overall computational time.

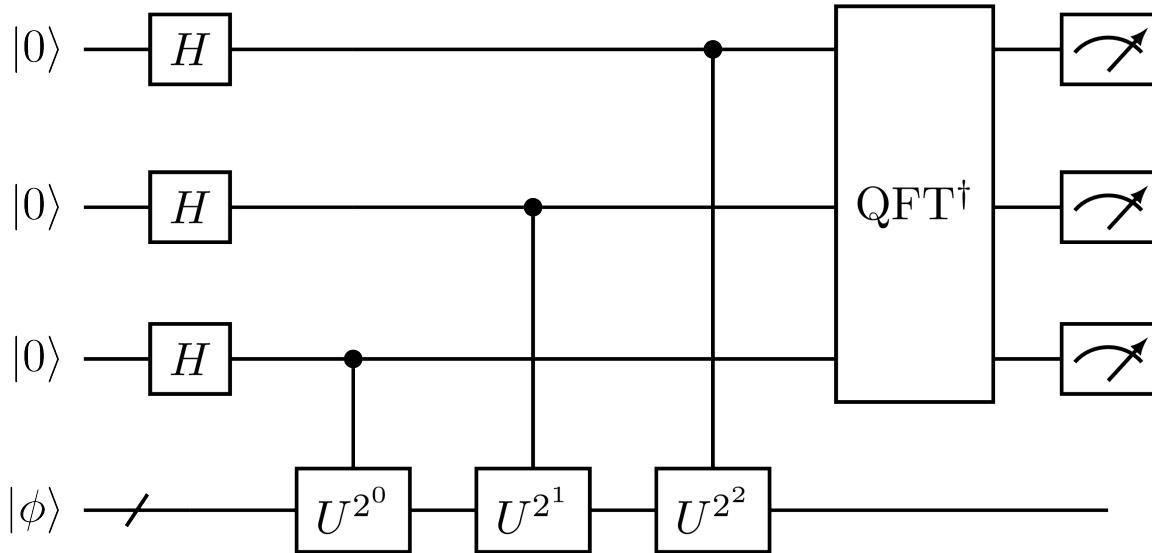


Fig. 5.2: Example of the canonical QPE circuit with three ancilla qubits.

[AlgorithmDeterministicQPE](#) may be used for performing the canonical QPE algorithm. The phase estimation circuit requires subcircuits which perform repeated sequences of the unitary evolution operator controlled upon the ancilla qubits. We refer to one of these sequences as $\text{ctrl-}U$. Here, to prepare $\text{ctrl-}U$ from the molecular Hamiltonian H , we follow the same steps as those for [AlgorithmVQE](#).

```
from inquanto.express import get_system
from inquanto.mappings import QubitMappingJordanWigner

target_data = "h2_sto3g.h5"
fermion_hamiltonian, fermion_fock_space, fermion_state = get_system(target_data)
mapping = QubitMappingJordanWigner()
qubit_hamiltonian = mapping.operator_map(fermion_hamiltonian)
```

Currently, InQuanto supports the Suzuki-Trotter decomposition to construct the $\text{ctrl}-U$ circuit (shown in the cell below). Several methods have been proposed in the literature with different asymptotic scaling, such as quantum signal processing. [17]

```
# Generate a list of qubit operators as exponents to be trotterized.
qubit_operator_list = qubit_hamiltonian.trotterize(trotter_number=1)

# The parameter 't' that is physically recognized as the time period in atomic units.
time = 1.5
```

The initial state $|\Phi\rangle$ is provided as a non-symbolic ansatz. The example below uses a modestly optimized UCCSD ansatz for our initial state preparation circuit.

```
from inquanto.ansatzes import FermionSpaceAnsatzUCCSD

# Preliminary calculated parameters.
ansatz_parameters = [-0.107, 0., 0.]
```

(continues on next page)

(continued from previous page)

```
# Generate a non-symbolic ansatz.
ansatz = FermionSpaceAnsatzUCCSD(fermion_fock_space, fermion_state, mapping)
parameters = dict(zip(ansatz.state_symbols, ansatz_parameters))
state_prep = ansatz.subs(parameters)
```

A list of qubit operators thus generated is passed to the constructor of `AlgorithmDeterministicQPE` as

```
from inquanto.algorithms import AlgorithmDeterministicQPE

algorithm = AlgorithmDeterministicQPE(
    state_prep,
    qubit_operator_list * time,
)
```

Then, we build a protocol to construct a canonical QPE circuit. `n_rounds` specifies the number of ancilla qubits of the first quantum register, which determines the precision of the computation. Together with the four qubit representation hydrogen molecule state the circuits have a total of eight qubits.

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

# Choose the backend.
backend = AerBackend()

# Set the number of rounds (ancilla qubits of the first quantum register)
n_rounds = 4

# Choose the protocol to specify how the circuit is handled.
protocol = CanonicalPhaseEstimation(
    backend=backend,
    n_rounds=n_rounds,
    n_shots=10,
)

# Build the algorithm to get it ready for experiments.
algorithm.build(
    protocol=protocol,
);
```

Now the circuit is run by `algorithm.run()` to produce the final results. The `algorithm.final_energy()` returns the energy estimate.

```
# Run the protocol.
algorithm.run()

# Display the final results.
energy = algorithm.final_energy(time=time)
print(f"energy estimate = {energy:8.4f} hartree")
```

```
energy estimate = -1.1667 hartree
```

5.2.4 Iterative Phase Estimation Algorithms

The iterative QPE algorithm, initially proposed by Kitaev [11, 18], runs a set of many QPE circuits with one ancilla qubit, each of which reads off partial information about the phase ϕ . The iterative QPE commonly uses the circuit as shown in Fig. 5.3.

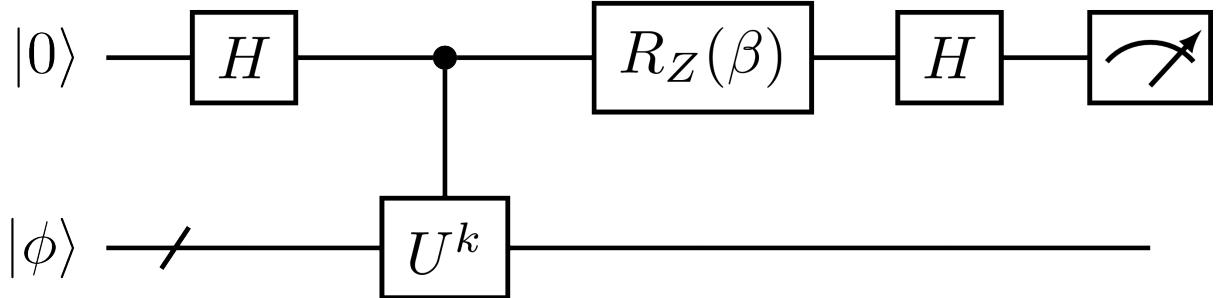


Fig. 5.3: Iterative QPE circuit parameterized by $k \in \mathbb{N}$ and $\beta \in [0, 2\pi)$.

The conditional probability of measuring m from this “basic measurement operation” [19] is expressed as

$$p(m|\phi, k, \beta) = \frac{1 + \cos(k\phi + \beta - m\pi)}{2} \quad (5.26)$$

where $m \in \{0, 1\}$. The basic idea of the iterative QPE algorithm is to perform the basic measurement operations on quantum hardware to generate samples and post-process them on classical hardware to infer the phase value. The iterative QPE algorithm needs a classical method for generating a sequence of circuit parameters (k, β) , and that for post-processing the samples of m .

As of now, InQuanto supports three iterative QPE algorithms:

- Kitaev’s QPE [AlgorithmKitaevQPE](#) [11]
- Information theory QPE [AlgorithmInfoTheoryQPE](#) [19]
- One flavor of Bayesian QPE [AlgorithmBayesianQPE](#) [20]

Kitaev’s QPE introduces the idea of replacing QFT with classical post-processing to produce a bit string of the phase value. Some algorithms [21, 22] inspired by Kitaev’s QPE become deterministic with no statistical inference under particular conditions. The information theory QPE [19] performs the maximum likelihood estimation. In this approach, one updates the distribution of the phase $P(\phi)$ based on the “evidences” (i.e., measurement samples), similar to ideas in machine learning. Information theory QPE has inspired several stochastic QPE algorithms such as Bayesian QPE [23, 24, 25], including the one implemented in the [InQuanto-Phayes](#) extension [20]. We explain the primary usage of these algorithm classes in the following subsections.

To begin the demonstration, we load in a chemical system and trotterize the Hamiltonian to obtain `qubit_operator_list`. We also prepare an ansatz object `state_prep` as the initial state $|\Phi\rangle$ using [FermionSpaceAnsatzUCCSD](#). The `time` parameter defines the duration of the time evolution. These will be used to construct the controlled unitary operation `ctrl-U`.

```

from inquanto.express import get_system
from inquanto.mappings import QubitMappingJordanWigner
from inquanto.ansatzes import FermionSpaceAnsatzUCCSD

# Generate the spin Hamiltonian from the molecular Hamiltonian.
  
```

(continues on next page)

(continued from previous page)

```

target_data = "h2_sto3g.h5"
fermion_hamiltonian, fermion_fock_space, fermion_state = get_system(target_data)
mapping = QubitMappingJordanWigner()
qubit_hamiltonian = mapping.operator_map(fermion_hamiltonian)

# Generate a list of qubit operators as exponents to be trotterized.
qubit_operator_list = qubit_hamiltonian.trotterize(trotter_number=1)

# The parameter `t` is physically recognized as the duration time in atomic units.
time = 0.25

# Preliminary calculated parameters.
ansatz_parameters = [-0.107, 0., 0.]

# Generate a non-symbolic ansatz.
ansatz = FermionSpaceAnsatzUCCSD(fermion_fock_space, fermion_state, mapping)
parameters = dict(zip(ansatz.state_symbols, ansatz_parameters))
state_prep = ansatz.subs(parameters)

```

Now we prepare the protocol that handles circuit-level operations.

```

from pytket.extensions.qiskit import AerBackend
from inquanto.protocols import IterativePhaseEstimation

# Prepare the pytket backend object.
backend = AerBackend()

# Construct the protocol to handle the iterative QPE circuits.
protocol = IterativePhaseEstimation(
    backend=backend,
    n_shots=30,
).build(
    state=state_prep,
    evolution_operator_exponents=qubit_operator_list * time,
)

```

This protocol object may be used in all the iterative QPE algorithms. Conceptually, `IterativePhaseEstimation` serves as a black box including quantum processors that takes (k, β) of the circuit shown in Fig. 5.3 as input to return the measurement outcome m .

Kitaev's QPE

The basic idea of Kitaev's QPE [11, 18] is to infer the phase bit by bit using the results of the basic measurement operations for $k = 2^{n-1}, 2^{n-2}, \dots, 2^0$, where n is the number of bits desired for representing the phase to the precision ϵ . β is chosen to be 0 and $-\frac{\pi}{2}$ to estimate the real and imaginary part of $e^{i2\pi\phi}$, respectively. This algorithm becomes similar to the Hadamard test if k is fixed to 1 and $O(1/\epsilon^2)$ measurement samples are considered. In Kitaev's QPE, the circuit depth scales as $O(1/\epsilon)$, whereas the number of shots scales $O(\log \epsilon)$.

The inference procedure starts from the least significant bit. Suppose that the phase is represented with $n + 2$ bits, i.e., $\phi = 0.\phi_1\phi_2 \cdots \phi_n\phi_{n+1}\phi_{n+2}$. We estimate $2^{n-1}\phi = 0.\phi_n\phi_{n+1}\phi_{n+2}$ similarly to the Hadamard test using the result of the basic measurement operation with $k = 2^{n-1}$. Then, ϕ_{n-1} is determined so that $0.\phi_{n-1}\phi_n\phi_{n+1}$ is closer to the phase estimate with $k = 2^{n-2}$. This inference process is repeated until we obtain the most significant bit ϕ_1 .

In the cell below, we build the `AlgorithmKitaevQPE` object. The circuit depth is controlled by `n_bits` such that the

maximum value of k is $2^{**} (n_bits - 1)$.

```
from inquanto.algorithms import AlgorithmKitaevQPE

# Parameters for the Kitaev's QPE.
n_bits = 6

# Prepare the algorithm.
algorithm = AlgorithmKitaevQPE(
    n_bits=n_bits,
).build(
    protocol=protocol,
)
```

The `build()` method takes the `IterativePhaseEstimation` object. This may be replaced with any of the other iterative QPE protocols (see [QPE protocols manual](#)). Run the algorithm to evaluate the phase and thus the energy as

```
# Run the algorithm.
algorithm.run()

# Display the final results.
mu, precision = algorithm.final_value()
energy_mu = -mu / time
energy_resolution = precision / time
print(f"Energy      = {energy_mu:10.6f} hartree")
print(f"precision = {energy_resolution:10.6f} hartree")
```

```
Energy      = -1.125000 hartree
precision =  0.015625 hartree
```

`AlgorithmKitaevQPE` returns the phase estimate and the precision to be used for computing the energy estimate. Note that the phase value is in the unit of half turns (i.e., π) following the pytket convention.

Information theory QPE

The basic idea of the information theory QPE is to estimate the phase ϕ by maximizing the likelihood of the sequence of samples of the basic measurement operation [19]. Since the measurements are independent events, the likelihood of obtaining a sequence of s measurement outcomes (m_1, m_2, \dots, m_s) for the sequence of parameters $((k_1, \beta_1), (k_2, \beta_2), \dots (k_s, \beta_s))$ is expressed as

$$P(m_1, \dots, m_s | \phi, k_1, \dots, k_s, \beta_1, \dots, \beta_s) := \prod_j^s P(m_j | \phi, k_j, \beta_j) \quad (5.27)$$

In practice, the phase ϕ is discretized as $\phi_l = \frac{l}{M}$, where $l = 0, 1, \dots, M - 1$, and perform s basic measurement operations for randomly selected $k \in [1, k_{\max}]$ and $\beta \in [0, 2\pi]$ from uniform distribution, where k_{\max} is the maximum value of k and M is the number of grid points to resolve ϕ . Then, the probability distribution is calculated with Eq. (5.27) to estimate the phase value. In the initial proposal in [19], the phase value is obtained as

$$\phi = \operatorname{argmax}_l P(m_1, \dots, m_s | \phi_l; k_1, \dots, k_s, \beta_1, \dots, \beta_s) \quad (5.28)$$

In InQuanto, it is generalized to return the mean μ and the standard deviation σ of $P(m_1, \dots, m_s | \phi_l; k_1, \dots, k_s, \beta_1, \dots, \beta_s)$.

Now we demonstrate `AlgorithmInfoTheoryQPE`. We prepare the protocol object similarly to the Kitaev's QPE case but with `n_shots=1` to perform the one-shot experiment for each pair of the circuit parameters.

```
# Construct the protocol to handle the iterative QPE circuits.
protocol = IterativePhaseEstimation(
    backend=backend,
    n_shots=1,
).build(
    state=state_prep,
    evolution_operator_exponents=qubit_operator_list * time,
)
```

Then we construct the `AlgorithmInfoTheoryQPE` with some parameters specifying the computational condition of the information theory QPE.

```
from inquanto.algorithms import AlgorithmInfoTheoryQPE

# Parameters.
n_bits = 6
resolution = 2 ** (n_bits + 4)
k_max = 2 ** (n_bits - 1)
n_samples = 50

# Set up the algorithm object.
algorithm = AlgorithmInfoTheoryQPE(
    resolution=resolution,
    k_max=k_max,
    n_samples=n_samples,
).build(
    protocol=protocol,
)
```

Run the algorithm to produce the energy estimate similarly to the case of Kitaev's QPE.

```
# Run the algorithm.
algorithm.run()

# Display the final results.
mu, sigma = algorithm.final_value()
energy_mu = -mu / time
energy_sigma = sigma / time
print(f"Energy(mu) = {energy_mu:10.6f} hartree")
print(f"Energy(sigma) = {energy_sigma:10.6f} hartree")
```

```
Energy(mu) = -1.129925 hartree
Energy(sigma) = 0.011065 hartree
```

5.3 Time evolution algorithms

InQuanto contains three classes which implement variational quantum simulation (VQS) - 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 [26] for a comprehensive review).

5.3.1 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 [26] 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 [26]. Following this reference, A is the real part of the ansatz metric tensor:

$$A = \Re\left(\frac{\partial\langle\phi(\theta(t))|}{\partial\theta_i}\frac{\partial|\phi(\theta(t))\rangle}{\partial\theta_j}\right) \quad (5.29)$$

and b is either:

$$b = \Re\left(\frac{\partial\langle\phi(\theta(t))|}{\partial\theta_i} H |\phi(\theta(t))\rangle\right) \quad (5.30)$$

for imaginary-time evolution, or:

$$b = \Im\left(\frac{\partial\langle\phi(\theta(t))|}{\partial\theta_i} H |\phi(\theta(t))\rangle\right) \quad (5.31)$$

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 $|1, 1, 0, 0\rangle$ and $|0, 0, 1, 1\rangle$ 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 ($|1, 1, 0, 0\rangle$, the Hartree-Fock ground state), and the other where both electrons are excited to the second spatial molecular orbital ($|0, 0, 1, 1\rangle$). 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
```

(continues on next page)

(continued from previous page)

```
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 inquanto.minimizers import NaiveEulerIntegrator
from inquanto.algorithms import AlgorithmMcLachlanRealTime
from inquanto.protocols import SparseStatevectorProtocol
from pytket.extensions.qiskit import AerStateBackend

time = numpy.linspace(0, 5, 501)
integrator = NaiveEulerIntegrator(
    time, disp=False, linear_solver=NaiveEulerIntegrator.linear_solver_scipy_pinvh
)
protocol = SparseStatevectorProtocol(AerStateBackend())

algodeint = AlgorithmMcLachlanRealTime(
    integrator,
    hamiltonian_operator,
    ansatz,
    initial_parameters=initial,
)
solution = algodeint.build(
    protocol=protocol,
).run();

# TIMER BLOCK-0 BEGINS AT 2024-10-30 10:10:13.252130

# TIMER BLOCK-0 ENDS - DURATION (s): 40.9487495 [0:00:40.948749]
```

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, ComputableList

evs_expression = ComputableList([
    ExpectationValue(ansatz, kernel) for kernel in [hamiltonian_operator, particle_
→number_operator, *orbital_number_operators]
])
runner = protocol.get_runner(evs_expression)

evs = algodeint.post_propagation_evaluation(runner)
evs = numpy.asarray(evs)
```

Here we plot the columns of the `evs` array to analyze results of the dynamics. We see that the electron number re-

mains 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 [26]. One can easily implement this EOM in InQuanto by creating a custom class, derived from the `ComputableNode` 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.

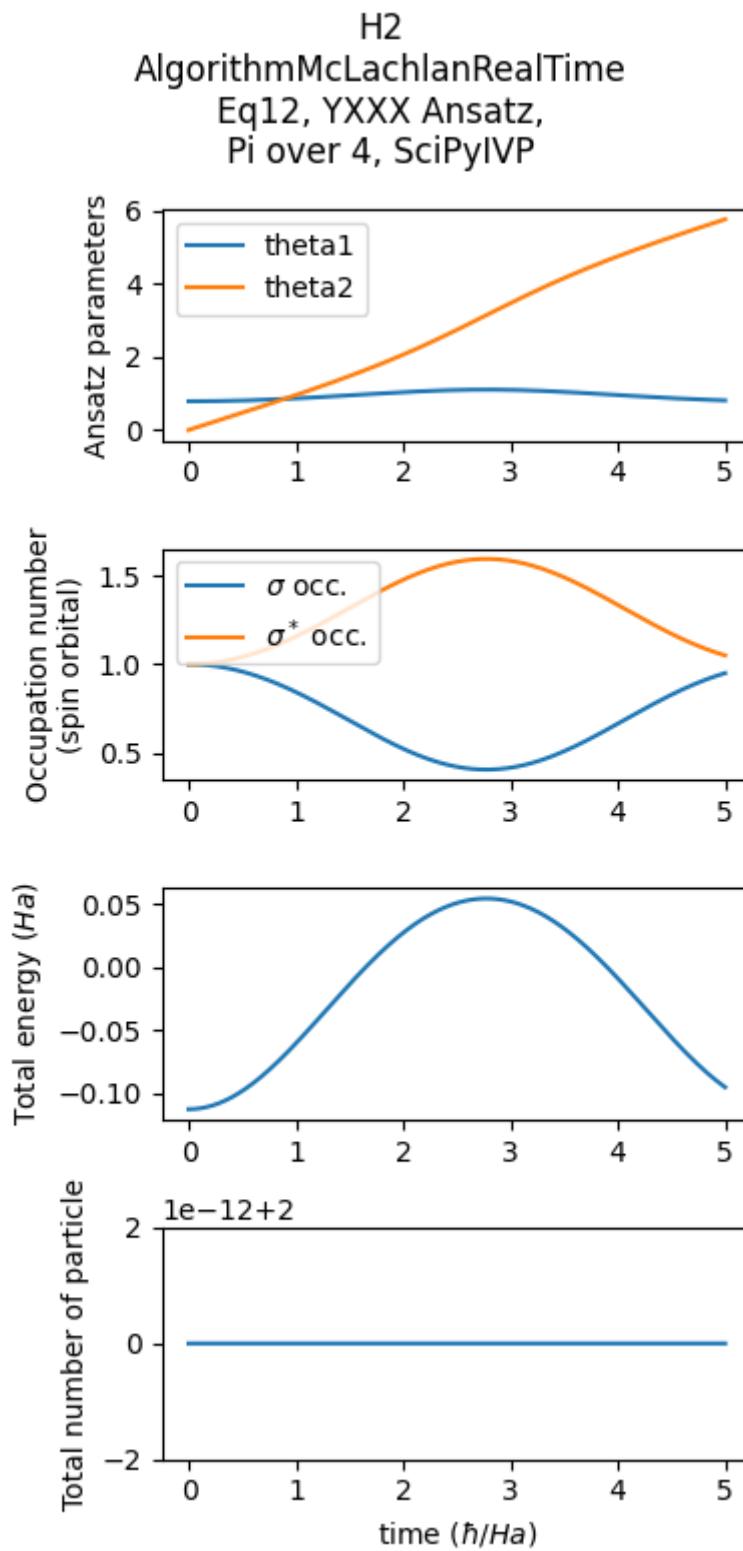


Fig. 5.4: Time evolution of various observables during a H₂ electronic wavepacket propagation.

COMPUTABLES

Computables are data structures designed for deferred evaluation, representing physical quantities that will, at some point, be measured on a quantum computer. They essentially function as placeholders, separating the construction of complex hierarchical structures from the measurement processes on quantum devices. This separation ensures that the actual evaluation only occurs once the measurement results are obtained from the quantum device.

A prime example of a computable is the `ExpectationValue`, which represents the expectation value of an operator with respect to a specific state. When an instance such as

```
exp_val_computable = ExpectationValue(state, operator)
```

is created, no immediate calculation occurs. Instead, the actual evaluation happens when the `evaluate()` method is invoked with an evaluator function:

```
exp_val = exp_val_computable.evaluate(evaluator = evaluator_function)
```

The `evaluator_function` must be capable of managing the details of quantum measurements in an independent workflow. This workflow may be custom-tailored to different simulators, hardware configurations, measurement protocols, or optimization methods. Regardless of the evaluation process, the `exp_val_computable` instance can effectively serve as a placeholder in complex expressions, for example, in Reduced Density Matrices (RDMs) or Green's functions, facilitating the construction of complex composite structures.

The underlying data structure of a general computable is a tree; each computable is a **node**, and the root node represents the final data structure to be evaluated. When the `evaluate()` method is called on the root node, the result will be calculated by calling the `evaluate()` method on the **child nodes**. This process is repeated recursively until the **leaf nodes** are reached, which will be evaluated by the `evaluator_function` passed at the root level.

We categorize computables into three main groups, introduced below:

1. Atomic Computables: These computables are leaf nodes in a computable expression tree; individual units that do not contain other computables, and are directly calculated using InQuanto `protocols`. An evaluator function must be able to handle them. A full list of atomic computables is given below:

- `ExpectationValue`
- `ExpectationValueNonHermitian`
- `Overlap`
- `OverlapReal`
- `OverlapImag`
- `OverlapSquared`
- `ExpectationValueDerivative`
- `ExpectationValueBraDerivativeReal`

- `ExpectationValueKetDerivativeReal`
- `ExpectationValueBraDerivativeImag`
- `ExpectationValueKetDerivativeImag`
- `MetricTensorReal`
- `MetricTensorImag`

2. Composite Computables: These objects describe computable quantities that may be represented in terms of smaller, atomic computables. Using these constructs can help to minimize redundant quantum measurements when computing complex chemical quantities such as Reduced Density Matrices (RDMs). Some examples of composite computables are discussed in more detail [here](#).

3. Primitive Computables: Primitive computables do not represent physical quantities themselves, but facilitate the construction of composite structures. These primitives can be parent classes, or collections such as arrays, lists, or tuples, functioning as building blocks for more complex computables. These computables are discussed in more detail [here](#).

6.1 Basic Usage and Composability

To briefly demonstrate the usage of computables, we compute the variance of a Hamiltonian given a quantum state. First, we load the necessary modules and data to acquire a Hamiltonian and generate an ansatz.

```
from inquanto.ansatzes import TrotterAnsatz
from inquanto.core import SymbolDict
from inquanto.express import load_h5
from inquanto.operators import QubitOperatorList
from inquanto.states import QubitState

# Load H2 Hamiltonian and convert to qubit basis
h2 = load_h5("h2_sto3g.h5", as_tuple=True)
qubit_hamiltonian = h2.hamiltonian_operator.qubit_encode().hermitian_part()

exponents = QubitOperatorList.from_string("theta [(1j, Y0 X1 X2 X3)]")
reference = QubitState([1, 1, 0, 0])
ansatz = TrotterAnsatz(exponents, reference)
parameters = SymbolDict(theta=-0.41)
```

We want to compute the variance $\text{Var} = \langle H^2 \rangle - \langle H \rangle^2$ where H is the Hamiltonian. We build this using the atomic computable `ExpectationValue` and the primitive `ComputableFunction`. At the construction of the variance computable, the actual value of the expectation value is not available. The `ComputableFunction` class allows us to build a new computable representing the variance.

```
from inquanto.computables import ExpectationValue
from inquanto.computables.primitive import ComputableFunction

c_variance = ComputableFunction(lambda x, y: x - y,
                                 ExpectationValue(ansatz, qubit_hamiltonian ** 2),
                                 ComputableFunction(lambda x: x ** 2, ExpectationValue(ansatz, qubit_
→hamiltonian)))
)
```

To evaluate the computable `c_variance`, an evaluator function is required. While this function typically entails quantum measurements, it is possible to define a simpler custom evaluator function for demonstration purposes. Furthermore, utilizing a custom evaluator can give you greater control over the evaluation process. Given that the variance expression

is constructed on the top of the atomic `ExpectationValue` class, we need an evaluator function capable of calculating the expectation value:

```
def my_evaluator_function(computable):
    if isinstance(computable, ExpectationValue):
        v = computable.state.get_numeric_representation(parameters)
        return computable.kernel.state_expectation(v).real
    return computable
```

The `ExpectationValue` is a simple dataclass, which stores the `state` and `kernel` operator as attributes. If `my_evaluator_function` is called on a computable which is an atomic `ExpectationValue`, it computes the expectation value via a simple statevector method. With the evaluator function we can evaluate the variance expression as follows:

```
print(c_variance.evaluate(evaluator=my_evaluator_function))
```

```
0.23089952010568593
```

During the evaluation process of `c_variance`, the `my_evaluator_function` function will be invoked on all atomic computables to obtain their evaluated values. In this example, there are two expectation values to be computed. After the values for the atomic computables have been calculated, the final value of the expression is also computed and returned by the root level `evaluate()` method.

One might notice that certain portions of this calculation are redundant; fortunately, it is possible to make a `my_evaluator_function` that can further optimize the evaluation process, for instance, by caching the results of `computable.state.get_numeric_representation(parameters)`. The advantage of using a computable expression is that the details of the optimization of the evaluation process are separated from the actual physical quantities calculated, in this case from the variance.

6.2 Evaluating Computables with Protocols

Protocols are designed to manage the lower-level details of calculations. In particular, in workflows that require quantum measurements, a protocol builds and compiles measurement circuits, post-processes measurement results, and interprets distributions. While protocols and computables are independent data structures, several protocols are provided that can help evaluate some of the atomic computables.

One of the most versatile protocols is the `SparseStatevectorProtocol`, which internally performs statevector calculations for various quantum expressions with the help of a statevector pytket backend. More details on InQuanto protocols can be found [here](#) and in the [API reference](#).

An instance of `SparseStatevectorProtocol` can provide an evaluator function via the `get_evaluator()` method. Computables may be symbolic objects, that is, they may depend on `Sympy symbols` originating from a symbolic ansatz. As a result, the `get_evaluator()` method requires a symbol-value map to substitute the numerical values in place of the symbols before the statevector computation takes place. The resulting evaluator function (`sv_evaluator` below) may then be passed to the computable's `evaluate()` method, to obtain the final results. We continue with the same example as the previous section to compute the Hamiltonian variance:

```
from pytket.extensions.qiskit import AerStateBackend
from inquanto.express import get_system
from inquanto.core import SymbolDict
from inquanto.operators import QubitOperatorList
from inquanto.ansatzes import TrotterAnsatz
from inquanto.protocols import SparseStatevectorProtocol
from inquanto.computables import ExpectationValue
```

(continues on next page)

(continued from previous page)

```

from inquanto.computables.primitive import ComputableFunction

ham, _, _ = get_system("h2_sto3g.h5")
qubit_hamiltonian = ham.qubit_encode().hermitian_part()

ansatz = TrotterAnsatz(
    exponents=QubitOperatorList.from_string("theta [(1j, Y0 X1 X2 X3)]"),
    reference=[1, 1, 0, 0],
)
parameters = SymbolDict(theta=-0.41)

c_variance = ComputableFunction(lambda x, y: x - y,
                                 ExpectationValue(ansatz, qubit_hamiltonian ** 2),
                                 ComputableFunction(lambda x: x ** 2, ExpectationValue(ansatz, qubit_
→hamiltonian)))
)

sv = SparseStatevectorProtocol(AerStateBackend())
sv_evaluator = sv.get_evaluator(parameters)
print(c_variance.evaluate(evaluator=sv_evaluator))

```

```
0.23089952010568726
```

The `SparseStatevectorProtocol` does not generate measurement circuits, but uses the backend to obtain the statevector of the ansatz, therefore the computational cost exponentially increases with the number of qubits.

In contrast, a more specialized protocol `PauliAveraging` is able to use a quantum device to calculate expectation values. This protocol builds measurement circuits, which may be submitted to a quantum device or shot-based simulator:

```

from pytket.extensions.qiskit import AerBackend
from inquanto.protocols import PauliAveraging
from pytket.partition import PauliPartitionStrat

pa = PauliAveraging(
    AerBackend(),
    shots_per_circuit=1000,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
)

pa.build_from(parameters, c_variance)
pa.run(seed=0)

pa_evaluator = pa.get_evaluator()
print(c_variance.evaluate(evaluator=pa_evaluator))

```

```
0.20430236328482176
```

It is important to note that since this protocol needs to build and run the measurement circuits, it requires the `build_from()` method, which builds the non-symbolic measurement circuits that are necessary to eventually evaluate `c_variance`, and requires the `run()` method which submits the circuits to the backend and retrieves results. The build phase is when measurement reduction takes place using the commuting set strategy. Since the protocol is built from a computable, this measurement reduction is applied to the entire computable expression tree, which is more advantageous than measuring the expectation values in the variance expression separately, in this particular example.

After the run phase, the protocol is ready to provide an evaluator with `get_evaluator()`. In this case there is no need to pass the symbol-value map to the evaluator; the build phase performs symbol substitution so that measurement circuits are already fully numerical.

Note that the `run()` method submits circuits to the backend, and waits for retrieval. As a result, it is ill-suited to submitted circuits to real quantum hardware, where there may be a long wait time. In this case, it is recommended to use the `launch()` and `retrieve()` methods:

```
handles = pa.launch(seed=0)
pa.retrieve(handles)
```

```
<inquito.protocols.averaging._pauli_averaging.PauliAveraging at 0x7f2e09f08090>
```

Together, these methods are equivalent to `run()`, but `launch()` does not block the runtime. The quantum computational details are stored in the protocol in this case, therefore all optimizations, redundancies, uncertainties and resources that are necessary to evaluate `c_variance` can be requested from the protocol for analysis.

Sometimes, details of the workflow are not important, therefore some protocols provide the `get_runner()` method which returns a function that takes in a symbol-value map and returns the value of the computable it was made for:

```
sv_runner_variance = sv.get_runner(c_variance)
print(sv_runner_variance(parameters))

pa_runner_variance = pa.get_runner(c_variance)
print(pa_runner_variance(parameters))
```

```
0.23089952010568726
0.23401851138155538
```

And it is often useful to get the result quickly for a computable, during prototyping for example. For this purpose, one can use:

```
print(c_variance.default_evaluate(parameters))
```

```
0.23089952010568726
```

which internally instantiates a `SparseStatevectorProtocol` protocol and calculates the result with it. This statevector protocol will attempt to use an `AerStateBackend` instance from `pytket-qiskit` then a `QulacsBackend` from `pytket-qulacs` to evaluate the protocol if the former is unavailable.

Note that there may be several protocols which are capable of calculating the same quantity, but by quite different methods. Also, some protocols can calculate various quantities, while others are capable of calculating only one specific quantity. It is recommended to refer to the [protocols manual page](#), and the [API documentation](#) to familiarize yourself with the capabilities and scopes of specific computables and protocols.

6.2.1 Evaluating Composite Computables with ProtocolList

This section contains some more advanced topics in the use of computables and protocols. It is recommended that the reader first familiarizes themselves with the use of `statevector` and `averaging` protocols and with `primitive` computable objects before reading this section.

`Composite` computable objects may require more than one shot-based protocol to build and run all of the circuits required for evaluation. One such example of this is a simple pair of expectation values with respect to two different states:

```
from inquanto.computables import ComputableTuple

ansatz2 = ansatz.copy().symbol_substitution("f_2")
parameters = SymbolDict(theta=-0.41, theta_2=1.0)

computable_tuple = ComputableTuple(
    ExpectationValue(ansatz, qubit_hamiltonian),
    ExpectationValue(ansatz2, qubit_hamiltonian),
)
```

The *PauliAveraging* protocol is capable of evaluating expectation values, but a single instance of any averaging protocol supports only a single ansatz state (or pair of states in the case of *overlap* and *overlap squared* protocols). Evaluating an object like this in InQuanto is made easier and more efficient using the *ProtocolList* class:

```
protocols = PauliAveraging.build_protocols_from(
    parameters=parameters,
    computable=computable_tuple,
    backend=AerBackend(),
    shots_per_circuit=10000,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
)

print(f"Number of protocols: {len(protocols)}\n")
print(protocols.dataframe_protocol_circuit())
```

Number of protocols: 2

	Protocol ID	Protocol Type	Qubits	Depth2q	Shots
0	139834317237392	PauliAveraging	4	6	10000
1	139834317237392	PauliAveraging	4	8	10000
2	139834350049552	PauliAveraging	4	6	10000
3	139834350049552	PauliAveraging	4	8	10000

The *build_protocols_from()* method parses the input computable and builds protocols for computable nodes that the chosen protocol is capable of evaluating, storing them in a *ProtocolList*. In this case, two *PauliAveraging* protocols are required, totaling four circuits.

We may then run all circuits through the *ProtocolList* interface, and generate an evaluator function for this composite computable:

```
protocols.run(seed=0)
computable_tuple.evaluate(protocols.get_evaluator())
```

(-0.9869527172517868, 0.20709664898096042)

If our composite computable contains two expectation values with respect to the *same* state, *build_protocols_from()* will collect all measurements required for both nodes into a single protocol. This allows measurement reduction (the collection of Pauli words into simultaneously measurable sets) over all Pauli words in *both* expectation values, potentially reducing the total number of circuits required. For example:

```
computable_tuple2 = ComputableTuple(
    ExpectationValue(ansatz, qubit_hamiltonian),
    ExpectationValue(ansatz, qubit_hamiltonian**2),
```

(continues on next page)

(continued from previous page)

```
)
protocols = PauliAveraging.build_protocols_from(
    parameters=parameters,
    computable=computable_tuple2,
    backend=AerBackend(),
    shots_per_circuit=10000,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
)
print(f"Number of protocols: {len(protocols)}\n")
print(protocols.dataframe_protocol_circuit())
```

Number of protocols: 1

	Protocol ID	Protocol Type	Qubits	Depth2q	Shots
0	139834272210832	PauliAveraging	4	6	10000
1	139834272210832	PauliAveraging	4	8	10000
2	139834272210832	PauliAveraging	4	8	10000

Other averaging protocols will perform similar measurement reduction when parsing composite computables with `build_protocols_from()`.

Composite computables may also consist of a mixture of different physical quantities. Again, we consider a simple example: a tuple containing two expectation values, and an overlap squared:

```
from inquanto.computables import OverlapSquared

computable_tuple_mix = ComputableTuple(
    ExpectationValue(ansatz, qubit_hamiltonian),
    ExpectationValue(ansatz2, qubit_hamiltonian),
    OverlapSquared(ansatz, ansatz2),
)
```

For this computable we will use two averaging protocols, `SwapTest` and `PauliAveraging`. We use `build_protocols_from()` again to parse the computable, and then compose a single `ProtocolList` for running and evaluating:

```
from inquanto.protocols import SwapTest

ovlp_protocols = SwapTest.build_protocols_from(
    parameters=parameters,
    computable=computable_tuple_mix,
    backend=AerBackend(),
    n_shots=10000,
)

eval_protocols = PauliAveraging.build_protocols_from(
    parameters=parameters,
    computable=computable_tuple_mix,
    backend=AerBackend(),
    shots_per_circuit=10000,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
```

(continues on next page)

(continued from previous page)

```
)
# Join together all protocols into a single ProtocolList
protocols = ovlp_protocols + eval_protocols
print(protocols.dataframe_protocol_circuit())

protocols.run(seed=0)
computable_tuple_mix.evaluate(protocols.get_evaluator())
```

	Protocol ID	Protocol Type	Qubits	Depth2q	Shots
0	139834370432080	SwapTest	9	27	10000
1	139834361712080	PauliAveraging	4	6	10000
2	139834361712080	PauliAveraging	4	8	10000
3	139835948810896	PauliAveraging	4	6	10000
4	139835948810896	PauliAveraging	4	8	10000

```
(-0.9869527172517868, 0.20709664898096042, 0.02899999999999997)
```

If we wish to evaluate part of the computable with a shot-based protocol, and another part with statevector simulation, we may use *partial evaluation*. For example, below we consider the same tuple of overlap squared and expectation values, but evaluate the overlap squared part using statevector simulation:

```
# First, build and run protocols for the expectation value nodes
protocol_list = PauliAveraging.build_protocols_from(
    parameters=parameters,
    computable=computable_tuple_mix,
    backend=AerBackend(),
    shots_per_circuit=10000,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
)
protocol_list.run(seed=1)

# Note the use of allow_partial=True for building a partial evaluator
shot_evaluator = protocol_list.get_evaluator(allow_partial=True)
partial_result = computable_tuple_mix.evaluate(shot_evaluator)
print(f"Partially evaluated computable:\n{partial_result}\n")

# Define statevector protocol to evaluate remaining node
sv_evaluator = SparseStatevectorProtocol(AerStateBackend()).get_evaluator(parameters)
final_result = partial_result.evaluate(sv_evaluator)
print(f"Fully evaluated computable:\n{final_result}")
```

```
Partially evaluated computable:
(-0.9869804954120792, 0.22476991621493542, OverlapSquared(bra_state=<inquanto.
˓→ansatzes._trotter_ansatz.TrotterAnsatz, qubits=4, gates=33, symbols=1>, ket_state=
˓→<inquanto.ansatzes._trotter_ansatz.TrotterAnsatz, qubits=4, gates=33, symbols=1>,_
˓→kernel={(): 1.0}))
```

```
Fully evaluated computable:
(-0.9869804954120792, 0.22476991621493542, 0.025633390578446377)
```

Beyond these simple cases, `ProtocolList` is useful for evaluating composite protocols in general. A natural example

of this is the overlap matrix, which consists of expectation values along the diagonal, and complex overlaps on the off-diagonal elements.

6.3 Composite Computables

Composite computables represent more complex chemical and physical quantities, which could be either structured containers or mathematical functions of atomic computables. Composite computable classes always have `Computable` appended to the end of their name. Some examples of these computables are discussed in more detail in the sections below. The reader is referred to the [API documentation](#) for a full list of composite computables.

6.3.1 Krylov subspace & Green's functions

InQuanto offers built-in support for measuring the moments of an operator to calculate quantities within a Krylov subspace, such as the Lanczos representation of a Hamiltonian or the Green's function. The key computable class for this is the `KrylovSubspaceComputable`, which is an example of a composite computable from the `inquanto.computables.composite` submodule. Given an operator and a state, this computable evaluates to a series of moments as the expectation values of the powers of the operator.

We can demonstrate this using a simple 2-site Hubbard model. First, we prepare the necessary state and operator:

```
from inquanto.computables.composite import KrylovSubspaceComputable

from inquanto.express import DriverHubbardDimer
from inquanto.operators import FermionOperator
from inquanto.ansatzes import FermionSpaceAnsatzChemicallyAwareUCCSD

driver = DriverHubbardDimer(t=0.3, u=2.15)
hamiltonian, space, state = driver.get_system()
qubit_hamiltonian = hamiltonian.qubit_encode()
ansatz = FermionSpaceAnsatzChemicallyAwareUCCSD(space, state)
parameters = ansatz.state_symbols.construct_random(2, 0.01, 0.1)
```

The computable can be instantiated as follows, where the Krylov space is expanded up to rank 4:

```
krylov_subspace_computable = KrylovSubspaceComputable(ansatz, qubit_hamiltonian, 4)
```

Once we have the computable, we can use a protocol to measure it. Since the `KrylovSubspaceComputable` calculates expectation values, we may use the `PauliAveraging` protocol:

```
from inquanto.protocols import PauliAveraging
from pytket.partition import PauliPartitionStrat
from pytket.extensions.qiskit import AerBackend

protocol = PauliAveraging(
    AerBackend(),
    shots_per_circuit=10000,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
)

protocol.build_from(parameters, krylov_subspace_computable)
protocol.run(seed=2)
```

```
<inquanto.protocols.averaging._pauli_averaging.PauliAveraging at 0x7fc1c0caf10>
```

Building the protocol from the Krylov subspace computable generates all measurement circuits, and running the protocol submits circuits to the backend and retrieves results. We may then inspect the protocol via dataframe helper methods, and use the evaluator to generate a [KrylovSubspace](#):

```
print("Measurements:")
print(protocol.dataframe_measurements())
print("Circuits measured:")
print(protocol.dataframe_circuit_shot())

krylov_subspace = krylov_subspace_computable.evaluate(evaluator=protocol.get_
→evaluator())
```

	Measurements:					
	pauli_string	mean	stderr	umean	sample_size	
0	Z0 X1 X3	-0.0970	0.009953	-0.097+/-0.010	10000	
1	Z2	0.8812	0.004728	0.881+/-0.005	10000	
2	X0 X1 Y2 Y3	-0.4608	0.008875	-0.461+/-0.009	10000	
3	X0 Z1 X2 Z3	0.1628	0.009867	0.163+/-0.010	10000	
4	Z0 X1 Z2 X3	-0.1606	0.009871	-0.161+/-0.010	10000	
5	Z0 Z1	-0.9844	0.001760	-0.9844+/-0.0018	10000	
6	Z0 Z3	0.9844	0.001760	0.9844+/-0.0018	10000	
7	Z0 Z1 Z2	-0.8786	0.004776	-0.879+/-0.005	10000	
8	Z0 Y1 Y3	-0.0970	0.009953	-0.097+/-0.010	10000	
9	Y0 Z1 Y2	-0.1114	0.009938	-0.111+/-0.010	10000	
10	X0 Y1 Y2 X3	0.4608	0.008875	0.461+/-0.009	10000	
11	Y0 Z1 Y2 Z3	0.1628	0.009867	0.163+/-0.010	10000	
12	X0 Z1 X2	-0.1114	0.009938	-0.111+/-0.010	10000	
13	Y1 Y3	0.1606	0.009871	0.161+/-0.010	10000	
14	Y0 Y2 Z3	0.1114	0.009938	0.111+/-0.010	10000	
15	Z0 Z1 Z2 Z3	1.0000	0.000000	1.0+/-0	10000	
16	X0 X2	-0.1628	0.009867	-0.163+/-0.010	10000	
17	Z0 Z2 Z3	0.8786	0.004776	0.879+/-0.005	10000	
18	Y0 Y2	-0.1628	0.009867	-0.163+/-0.010	10000	
19	Z0 Z1 Z3	0.8812	0.004728	0.881+/-0.005	10000	
20	Y0 X1 X2 Y3	0.4608	0.008875	0.461+/-0.009	10000	
21	Z0	-0.8812	0.004728	-0.881+/-0.005	10000	
22	Y1 Z2 Y3	0.0970	0.009953	0.097+/-0.010	10000	
23	Z3	-0.8786	0.004776	-0.879+/-0.005	10000	
24	Z1 Z2 Z3	-0.8812	0.004728	-0.881+/-0.005	10000	
25	Z0 Z2	-1.0000	0.000000	-1.0+/-0	10000	
26	Z1 Z2	0.9844	0.001760	0.9844+/-0.0018	10000	
27	X0 Y1 X2 Y3	-0.4764	0.008793	-0.476+/-0.009	10000	
28	Z1 Z3	-1.0000	0.000000	-1.0+/-0	10000	
29	Z0 Y1 Z2 Y3	-0.1606	0.009871	-0.161+/-0.010	10000	
30	Y0 X1 Y2 X3	-0.4764	0.008793	-0.476+/-0.009	10000	
31	Y0 Y1 Y2 Y3	-0.4764	0.008793	-0.476+/-0.009	10000	
32	X1 Z2 X3	0.0970	0.009953	0.097+/-0.010	10000	
33	Y0 Y1 X2 X3	-0.4608	0.008875	-0.461+/-0.009	10000	
34	X0 X1 X2 X3	-0.4764	0.008793	-0.476+/-0.009	10000	
35	Z2 Z3	-0.9844	0.001760	-0.9844+/-0.0018	10000	
36	X1 X3	0.1606	0.009871	0.161+/-0.010	10000	
37	X0 X2 Z3	0.1114	0.009938	0.111+/-0.010	10000	
38	Z1	0.8786	0.004776	0.879+/-0.005	10000	

(continues on next page)

(continued from previous page)

Circuits measured:

	Qubits	Depth	Depth2q	DepthCX	Shots
0	4	42	23	23	10000
1	4	43	24	24	10000
2	4	41	23	23	10000
Sum	-	-	-	-	30000

`krylov_subspace` is an instance of the `KrylovSubspace` class that offers various methods to calculate eigenvalues, Lanczos coefficients, Green's functions, and more.

```
print("Eigenvalues of the Lanczos matrix:", krylov_subspace.eigenvalues())
exact = qubit_hamiltonian.eigenspectrum(hamming_weight=state.single_term.hamming_
                                         ↴weight)
print("Exact diagonalization: ", exact)
```

```
Eigenvalues of the Lanczos matrix: [-0.156 -0.      2.15    2.306]
Exact diagonalization:  [-0.156 -0.      0.      2.15    2.306]
```

6.3.2 Reduced Density Matrices

Reduced Density Matrices (RDMs) are a use case where measurement reduction in combination with computables is very useful. Naively, we can evaluate each term in the n-body RDM by evaluating many `ExpectationValue` computables independently on each spin-traced excitation operator. However, across the entire RDM, this would result in many redundant measurements, making it advantageous to create a single, composite computable for an RDM ndarray and use a protocol to build the optimal number of circuits for the whole RDM.

To build a 1-RDM operator, we can use InQuanto's `FermionSpace`:

```
import numpy
from inquanto.spaces import FermionSpace

space = FermionSpace(4)
rdm_operators = space.construct_one_body_spatial_rdm_operators((FermionSpace.SPIN_UP, ↴
                                                               FermionSpace.SPIN_UP))
print(rdm_operators)
print(type(rdm_operators))
```

```
[{{((0, 1), (0, 0)): 1.0} {((0, 1), (2, 0)): 1.0}]
 [{{(2, 1), (0, 0)): 1.0} {((2, 1), (2, 0)): 1.0}]]
<class 'numpy.ndarray'>
```

As a result, we obtain a `numpy ndarray` of `FermionOperator` objects. To compute the expectation values, we also need a specific state. In this example, we will use the `TrotterAnsatz`:

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

ansatz = TrotterAnsatz(
    QubitOperatorList.from_list([QubitOperator("Y0 X1 X2 X3", 1j)]),
    QubitState([1, 1, 0, 0]),
)
```

(continues on next page)

(continued from previous page)

```
parameters = ansatz.state_symbols.construct_from_array([0.11])
```

With the ansatz and RDM operators, we can construct a single computable for the RDM:

```
from inquanto.computables import ExpectationValueNonHermitian
from inquanto.computables.primitive import ComputableNDArray

# We make a broadcast function over the ndarray that converts every element into
# an ExpectationValueNonHermitian computable
ndarray_broadcast = numpy.vectorize(
    lambda operator: ExpectationValueNonHermitian(ansatz, operator.qubit_encode())
)

# Then, we can convert the ndarray computables to a ComputableNDArray of computables
# to ensure that we can run and evaluate recursively.
rdm_computable = ComputableNDArray(ndarray_broadcast(rdm_operators))

print(rdm_computable)
print(type(rdm_computable))
```

```
[ [ExpectationValueNonHermitian(state=<inquanto.ansatzes._trotter_ansatz.TrotterAnsatz,
↪ qubits=4, gates=33, symbols=1>, kernel={(): 0.5, (Zq[0]): -0.5})
  ExpectationValueNonHermitian(state=<inquanto.ansatzes._trotter_ansatz.TrotterAnsatz,
↪ qubits=4, gates=33, symbols=1>, kernel={(Yq[0], Zq[1], Xq[2]): (-0-0.25j), (Yq[0],_
↪ Zq[1], Yq[2]): 0.25, (Xq[0], Zq[1], Xq[2]): 0.25, (Xq[0], Zq[1], Yq[2]): 0.25j})}
  [ExpectationValueNonHermitian(state=<inquanto.ansatzes._trotter_ansatz.TrotterAnsatz,
↪ qubits=4, gates=33, symbols=1>, kernel={(Yq[0], Zq[1], Xq[2]): 0.25j, (Xq[0],_
↪ Zq[1], Xq[2]): 0.25, (Yq[0], Zq[1], Yq[2]): 0.25, (Xq[0], Zq[1], Yq[2]): (-0-0.25j)}
↪ )
  ExpectationValueNonHermitian(state=<inquanto.ansatzes._trotter_ansatz.TrotterAnsatz,
↪ qubits=4, gates=33, symbols=1>, kernel={(): 0.5, (Zq[2]): -0.5})]
<class 'inquanto.computables.primitive.collections._ndarray.ComputableNDArray'>
```

With this RDM computable, we can use protocols to create an evaluator function. We first use `default_evaluate()` to perform a statevector calculation as a test:

```
print(rdm_computable.default_evaluate(parameters))
```

```
[[ (0.9879487246653036+0j) (3.448477954431592e-18-1.3465787677460238e-19j)
  [(3.448477954431592e-18+1.3465787677460238e-19j)
   (0.012051275334697197+0j)]]
```

Now, we use the shot-based `PauliAveraging` protocol to evaluate the RDM computable:

```
from pytket.extensions.qiskit import AerBackend
from inquanto.protocols import PauliAveraging
from pytket.partition import PauliPartitionStrat

# We can use the PauliAveraging protocol to get an evaluator
evaluator = (
    PauliAveraging(
```

(continues on next page)

(continued from previous page)

```

        AerBackend(),
        shots_per_circuit = 10000,
        pauli_partition_strategy=PauliPartitionStrat.CommutingSets
    )
    .build_from(parameters, rdm_computable)
    .run(seed=0)
    .get_evaluator()
)

rdm_measured = rdm_computable.evaluate(evaluator)

print(rdm_measured)

```

```

[[ (0.9867+0j) (0.0029+0.0041j)]
 [(0.0029-0.0041j) (0.01329999999999979+0j)]]

```

InQuanto provides built-in support for RDM computables such as the above, for example with the `RestrictedOne-BodyRDMComputable` class, which evaluates to a `RestrictedOneBodyRDM`. A suite of other density matrix *composite computables* are supported, for instance, the `SpinlessNBodyRDMArrayRealComputable` can build n-body RDMs and reduce measurement circuits further by taking symmetries into account:

```

from inquanto.ansatzes import FermionSpaceAnsatzChemicallyAwareUCCSD
from inquanto.computables.composite import SpinlessNBodyRDMArrayRealComputable
from inquanto.express import load_h5
from inquanto.mappings import QubitMappingJordanWigner

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

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 = SpinlessNBodyRDMArrayRealComputable(
    1,
    space,
    ansatz,
    QubitMappingJordanWigner(),
)

```

(continues on next page)

(continued from previous page)

```

    symmetry_operators,
)

print(rdm1)

<inquito.computables.composite.rdm._rdm_real.SpinlessNBodyRDMArrayRealComputable_
object at 0x7f758c133d90>

```

Above, we used the `express` module to load a Hamiltonian, and use a Fermion space to generate the *Z2 symmetries* of our Hamiltonian. The `SpinlessNBodyRDMArrayRealComputable` uses the symmetry information to filter for Z2 symmetry.

6.3.3 Overlap Matrices & Non-Orthogonal Subspaces

InQuanto provides the composite computable `OverlapMatrixComputable` for calculating overlap matrices:

$$S_{ij} = \langle \Psi_i | \Psi_j \rangle \quad (6.1)$$

given a list of ansatzes $\{|\Psi_0\rangle, |\Psi_1\rangle, \dots\}$. It also supports matrices of overlaps with a Hermitian kernel:

$$O_{ij} = \langle \Psi_i | \hat{O} | \Psi_j \rangle \quad (6.2)$$

where \hat{O} is represented by a `QubitOperator`. Focusing on the latter of these objects, we first show a statevector calculation with a simple kernel:

```

from inquito.ansatzes import TrotterAnsatz
from inquito.states import QubitState
from inquito.operators import QubitOperatorList, QubitOperator
from inquito.computables.composite import OverlapMatrixComputable
from inquito.core import SymbolDict

states = [
    TrotterAnsatz(
        QubitOperatorList.from_string("a [(1j, Y0 X1 X2 X3)]"),
        QubitState([1, 1, 0, 0])
    ),
    TrotterAnsatz(
        QubitOperatorList.from_string("b [(1j, Y0 Z1 Z2 Z3)]"),
        QubitState([1, 1, 0, 0])
    ),
]
kernel = QubitOperator.from_string("(1, Z0 Z1)")
params = SymbolDict(a=0.5, b=0.5)

om_computable = OverlapMatrixComputable(states, kernel)
om_result_statevector = om_computable.default_evaluate(params)
print(om_result_statevector)

[[1. +0.j 0.77+0.j]
 [0.77-0.j 0.54+0.j]]

```

This computable uses `ExpectationValue` computables for the diagonal elements of the overlap matrix, and `Overlap` computables for the off-diagonal elements. Since the kernel must be Hermitian, it is true that $O_{ij} = O_{ji}^\dagger$. The `OverlapMatrixComputable` uses this symmetry to this result to reduce the total number of calculations required.

To perform a shot-based experiment, we require two different InQuanto protocols, one for calculating the `ExpectationValue` components, and one for the `Overlap` components. In this example we choose the `PauliAveraging` and `HadamardTestOverlap` protocols for these respective tasks. See the `Protocols` manual page for more information.

Below we use the `build_protocols_from()` method to parse the overlap matrix computable and construct a `ProtocolList` object. The `ProtocolList` class groups shot-based protocols together to measure a composite computable:

```
from inquanto.protocols import PauliAveraging, HadamardTestOverlap
from pytket.extensions.qiskit import AerBackend
from pytket.partition import PauliPartitionStrat

shot_backend = AerBackend()

expval_protocols = PauliAveraging.build_protocols_from(
    parameters=params,
    computable=om_computable,
    backend=shot_backend,
    shots_per_circuit=10000,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
)

overlap_protocols = HadamardTestOverlap.build_protocols_from(
    parameters=params,
    computable=om_computable,
    backend=shot_backend,
    shots_per_circuit=10000,
    direct=True,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
)

protocol_list = expval_protocols + overlap_protocols
print(protocol_list.dataframe_protocol_circuit())
```

	Protocol ID	Protocol Type	Qubits	Depth2q	Shots
0	140143709395216	PauliAveraging	4	6	10000
1	140143645290448	PauliAveraging	4	6	10000
2	140144780873296	HadamardTestOverlap	5	87	10000
3	140144780873296	HadamardTestOverlap	5	87	10000

The `dataframe_protocol_circuit()` method shows a breakdown of the circuits required to measure the overlap matrix. Two circuits come from the `PauliAveraging` protocol for measuring the expectation values on the diagonal, and two more from the `HadamardTestOverlap` protocol for measuring the real and imaginary parts of the single, unique off-diagonal element.

Below, we run these circuits using the pytket `AerBackend`, evaluate the overlap matrix, and compare to the statevector result obtained above:

```
protocol_list.run(seed=0)
om_result_shotbased = om_computable.evaluate(protocol_list.get_evaluator())

print(f"Statevector result: \n{om_result_statevector}\n")
print(f"Shot-based result: \n{om_result_shotbased}")
```

```

Statevector result:
[[1. +0.j 0.77+0.j]
 [0.77-0.j 0.54+0.j]]

Shot-based result:
[[1. +0.j 0.772-0.01j]
 [0.772+0.01j 0.53 +0.j ]]

```

InQuanto also provides the specialized composite computable: `NonOrthogonalMatricesComputable` for building the generalized eigenvalue problem in a non-orthogonal subspace:

$$HC = SCE \quad (6.3)$$

where H is the hamiltonian matrix and S is the overlap matrix in some subspace $\{|\Psi_i\rangle\}$:

$$H_{ij} = \langle \Psi_i | \hat{H} | \Psi_j \rangle \quad (6.4)$$

$$S_{ij} = \langle \Psi_i | \Psi_j \rangle. \quad (6.5)$$

Given a set of states, the `NonOrthogonalMatricesComputable` constructs both such matrices using `OverlapMatrixComputables` discussed above. Once evaluated, we may solve the eigenvalue problem straightforwardly with the `pd_safe_eigh()` function from `inquanto.core`. The ground state energy obtained in this approach is bounded from below by the configuration interaction (CI) energy.

Below we demonstrate a simple calculation of H_2 in a minimal basis, using the two Trotter states defined above as a crude, non-orthogonal subspace:

```

from inquanto.computables.composite import NonOrthogonalMatricesComputable
from inquanto.express import load_h5
from inquanto.core import pd_safe_eigh

h2_data = load_h5("h2_sto3g.h5")
qubit_hamiltonian = h2_data["hamiltonian_operator"].qubit_encode()

no_computable = NonOrthogonalMatricesComputable(
    qubit_hamiltonian,
    states
)

# Using the same ansatz parameters as above
expval_protocols = PauliAveraging.build_protocols_from(
    parameters=params,
    computable=no_computable,
    backend=shot_backend,
    shots_per_circuit=10000,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
)

overlap_protocols = HadamardTestOverlap.build_protocols_from(
    parameters=params,
    computable=no_computable,
    backend=shot_backend,
    shots_per_circuit=10000,
    direct=True,
)

```

(continues on next page)

(continued from previous page)

```

    pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
)

no_protocols = expval_protocols + overlap_protocols
no_protocols.run(seed=0)

h, s = no_computable.evaluate(no_protocols.get_evaluator())
e, _, _ = pd_safe_eigh(h, s)

print(f"CASCI energy: {h2_data['energy_casci']}")
print(f"NO shot-based energy: {e[0]}")

```

```
CASCI energy: -1.1368465754720547
NO shot-based energy: -0.9825372067689357
```

6.4 Primitive Computables

The `inquanto.computables.primitive` submodule provides primitive computable objects which do not represent physical quantities themselves but function as building blocks for more complex structures. Moreover, primitive computables can evaluate themselves recursively without the need for an evaluator, as they are generally not the leaf nodes in a computable tree.

A basic computable, which is primarily intended for demonstration purposes, is `ComputableInt`. It stores an integer value and returns this value upon evaluation. As it does not require any quantum measurement, but returns directly the stored integer, no evaluator function needs to be passed to the `evaluate()` method.

```

from inquanto.computables.primitive import ComputableInt

cint = ComputableInt(2)
print(cint)
print(cint.evaluate())

```

```
ComputableInt(value=2)
2
```

Other primitive computables extend to more advanced data structures such as lists, arrays, tuples, and callables. These structures can house other computables, with evaluations performed recursively. The `ComputableInt` will serve as an atomic computable, demonstrating how to build these more complex data structures with it.

6.4.1 Computable Lists and Tuples

In this section, we illustrate the usage of `ComputableTuple` and `ComputableList`. These classes mirror Python's tuple and list data structures, but can also contain other computable structures. When the `evaluate()` method is called, the corresponding `evaluate()` methods of the child computables (the elements of the tuple or list that are also computables), are invoked.

```

from inquanto.computables.primitive import ComputableTuple, ComputableList

ctuple = ComputableTuple(ComputableInt(1), ComputableInt(0), -1)
print(ctuple)
print(ctuple.evaluate())

```

(continues on next page)

(continued from previous page)

```
clist = ComputableList([ComputableInt(3), ComputableInt(4), 5])
print(clist)
print(clist.evaluate())
```

```
(ComputableInt(value=1), ComputableInt(value=0), -1)
(1, 0, -1)
[ComputableInt(value=3), ComputableInt(value=4), 5]
[3, 4, 5]
```

You can explore the child computables contained within these structures using built-in methods:

```
print(list(ctuple.children()))
```

```
[ComputableInt(value=1), ComputableInt(value=0)]
```

This will print the child computables. Note that the `ctuple` has three elements, but only two children. Additionally, you can inspect the tree structure of computables using the `print_tree()` method:

```
ctuple.print_tree()
```

```
(ComputableInt(value=1), ComputableInt(value=0), -1)
  ComputableInt(value=1)
  ComputableInt(value=0)
```

6.4.2 Iterating Over Computable Trees

The computables can be further composed into larger structures.

```
co = ComputableTuple(ctuple, clist, "something else")
```

You can iterate over the nodes of a computable tree using the `walk()` method, which allows for detailed exploration of the tree structure:

```
for cnode, depth in co.walk():
    print(cnode, depth)
```

```
((ComputableInt(value=1), ComputableInt(value=0), -1), [ComputableInt(value=3),  

  ↪ ComputableInt(value=4), 5], 'something else') 0
(ComputableInt(value=1), ComputableInt(value=0), -1) 1
ComputableInt(value=1) 2
ComputableInt(value=0) 2
[ComputableInt(value=3), ComputableInt(value=4), 5] 1
ComputableInt(value=3) 2
ComputableInt(value=4) 2
```

6.4.3 Computable Multi-dimensional Arrays

The `ComputableNDArray` class allows for handling multi-dimensional arrays of computables, utilizing `numpy`'s `ndarray`. This computable `ndarray` stores computables as objects and returns an `ndarray` with evaluated values in their respective locations upon successful evaluation of all child computables. The type of values depends on the computables housed in the computable `ndarray`; hence, the returned `ndarray` will retain `object` `dtype`.

```
from inquanto.computables.primitive import ComputableNDArray

carr = ComputableNDArray([[ComputableInt(3), ComputableInt(4)], [ComputableInt(3),  
    ↪ComputableInt(4)]])
print(carr)
print(carr.evaluate())
```

```
[[ComputableInt(value=3) ComputableInt(value=4)]  
[ComputableInt(value=3) ComputableInt(value=4)]]  
[[3 4]  
[3 4]]
```

6.4.4 Calling a Function with Computables

The `ComputableFunction` class enables the conversion of any lambda function that operates on values into a function that operates on computables. For instance, if we have a list of computables and want to evaluate the sum of their values, we might proceed as follows:

```
from inquanto.computables.primitive import ComputableFunction
csum = ComputableFunction(lambda x: sum(x), clist)

print(csum)
print(csum.evaluate()) # 12
```

```
ComputableFunction(<lambda>, [ComputableInt(value=3), ComputableInt(value=4), 5])
12
```

Another example might be the division of one computable by another:

```
from inquanto.computables.primitive import ComputableFunction
cfunc = ComputableFunction(lambda x, y: x / y, csum, ComputableInt(4))
print(cfunc)
print(cfunc.evaluate())
```

```
ComputableFunction(<lambda>, ComputableFunction(<lambda>, [ComputableInt(value=3),  
    ↪ComputableInt(value=4), 5]), ComputableInt(value=4))
3.0
```

6.5 Custom Computables & Partial Evaluations

InQuanto provides the flexibility to create custom computables using `primitive` objects. In this section, we demonstrate how to build and evaluate custom computables. Consider the simple `MyComputable` class defined below:

```
from inquanto.computables.primitive import ComputableNode, ComputableTuple,  
    ↪ComputableInt

class MyComputable(ComputableNode):
    def __init__(self, value):
        self.value = value

    def evaluate(self, evaluator):
```

(continues on next page)

(continued from previous page)

```

    return evaluator(self)

c = ComputableTuple(ComputableInt(2), 3, MyComputable("value"))

print(c.evaluate(evaluator=lambda v: v.value + "_evaluated"))

```

```
(2, 3, 'value_evaluated')
```

The evaluator here is a simple lambda function. To facilitate the evaluation of custom computables, you may need to create custom evaluators. These evaluators can range from simple functions to class methods utilizing decorators for method dispatching. Furthermore, the computable type can control how the computable instance should be evaluated. For example, below we introduce `MyOtherComputable` and two different styles of evaluators that are capable of handling both custom computables:

```

class MyOtherComputable(ComputableNode):
    def __init__(self, value):
        self.value = value

    def evaluate(self, evaluator):
        return evaluator(self)

c = ComputableTuple(ComputableInt(2), 3, MyComputable("value"), MyOtherComputable(
    "other_value"))

def my_evaluator(node):
    if isinstance(node, MyComputable):
        return node.value + "_evaluated"
    elif isinstance(node, MyOtherComputable):
        return node.value + "_evaluated_differently"

print(c.evaluate(evaluator=my_evaluator))

from functools import singledispatchmethod

class MyEvaluator:
    @singledispatchmethod
    def __call__(self, node):
        raise NotImplementedError(f"{node} does not have an evaluator")

    @_call_.register(MyComputable)
    def _(self, node):
        return node.value + "_evaluated_in_class"

    @_call_.register(MyOtherComputable)
    def _(self, node):
        return node.value + "_evaluated_differently_in_class"

print(c.evaluate(evaluator=MyEvaluator()))

```

```
(2, 3, 'value_evaluated', 'other_value_evaluated_differently')
(2, 3, 'value_evaluated_in_class', 'other_value_evaluated_differently_in_class')
```

Partial evaluation of the computable tree is also allowed, provided the evaluator is designed as such. For example, the `MyPartialEvaluator` below is only capable of evaluating `MyComputable`, but not `MyOtherComputable`:

```
class MyPartialEvaluator:
    @singledispatchmethod
    def __call__(self, node):
        return node

    @_call_.register(MyComputable)
    def _(self, node):
        return node.value + "_evaluated_in_class"

print(c.evaluate(evaluator=MyPartialEvaluator()))
# But note that the return value in case of partial evaluation is still a computable

# That is
print(type(c.evaluate(evaluator=MyPartialEvaluator())))
# whereas a complete evaluation results in a bare python type
print(type(c.evaluate(evaluator=MyEvaluator())))

# Therefore, multiple evaluators can be applied in sequence, if the first one is ↵
# partial evaluation
c_partial = c.evaluate(evaluator=MyPartialEvaluator())
print(c_partial.evaluate(evaluator=MyEvaluator()))
```

```
(2, 3, 'value_evaluated_in_class', <__main__.MyOtherComputable object at
 ↪0x7fe3cd348e90>
<class 'inquanto.computables.primitive.collections._tuple.ComputableTuple'>
<class 'tuple'>
(2, 3, 'value_evaluated_in_class', 'other_value_evaluated_differently_in_class')
```

Custom evaluators and custom computables will work in harmony with other computables. For example:

```
from inquanto.computables.primitive import ComputableFunction

cf = ComputableFunction(lambda x, y: f"{x}_{y}", MyComputable("one"),
                         MyOtherComputable("two"))
print(cf.evaluate(evaluator=MyEvaluator()))
```

```
one_evaluated_in_class_two_evaluated_differently_in_class
```

On evaluation, the `evaluate()` method walks over the `ComputableFunction` expression tree and invokes `MyEvaluator` where applicable.

PROTOCOLS

Protocols represent low-level strategies through which useful quantities such as expectation values, phases, overlaps, fidelities, and derivatives can be calculated and measured using quantum circuits. Protocols can also provide evaluator functions to computables and algorithms which work based on real measurements, statevector or shot-based simulations. Moreover, they can work together with noise mitigation methods, provide resource estimation, and apply measurement optimization. While some protocols are multifunctional, others are highly specialized in computing a single quantity.

Crucially, protocols are the entry point for defining the backend device on which to run experiments. InQuanto makes use of the [pytket backend class](#), and supports the use of third-party backends through [pytket extensions](#).

Generally, protocols fall into three major categories:

1. **Statevector Protocols:** These protocols, such as [SparseStatevectorProtocol](#) or [SymbolicProtocol](#), calculate quantities that can be expressed in the form of $\langle \text{bra} | \text{operator} | \text{ket} \rangle$, where the $\langle \text{bra} |$ and $| \text{ket} \rangle$ states internally will be represented as statevectors.
2. **Averaging Protocols:** These protocols build measurement circuits for Pauli strings, typically averaging over the distributions retrieved from a quantum device or simulator. Examples include [PauliAveraging](#) and [HadamardTest](#). These protocols are specialized for one or only a few quantities. For instance, [PauliAveraging](#) calculates expectation values.
3. **Quantum Phase Estimation Protocols:** These come in various flavors, including those for the construction of canonical and iterative phase estimation variants, and the interpretation of relevant measurement results.

While some protocols share a common API, there is generally no strictly enforced universal API across all protocols, given their potentially diverse functionalities and the different quantities they calculate. In the subsections, protocols are discussed in more details.

7.1 Protocols for expectation values

Shot-based protocols for measuring expectation values use quantum circuits to evaluate expressions of the form:

$$\langle \Psi(\theta) | \hat{O} | \Psi(\theta) \rangle = \sum_i h_i \langle \Psi(\theta) | P_i | \Psi(\theta) \rangle. \quad (7.1)$$

where $|\Psi(\theta)\rangle$ is a parameterized [ansatz](#) and the operator kernel is expressed as a linear combination of Pauli strings i.e. a [QubitOperator](#):

$$\hat{O} = \sum_i h_i P_i \quad (7.2)$$

This operator might be a chemistry Hamiltonian, obtained by means of [Jordan-Wigner](#) or [Bravyi-Kitaev mapping](#) for example.

Each term in (7.1) is an expectation value of a Pauli string. Each of these terms may be measured by preparing the ansatz of choice and appending some measurement gadget. InQuanto provides two methods to measure expectation values in this

way, which are discussed below. The first is the *PauliAveraging* protocol, which directly operates on and measures the state register, and uses measurement reduction. The second is the *HadamardTest* protocol, which uses an ancilla qubit for measurement.

Note

All protocols discussed here are designed to handle a single input ansatz $|\Psi(\theta)\rangle$. To efficiently generate circuits for expectation values with a range of input states, use the *ProtocolList* class, or *build_protocols_from()* method to group supported protocols together.

7.1.1 PauliAveraging

The *PauliAveraging* protocol uses Pauli partitioning to collect operator terms into simultaneously measurable sets. Consider the example below for minimal basis H_2 with a simple ansatz:

```
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
from inquanto.computables import ExpectationValue
from inquanto.protocols import PauliAveraging
from pytket.extensions.qiskit import AerBackend
from pytket.partition import PauliPartitionStrat

h2 = load_h5("h2_sto3g.h5", as_tuple=True)
hamiltonian = h2.hamiltonian_operator.qubit_encode()
print("Number of terms in hamiltonian: ", len(hamiltonian))

theta = sympify("theta")
exponents = QubitOperatorList(QubitOperator("Y0 X1 X2 X3", 1j), theta)
reference = QubitState([1, 0, 0])
ansatz = TrotterAnsatz(exponents, reference)

energy = ExpectationValue(ansatz, hamiltonian)

protocol = PauliAveraging(
    backend=AerBackend(),
    shots_per_circuit=1000,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets
)
protocol.build_from({theta: -0.111}, energy)
protocol.run()

print("Number of circuits: ", protocol.n_circuit)

energy.evaluate(protocol.get_evaluator())
```

```
Number of terms in hamiltonian: 15
Number of circuits: 2
```

-1.1497315764351534

The full qubit-encoded hamiltonian is given by:

$$\begin{aligned}\hat{H} = & h_0 + h_1 Z_0 + h_2 Z_2 + h_3 Z_1 + h_4 Z_3 \\ & + h_5 Z_0 Z_2 + h_6 Z_1 Z_3 + h_7 Z_0 Z_1 + h_8 Z_1 Z_2 \\ & + h_9 Y_0 X_1 X_2 Y_3 + h_{10} X_0 X_1 Y_2 Y_3 + h_{11} Y_0 Y_1 X_2 X_3 \\ & + h_{12} X_0 Y_1 Y_2 X_3 + h_{13} Z_0 Z_3 + h_{14} Z_2 Z_3\end{aligned}\quad (7.3)$$

The `ExpectationValue` computable for the total energy is provided to the protocol via the `build_from()` method, along with numerical values for any ansatz parameters (`theta` above). The Hamiltonian terms are grouped into two commuting sets, corresponding to two measurement circuits:

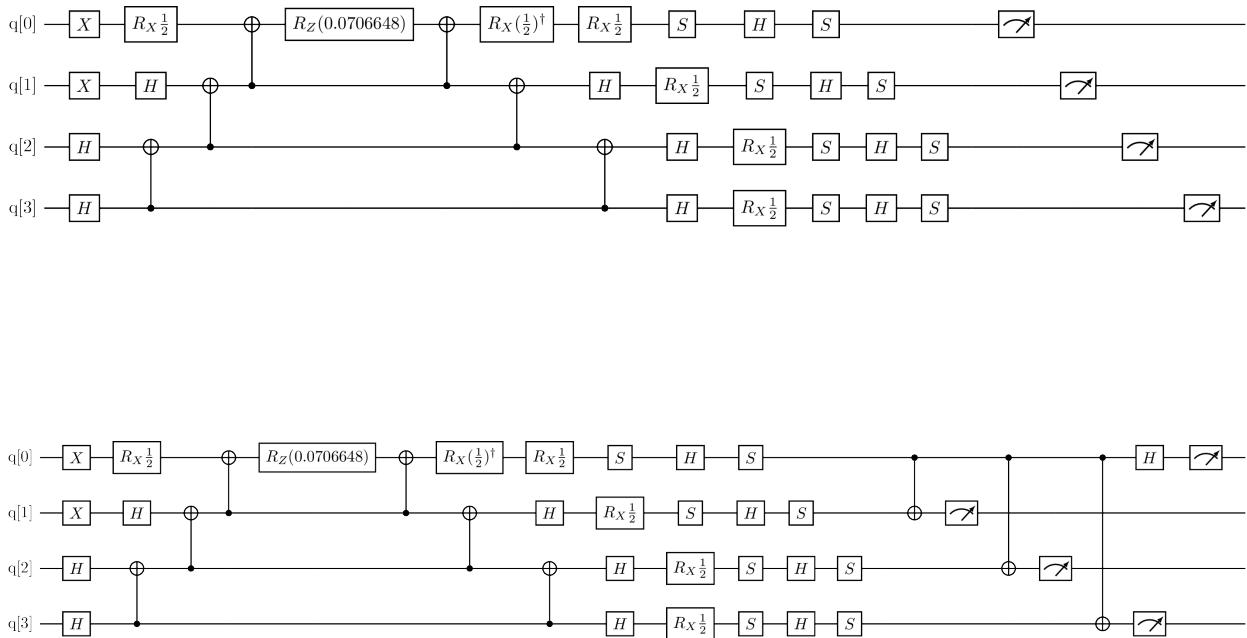


Fig. 7.1: Measurement circuits generated by the `PauliAveraging` example above.

The first circuit measures the expectation value of the individual Z terms in the Hamiltonian. The second circuit measures the remainder. The final expectation value of the operator is then the sum over the product of Pauli term weights h_i with the measured expectations $\langle P_i \rangle$.

In general, the measurement circuits generated by `PauliAveraging` take the form:

Note

Rerunning a protocol will replace the results they contain. One can perform multiple evaluations with their results and then may choose to rerun to examine, for example, a different seed or number of shots.

7.1.2 HadamardTest

The `HadamardTest` protocol constructs one measurement circuit per Pauli word P_i in the operator. Each term is measured by performing a Hadamard test, which appends the Pauli string to the ansatz, controlled on an ancilla in the $|+\rangle$ state.

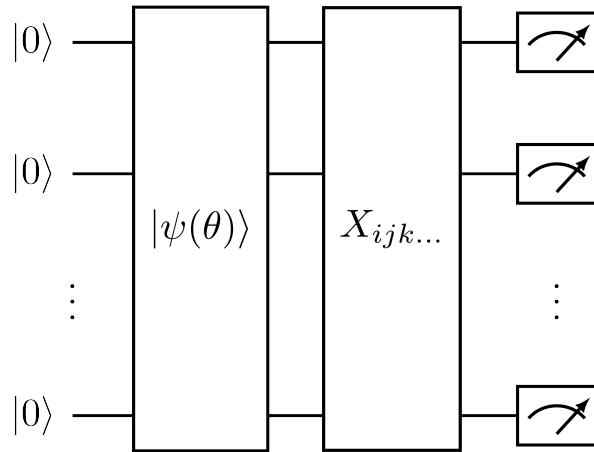


Fig. 7.2: Measurement circuit generated by PauliAveraging, where $X_{ijk\dots}$ is the set of gates required to measure a commuting set of Pauli terms in the Hamiltonian.

We construct the measurement circuits similarly to above:

```
from inquanto.protocols import HadamardTest

protocol = HadamardTest(AerBackend(), shots_per_circuit=1000)
protocol.build_from({theta: -0.111}, energy)
protocol.run()

print("Number of circuits: ", protocol.n_circuit)

energy.evaluate(protocol.get_evaluator())
```

Number of circuits: 14

-1.1334517312505805

One such measurement circuit for the Hamiltonian above is given by:

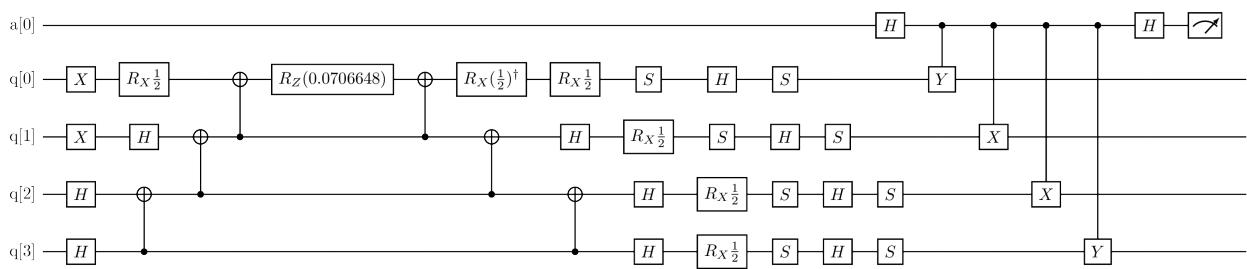


Fig. 7.3: Measurement circuit for the $Y_0X_1X_2Y_3$ term of the Hamiltonian, generated by the HadamardTest example above.

The expectation value of the corresponding Pauli string is determined by measuring the state of the ancilla qubit:

$$\langle P_i \rangle = p(0) - p(1) \quad (7.4)$$

where $p(b)$ is the probability of measuring the ancilla qubit in state $b \in 0, 1$. The expectation value of the full operator is then the sum over the product of weighted Pauli strings with the measured expectations.

In general, the measurement circuits generated by `HadamardTest` take the form:

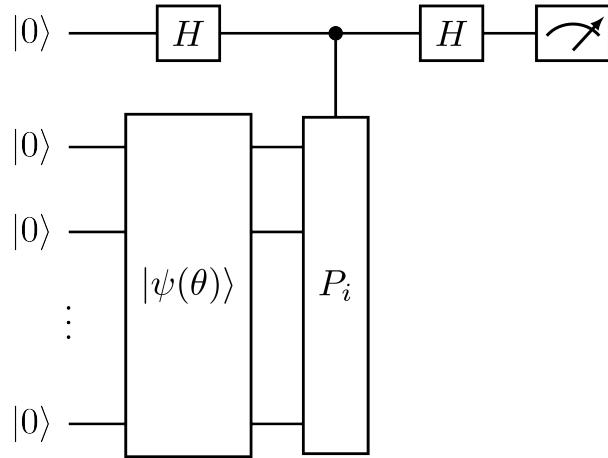


Fig. 7.4: Measurement circuit generated by `HadamardTest`.

7.2 Protocols for overlap squared

The `OverlapSquared`, or fidelity, of two quantum states $|\Psi_0\rangle$ and $|\Psi_1\rangle$ is given by:

$$|\langle\Psi_0|\Psi_1\rangle|^2 \quad (7.5)$$

Shot-based calculations of the overlap squared are supported by three InQuanto protocols: `ComputeUncompute`, `SwapTest`, and `DestructiveSwapTest`. In all cases one may calculate the bare fidelity, as above, or with a kernel consisting of a single Pauli string:

$$|\langle\Psi_0|hP|\Psi_1\rangle|^2 \quad (7.6)$$

where h is a scalar factor. All of these protocols use a single quantum circuit to perform the measurement.

7.2.1 ComputeUncompute

Let $|\Psi_0\rangle = U_0 |\bar{0}\rangle$ and $|\Psi_1\rangle = U_1 |\bar{0}\rangle$, where U_0 and U_1 are state preparation unitaries. The fidelity is hence given by $|\langle\bar{0}|U_0^\dagger U_1 |\bar{0}\rangle|^2$. The `ComputeUncompute` protocol prepares the state $U_0^\dagger U_1 |\bar{0}\rangle$ directly by concatenating the state preparation unitaries (“computing” the $|\Psi_1\rangle$ state and “uncomputing” $|\Psi_0\rangle$) [27], generating a circuit of the form:

The full register is measured in the computational basis, and the overlap squared is given by the probability of measuring all qubits in the zero state, $|\bar{0}\rangle$. See a simple example below:

```
from sympy import sympify
from inquanto.operators import QubitOperator, QubitOperatorList
from inquanto.states import QubitState
from inquanto.ansatzes import TrotterAnsatz
from inquanto.computables import OverlapSquared
from inquanto.protocols import ComputeUncompute
from pytket.extensions.qiskit import AerBackend
```

(continues on next page)

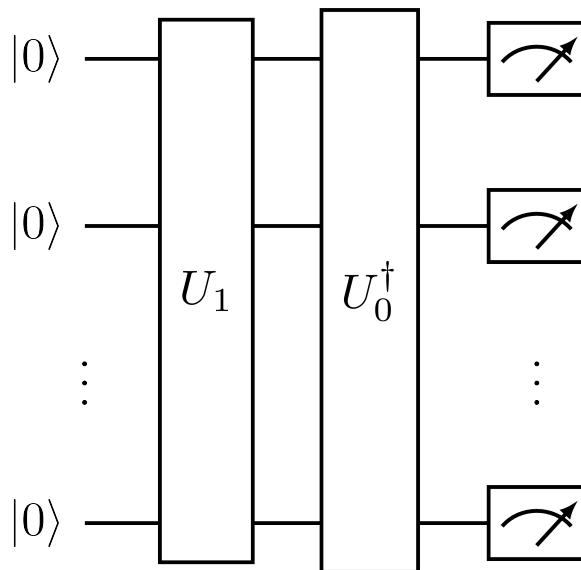


Fig. 7.5: Measurement circuit generated by `ComputeUncompute`.

(continued from previous page)

```
reference = QubitState([1, 1, 0, 0])
theta0, theta1 = sympify("theta0"), sympify("theta1")

ansatz0 = TrotterAnsatz(
    QubitOperatorList(QubitOperator("Y0 X1 X2 X3", 1j), theta0),
    reference
)

ansatz1 = TrotterAnsatz(
    QubitOperatorList(QubitOperator("X0 Z1 Y2 Z3", 1j), theta1),
    reference
)

fidelity = OverlapSquared(ansatz0, ansatz1)

protocol = ComputeUncompute(AerBackend(), n_shots=1000)
protocol.build_from({theta0: -0.111, theta1: 2.5}, fidelity)
protocol.run(seed=0)

circuit = protocol.get_circuits()[0]
print("Circuit depth: ", circuit.depth())
print("Num qubits: ", circuit.n_qubits)

fidelity.evaluate(protocol.get_evaluator())
```

```
Circuit depth: 19
Num qubits: 4
```

```
0.634
```

7.2.2 SwapTest

The `SwapTest` protocol implements the canonical swap test for evaluating the overlap squared [28]. This procedure prepares both states in parallel registers and performs a SWAP operation between them, controlled on an ancilla qubit in the $|+\rangle$ state. In general, this circuit takes the form:

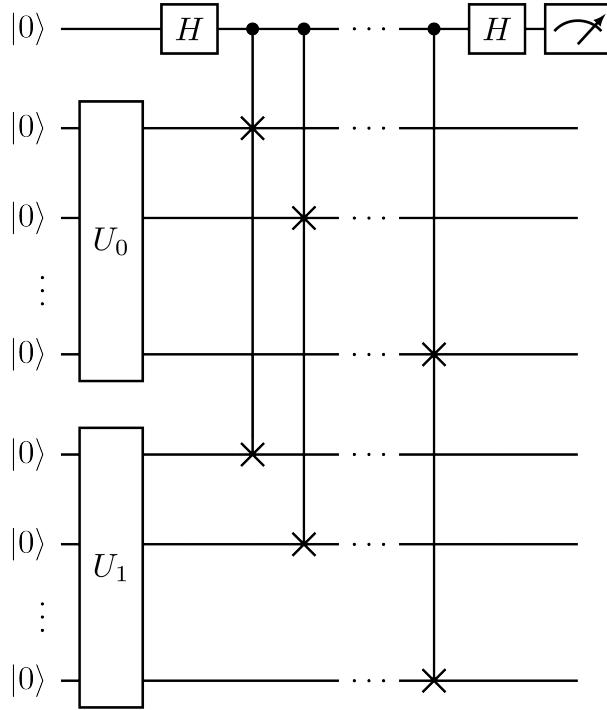


Fig. 7.6: Measurement circuit generated by `SwapTest`.

The probability of measuring the ancilla qubit in the $|0\rangle$ state is then given by $p(0) = \frac{1}{2}(1 + |\langle\Psi_0|\Psi_1\rangle|^2)$, from which the overlap squared is extracted. Similarly to above, this protocol is used as follows:

```
from inquanto.protocols import SwapTest

protocol = SwapTest(AerBackend(), n_shots=1000)
protocol.build_from({theta0: -0.111, theta1: 2.5}, fidelity)
protocol.run(seed=0)

circuit = protocol.get_circuits()[0]
print("Circuit depth: ", circuit.depth())
print("Num qubits: ", circuit.n_qubits)

fidelity.evaluate(protocol.get_evaluator())
```

```
Circuit depth: 45
Num qubits: 9
```

```
0.6499999999999999
```

7.2.3 DestructiveSwapTest

Similarly to the *SwapTest*, the *DestructiveSwapTest* prepares both states in parallel registers, but does not require an ancilla qubit. This method performs an effective XOR operation between the states by appending a ladder of CNOT gates between the i-th qubits of each register, and a layer of Hadamard gates on the control register [29]. In general, this circuit takes the form:

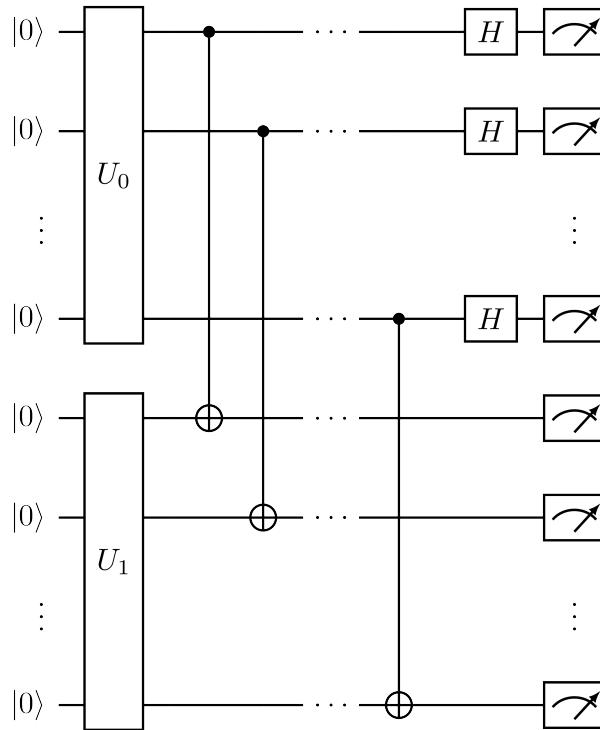


Fig. 7.7: Measurement circuit generated by `DestructiveSwapTest`.

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 register. A particular shot is deemed successful if the bitwise AND operation between the two state registers has even parity. The probability of success is given by $p_s = \frac{1}{2}(1 + |\langle \Psi_0 | \Psi_1 \rangle|^2)$, from which the overlap squared is computed. Similarly to above, this protocol is used as follows:

```
from inquanto.protocols import DestructiveSwapTest

protocol = DestructiveSwapTest(AerBackend(), n_shots=1000)
protocol.build_from({theta0: -0.111, theta1: 2.5}, fidelity)
protocol.run(seed=0)

circuit = protocol.get_circuits()[0]
print("Circuit depth: ", circuit.depth())
print("Num qubits: ", circuit.n_qubits)

fidelity.evaluate(protocol.get_evaluator())
```

```
Circuit depth: 12
Num qubits: 8
```

```
0.6439999999999999
```

7.3 Protocols for overlaps

The *Overlap* of the two quantum states $|\Psi_0\rangle$ and $|\Psi_1\rangle$ with a kernel \hat{A} is, in general, a complex number given by

$$\langle \Psi_0 | \hat{A} | \Psi_1 \rangle \quad (7.7)$$

where \hat{A} is a *QubitOperator* i.e. it is written as a linear combination of Pauli strings $\hat{A} = \sum_i c_i P_i$, and may be an identity. InQuanto supports shot-based calculations of overlaps with the *HadamardTestOverlap*, *SwapFactorizedOverlap*, and *ComputeUncomputeFactorizedOverlap* protocols, discussed below.

7.3.1 HadamardTestOverlap

Let $|\Psi_0\rangle = U_0 |\bar{0}\rangle$ and $|\Psi_1\rangle = U_1 |\bar{0}\rangle$, where U_0 and U_1 are state preparation unitaries. The *HadamardTestOverlap* protocol uses a “linear combination of unitaries” approach with a single ancilla on which to control the state preparation unitaries [30]. This protocol offers two measurement options: `direct=False`, where measurement is performed on the ancilla qubit only, and `direct=True`, where measurement is performed on both the ancilla and state registers [31]. First, we discuss the `direct=False` case.

To calculate the real part of the overlap, $\text{Re}\langle \Psi_0 | \Psi_1 \rangle$, a single circuit is required which takes the form:

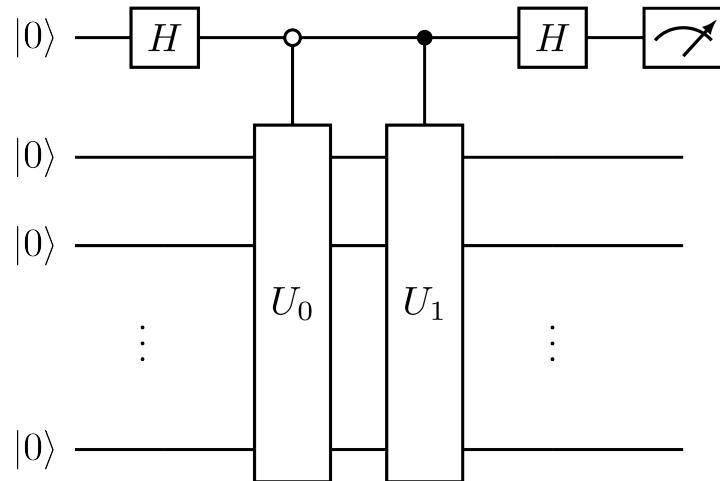


Fig. 7.8: Measurement circuit for the real part of the overlap with identity kernel, generated by the *HadamardTestOverlap* protocol with `direct=False`. The first qubit is the ancilla and the remaining qubits comprise the state register.

This circuit prepares the quantum state:

$$\frac{1}{2} [|0\rangle \otimes |\Psi_0 + \Psi_1\rangle + |1\rangle \otimes |\Psi_0 - \Psi_1\rangle] \quad (7.8)$$

where the first ket in each term is the ancilla, and $|\Psi_0 \pm \Psi_1\rangle = (U_0 \pm U_1)|\bar{0}\rangle$ is the state register. Given this state, the real part of the overlap is given by $\text{Re}\langle \Psi_0 | \Psi_1 \rangle = p(0) - p(1)$, where $p(b)$ is the probability of measuring the ancilla qubit in the state b . To compute the imaginary part of the overlap, a similar circuit is required with a small modification compared to the circuit above:

and the imaginary part is given equivalently by $\text{Im}\langle \Psi_0 | \Psi_1 \rangle = p(0) - p(1)$.

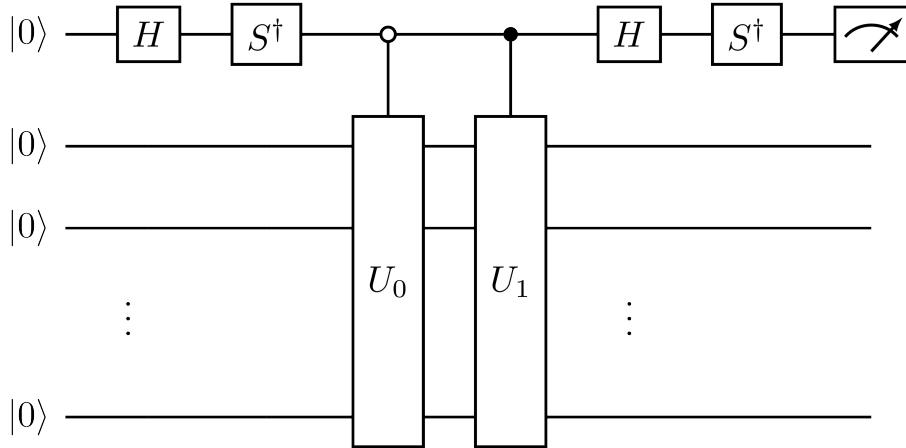


Fig. 7.9: Measurement circuit for the imaginary part of the overlap with identity kernel, generated by the `HadamardTestOverlap` protocol with `direct=False`.

For an overlap with a kernel $\hat{A} = \sum_i c_i P_i$, we may write:

$$\langle \Psi_0 | \hat{A} | \Psi_1 \rangle = \sum_i c_i \langle \Psi_0 | P_i | \Psi_1 \rangle = \sum_i c_i \langle \Psi_0 | \Psi_1^i \rangle \quad (7.9)$$

where each Pauli word has been appended to the $|\Psi_1\rangle$ state preparation; $|\Psi_1^i\rangle = P_i U_1 |\bar{0}\rangle$. Each term in this sum is then computed independently as described above. Thus, with `direct=False`, to compute the complex overlap with a kernel of N terms, $2N$ circuits are required.

In the `direct=True` case, the Pauli words in \hat{A} are partitioned into simultaneously measurable sets (commuting sets, for example). Measurement circuits for each set are then appended to the end of the state register, similarly to the [Pauli averaging protocol](#). These circuits take the form:

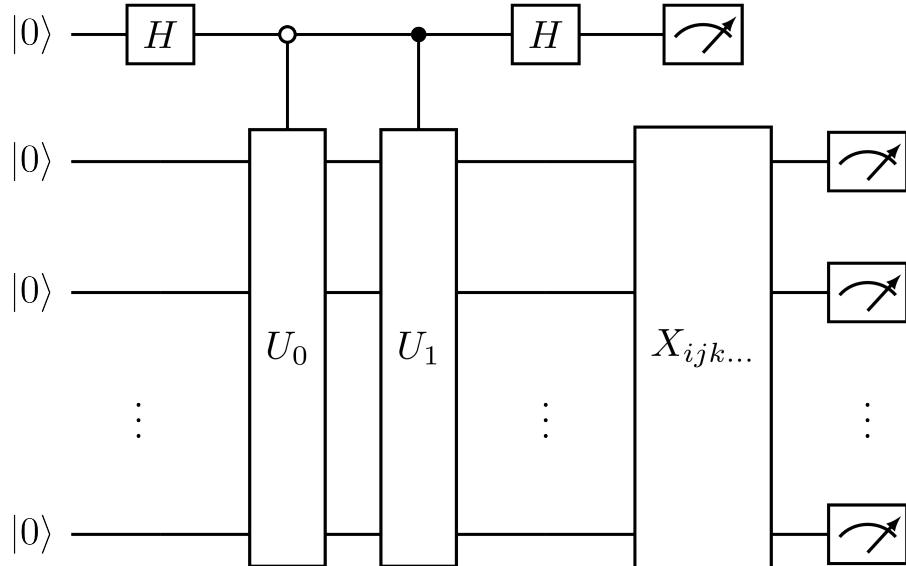


Fig. 7.10: Measurement circuit for the real part of the overlap, generated by `HadamardTestOverlap` with `direct=True`, where $X_{ijk\dots}$ is the set of circuits required to measure the matrix elements of a set of simultaneously measurable Pauli words. The first qubit is the ancilla and the remaining qubits comprise the state register.

In this case, measurement of the state register measures $\langle \Psi_0 + \Psi_1 | P_i | \Psi_0 + \Psi_1 \rangle$ when the ancilla is $|0\rangle$, and $\langle \Psi_0 - \Psi_1 | P_i | \Psi_0 - \Psi_1 \rangle$ when the ancilla is $|1\rangle$. By taking a linear combination of these outcomes we can retrieve $\langle \Psi_0 | P_i | \Psi_1 \rangle$. Thus, with `direct=True`, to compute the complex overlap with a kernel we require $2N_p$ circuits, where N_p is the number of simultaneously measurable sets of Pauli words in the kernel.

A simple example of using this protocol is given below:

```
from inquanto.operators import QubitOperator
from inquanto.states import QubitState, FermionState
from inquanto.ansatzes import FermionSpaceAnsatzUCCSD, HardwareEfficientAnsatz
from inquanto.computables import Overlap
from inquanto.protocols import HadamardTestOverlap
from pytket import OpType
from pytket.extensions.qiskit import AerBackend
from pytket.partition import PauliPartitionStrat

bra = HardwareEfficientAnsatz([OpType.Rx, OpType.Ry], QubitState([1, 1, 0, 0]), 2)
ket = FermionSpaceAnsatzUCCSD(4, FermionState([1, 1, 0, 0], 1))
params = (
    bra.state_symbols.construct_random().to_dict()
    | ket.state_symbols.construct_random().to_dict()
)
kernel = QubitOperator.from_string("(-0.1, Z0), (0.1, Z1), (0.25, X0 X1)")

ovlp = Overlap(bra, ket, kernel)

protocol = HadamardTestOverlap(
    AerBackend(),
    shots_per_circuit=int(12e3),
    direct=True,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets
)
protocol.build_from(params, ovlp)
protocol.run(seed=0)

circs = protocol.get_circuits()
print("Circuit count: ", len(circs))
ovlp.evaluate(protocol.get_evaluator())
```

Circuit count: 4

$(-0.05150833333333337+0.03003333333333332j)$

7.3.2 FactorizedOverlap

Protocols of the abstract type `FactorizedOverlap` work on pairs of ansatzes that satisfy the following conditions:

- Reference preparation can be factorized out from both state preparation unitaries i.e. $|\Psi_0\rangle = U_0 U_{\text{ref}} |\bar{0}\rangle$ and $|\Psi_1\rangle = U_1 U_{\text{ref}} |\bar{0}\rangle$
- Both ansatzes have the same reference state preparation circuit U_{ref}
- Non-reference factors U_0 and U_1 in the state preparation unitaries yield $|\bar{0}\rangle$ when acting on $|\bar{0}\rangle$; i.e. $U_0 |\bar{0}\rangle = U_1 |\bar{0}\rangle = |\bar{0}\rangle$.

Exploitation of these properties admits more efficient overlap measurement circuits that avoid application of a control to the entire state preparations as is done in `HadamardTestOverlap`. Rather, only the reference parts of the state preparations must be controlled by an ancilla qubit.

The $U_0 |\bar{0}\rangle = U_1 |\bar{0}\rangle = |\bar{0}\rangle$ property is enforced by raising a `TypeError` upon preparation of the protocol circuits if either of the specified ansatzes is of a type that is not guaranteed to satisfy it. All ansatz classes derived from `FermionSpaceStateExp` and `FermionSpaceStateExpChemicallyAware` are compatible with `FactorizedOverlap`. These are:

- `FermionSpaceStateExp`
- `FermionSpaceAnsatzkUpCCGD`
- `FermionSpaceAnsatzkUpCCGSD`
- `FermionSpaceAnsatzkUpCCGSDSinglet`
- `FermionSpaceAnsatzUCCGD`
- `FermionSpaceAnsatzUCCGSD`
- `FermionSpaceAnsatzUCCSDSinglet`
- `FermionSpaceAnsatzUCCSD`
- `FermionSpaceAnsatzUCCD`
- `FermionSpaceAnsatzChemicallyAwareUCCSD`
- `FermionSpaceStateExpChemicallyAware`

Note

The chemically aware ansatzes are only supported when both bra and ket state are chemically aware. The reason for this is that the chemically aware ansatz reference circuits are in *spatial* orbital Jordan–Wigner encoding, and so the reference state preparation circuit is not identical to the *spin*-orbital Jordan–Wigner encoded reference of the non-chemically aware ansatz, even if both ansatzes share the same fermionic reference state. See [here](#) for more details.

Currently, two classes derived from `FactorizedOverlap` type are implemented: `SwapFactorizedOverlap` and `ComputeUncomputeFactorizedOverlap`. These correspond to the two overlap measurement circuits presented in [32].

SwapFactorizedOverlap

This approach introduces an ancillary state register (lower in figure) to prepare the same linear combinations on the upper state register as in the case of the `HadamardTestOverlap`.

$$\frac{1}{2} [|0\rangle \otimes |\Psi_0 + \Psi_1\rangle + |1\rangle \otimes |\Psi_0 - \Psi_1\rangle] \quad (7.10)$$

Since this final state is identical to that of `HadamardTestOverlap`, `SwapFactorizedOverlap` is compatible with direct operator averaging as in the circuit shown below:

A simple example usage of the direct `SwapFactorizedOverlap` protocol is given below.

```
from inquanto.operators import QubitOperator
from inquanto.states import FermionState
from inquanto.ansatzes import FermionSpaceAnsatzkUpCCGSD
from inquanto.computables import Overlap
from inquanto.protocols import SwapFactorizedOverlap
from pytket.extensions.qiskit import AerBackend
```

(continues on next page)

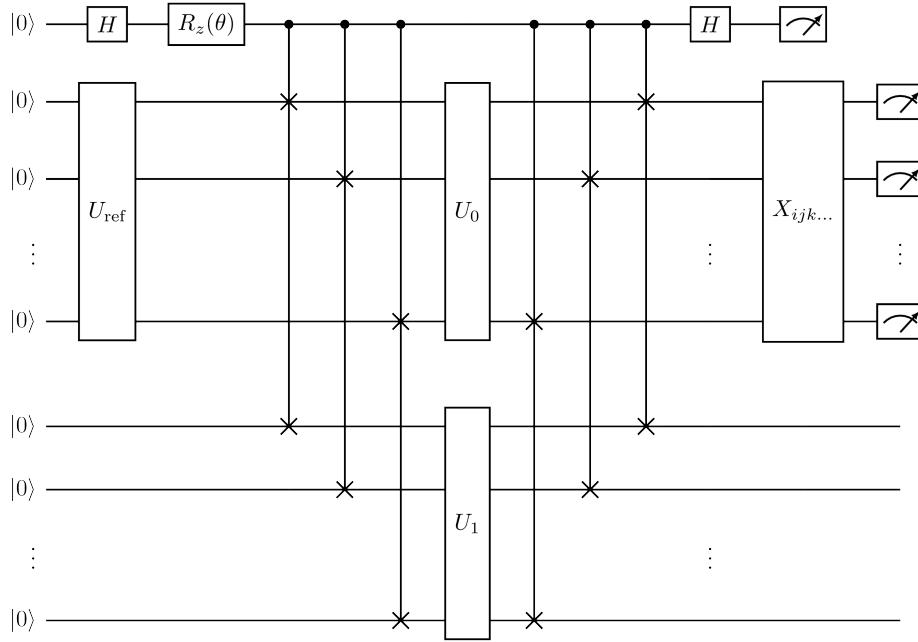


Fig. 7.11: Measurement circuit for the real ($\theta = 0$) or imaginary ($\theta = \pi/2$) part of the overlap, generated by `SwapFactorizedOverlap` with `direct=True`, where $X_{ijk\dots}$ is the set of circuits required to measure the matrix elements of a set of simultaneously measurable Pauli words. The first qubit is the ancilla and the remaining qubits comprise the two state registers.

(continued from previous page)

```
from pytket.partition import PauliPartitionStrat

bra = FermionSpaceAnsatzkUpCCGSD(4, FermionState([1, 1, 0, 0], 1), 2)
ket = FermionSpaceAnsatzkUpCCGSD(4, FermionState([1, 1, 0, 0], 1), 2)
bra.symbol_substitution('{}_bra')
ket.symbol_substitution('{}_ket')
params = (
    bra.state_symbols.construct_random(seed=1).to_dict()
    | ket.state_symbols.construct_random(seed=2).to_dict()
)
kernel = QubitOperator.from_string("(-0.1, Z0), (0.1, Z1), (0.25, X0 X1)")

ovlp = Overlap(bra, ket, kernel)

protocol = SwapFactorizedOverlap(
    AerBackend(),
    shots_per_circuit=int(12e3),
    direct=True,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets
)
protocol.build_from(params, ovlp)
protocol.run(seed=0)

circs = protocol.get_circuits()
print("Circuit count: ", len(circs))
```

(continues on next page)

(continued from previous page)

```
ovlp.evaluate(protocol.get_evaluator())
```

```
Circuit count: 4
```

```
(-0.0407833333333345+0.0032333333333337j)
```

Indirect measurement is also possible with this protocol, for which the required Pauli words are appended to the ket state preparation as shown below.

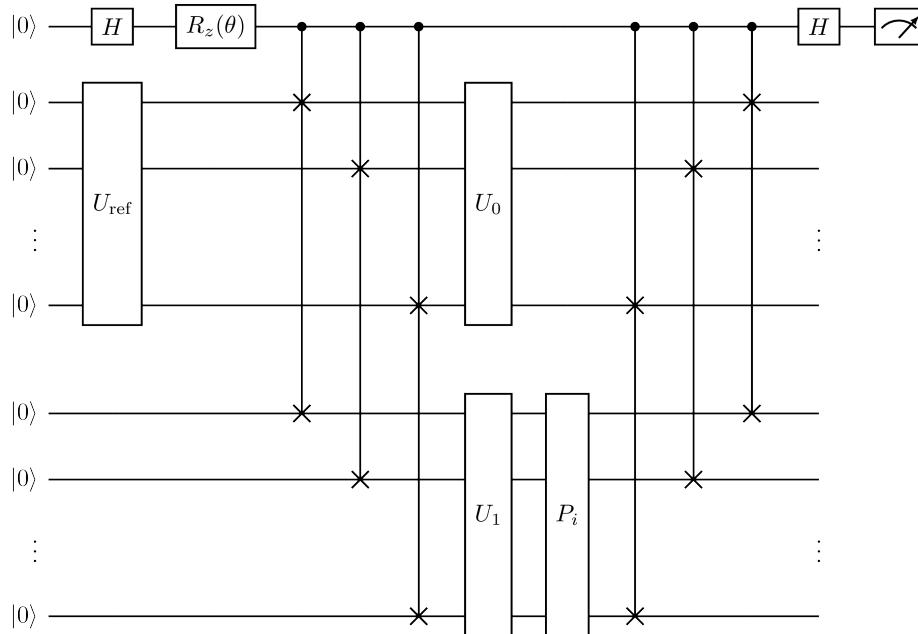


Fig. 7.12: Measurement circuit for the real ($\theta = 0$) or imaginary ($\theta = \pi/2$) part of an overlap with a non-identity kernel, generated by `SwapFactorizedOverlap` with `direct=False`, where P_i is the i th Pauli word appearing in the kernel. The first qubit is the ancilla and the remaining qubits comprise the two state registers.

A simple example usage of the indirect `SwapFactorizedOverlap` protocol is given below.

```
from inquanto.operators import QubitOperator
from inquanto.states import FermionState
from inquanto.ansatzes import FermionSpaceAnsatzkUpCCGSD
from inquanto.computables import Overlap
from inquanto.protocols import SwapFactorizedOverlap
from pytket.extensions.qiskit import AerBackend

bra = FermionSpaceAnsatzkUpCCGSD(4, FermionState([1, 1, 0, 0], 1), 2)
ket = FermionSpaceAnsatzkUpCCGSD(4, FermionState([1, 1, 0, 0], 1), 2)
bra.symbol_substitution('{}_bra')
ket.symbol_substitution('{}_ket')
params = (
    bra.state_symbols.construct_random(seed=1).to_dict()
    | ket.state_symbols.construct_random(seed=2).to_dict()
)
```

(continues on next page)

(continued from previous page)

```

kernel = QubitOperator.from_string("(-0.1, Z0), (0.1, Z1), (0.25, X0 X1)")

ovlp = Overlap(bra, ket, kernel)

protocol = SwapFactorizedOverlap(AerBackend(), shots_per_circuit=int(12e3), ↴
    ↪direct=False)
protocol.build_from(params, ovlp)
protocol.run(seed=0)

circs = protocol.get_circuits()
print("Circuit count: ", len(circs))
ovlp.evaluate(protocol.get_evaluator())

```

Circuit count: 6

(-0.0429666666666668-0.0006833333333333j)

ComputeUncomputeFactorizedOverlap

An alternative protocol in the `FactorizedOverlap` family is `ComputeUncomputeFactorizedOverlap`, which takes inspiration from the `ComputeUncompute` method for overlaps squared, and prepares the state:

$$\frac{1}{2} \left[|0\rangle \otimes \left(|\bar{0}\rangle + U_{\text{ref}}^\dagger U_1^\dagger U_0 U_{\text{ref}} |\bar{0}\rangle \right) + |1\rangle \otimes \left(|\bar{0}\rangle - U_{\text{ref}}^\dagger U_1^\dagger U_0 U_{\text{ref}} |\bar{0}\rangle \right) \right] \quad (7.11)$$

This is accomplished by the following circuit:

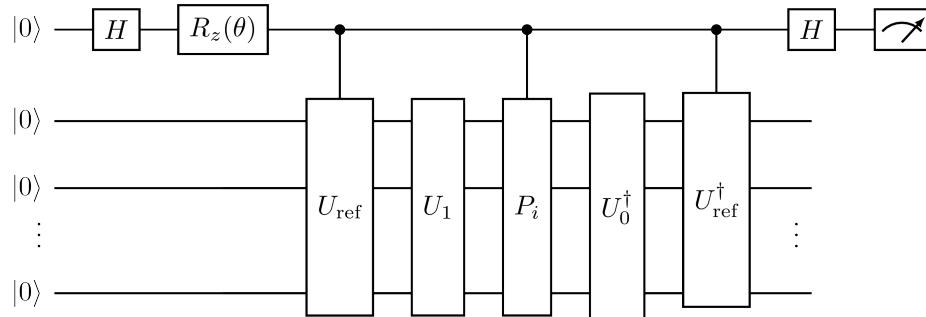


Fig. 7.13: Measurement circuit for the real ($\theta = 0$) or imaginary ($\theta = \pi/2$) part of an overlap with a non-identity kernel, generated by `ComputeUncomputeFactorizedOverlap`, where P_i is the i th Pauli word appearing in the kernel. The first qubit is the ancilla and the remaining qubits comprise the state register.

Which has the advantage of fewer qubits than `SwapFactorizedOverlap`, requiring only a single state register and one ancilla qubit, however it does produce deeper circuits. Since projection of Pauli words on the final state does not correspond to terms in the matrix element of the kernel, it is not possible to use this protocol in conjunction with the direct operator averaging scheme described above.

A simple example usage of the direct `ComputeUncomputeFactorizedOverlap` protocol is given below.

```

from inquanto.operators import QubitOperator
from inquanto.states import FermionState
from inquanto.ansatzes import FermionSpaceAnsatzkUpCCGSD

```

(continues on next page)

(continued from previous page)

```

from inquanto.computables import Overlap
from inquanto.protocols import ComputeUncomputeFactorizedOverlap
from pytket.extensions.qiskit import AerBackend

bra = FermionSpaceAnsatzkUpCCGSD(4, FermionState([1, 1, 0, 0], 1), 2)
ket = FermionSpaceAnsatzkUpCCGSD(4, FermionState([1, 1, 0, 0], 1), 2)
bra.symbol_substitution('{}_bra')
ket.symbol_substitution('{}_ket')
params = (
    bra.state_symbols.construct_random(seed=1).to_dict()
    | ket.state_symbols.construct_random(seed=2).to_dict()
)
kernel = QubitOperator.from_string("(-0.1, Z0), (0.1, Z1), (0.25, X0 X1)")

ovlp = Overlap(bra, ket, kernel)

protocol = ComputeUncomputeFactorizedOverlap(AerBackend(), shots_per_
    ↪circuit=int(12e3))
protocol.build_from(params, ovlp)
protocol.run(seed=0)

circs = protocol.get_circuits()
print("Circuit count: ", len(circs))
ovlp.evaluate(protocol.get_evaluator())

```

Circuit count: 6

(-0.04065000000000001+0.004800000000000016j)

7.4 Protocols for derivatives

Shot-based calculations of derivatives with respect to ansatz parameters are supported by three InQuanto protocols: *HadamardTestDerivative*, *PhaseShift*, and *HadamardTestDerivativeOverlap*, each of which has different capabilities. Given a parameterized state $|\psi(\theta)\rangle$, in general one may be interested in a range of different derivative computables. Those supported by one or more of the protocols in this section are:

- *ExpectationValueDerivative*, with a Hermitian kernel $H = \sum_i h_i P_i$ written as a linear combination of Pauli strings:

$$\frac{\partial}{\partial \theta_i} \langle \psi(\theta) | H | \psi(\theta) \rangle \quad (7.12)$$

- Bra or ket derivatives (see *ExpectationValueBraDerivativeReal* and related classes):

$$\left\langle \frac{\partial \psi}{\partial \theta_i} \middle| H \middle| \psi(\theta) \right\rangle \quad (7.13)$$

- Derivative overlaps, also referred to as the metric tensor (see *MetricTensorReal*):

$$\left\langle \frac{\partial \psi}{\partial \theta_i} \middle| \frac{\partial \psi}{\partial \theta_j} \right\rangle \quad (7.14)$$

In general, a parameterized ansatz in InQuanto may be written as:

$$|\psi(\theta_0, \theta_1, \dots)\rangle = U(\theta_0, \theta_1, \dots) |\bar{0}\rangle = \prod_i U_i(\theta_0, \theta_1, \dots) |\bar{0}\rangle \quad (7.15)$$

where each gate in the state preparation unitary may depend on any number of parameters. In all the methods discussed below, the ansatz is “regularized” such that each gate in the state preparation unitary contains only one parameter, each parameter appears only once, and all parameterized gates are single-qubit rotation gates. This regularization procedure is done by the protocols internally. In what follows we do so by rewriting the ansatz in terms of the dummy parameters γ_i (non-parameterized gates are implicitly included):

$$|\psi(\gamma_0, \gamma_1, \dots)\rangle = \prod_i U_i(\gamma_i) |\bar{0}\rangle \quad (7.16)$$

where the parameterized gates now take the form $U_j(\gamma_j) = \exp(-i\gamma_j P_j)$. The derivatives of interest are written using the chain rule:

$$\left| \frac{\partial \psi}{\partial \theta_j} \right\rangle = \sum_i J_{ij} \left| \frac{\partial \psi}{\partial \gamma_i} \right\rangle \quad (7.17)$$

where $J_{ij} = \partial \gamma_i / \partial \theta_j$ is the Jacobian matrix for this transformation. Derivatives with respect to regularized parameters γ_i are simpler to compute, and are the quantities computed at the circuit level in all of the protocols below.

7.4.1 HadamardTestDerivative

The partial (bra) derivative of an expectation value can be written as:

$$\left\langle \frac{\partial \psi}{\partial \gamma_k} \middle| \hat{O} \middle| \psi(\gamma) \right\rangle = -i \left\langle \bar{0} \middle| \hat{W}_0^{k-1\dagger} \hat{G}_k \hat{W}_k^{n\dagger} \hat{O} \middle| \psi(\gamma) \right\rangle \quad (7.18)$$

where $\hat{W}_k^n = U_n(\gamma_n) \dots U_{k+1}(\gamma_{k+1}) U_k(\gamma_k)$ and $\frac{\partial U}{\partial \gamma_k} = -i \hat{G}_k e^{-i\gamma_k \hat{G}_k}$ is the gate (unitary) derivative [33].

The `HadamardTestDerivative` protocol directly implements the above formula. The \hat{G}_k gate, which is a Pauli word in a regularized circuit, is placed in the middle of a parameterized circuit, controlled and measured on an ancilla qubit. A specific Pauli (Y or Z) gate being applied to the ancilla register determines whether one obtains real or imaginary part of the derivative. The Pauli strings constituting the observable can be either measured directly on the system register qubits as in `PauliAveraging` or also on the ancilla register as in `HadamardTest`.

```
from inquanto.express import load_h5
from inquanto.operators import QubitOperatorList
from inquanto.states import QubitState
from inquanto.ansatzes import TrotterAnsatz
from inquanto.computables import ExpectationValueBraDerivativeReal
from inquanto.protocols import HadamardTestDerivative
from pytket.extensions.qiskit import AerBackend

h2 = load_h5("h2_sto3g.h5", as_tuple=True)
hamiltonian = h2.hamiltonian_operator.qubit_encode()

exponents = QubitOperatorList.from_string("theta0 + 0.2*theta1 [(1j, Y0 X1 X2 X3)]")
reference = QubitState([1, 0, 0, 0])
ansatz = TrotterAnsatz(exponents, reference)

parameters = ansatz.state_symbols.construct_from_array([-0.111, -0.0555])

gradient_expression = ExpectationValueBraDerivativeReal(
    ansatz, hamiltonian, ansatz.free_symbols_ordered()
)

protocol = HadamardTestDerivative(
```

(continues on next page)

(continued from previous page)

```
AerBackend(), shots_per_circuit=50000, direct=True
)
protocol.build_from(parameters, gradient_expression)
protocol.run()

gradient_expression.evaluate(protocol.get_evaluator())
```

```
{theta0: -0.02431450531408247, theta1: -0.004862901062816496}
```

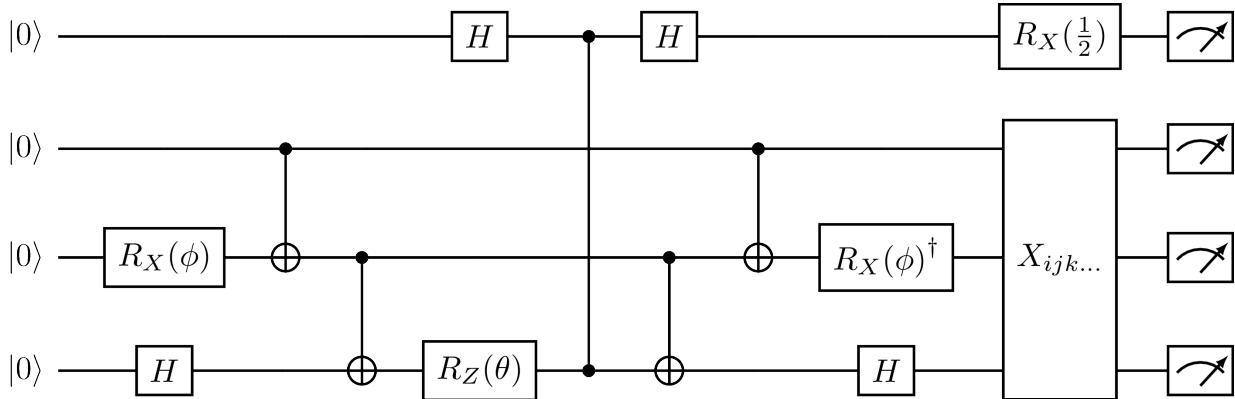


Fig. 7.14: Measurement circuit generated by `HadamardTestDerivative` with `direct=True`, where $X_{ijk\dots}$ is the set of gates required to measure a commuting set of Pauli terms in the Hamiltonian.

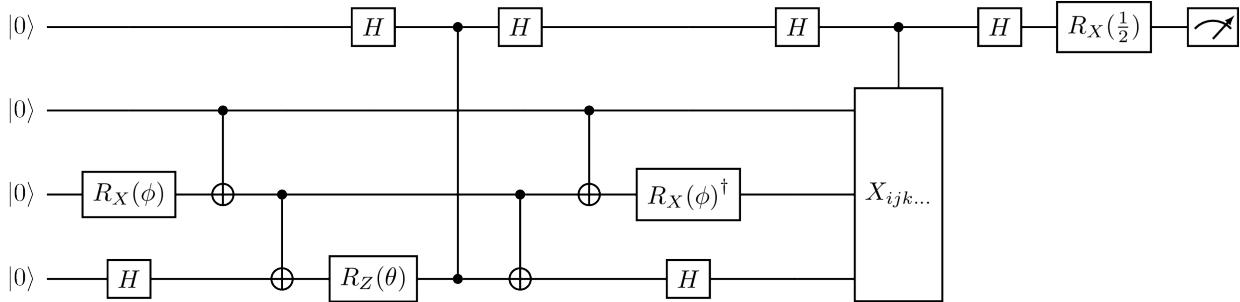


Fig. 7.15: Measurement circuit generated by `HadamardTestDerivative` with `direct=False`, where $X_{ijk\dots}$ is the set of gates required to measure a commuting set of Pauli terms in the Hamiltonian.

7.4.2 HadamardTestDerivativeOverlap

The `HadamardTestDerivativeOverlap` is a shot-based protocol for computing elements of the metric tensor, based on refs [30, 34]. In terms of the regularized parameters, this is given by:

$$\left\langle \frac{\partial \psi}{\partial \theta_i} \middle| \frac{\partial \psi}{\partial \theta_j} \right\rangle = \sum_{kl} J_{ki} J_{lj} \left\langle \frac{\partial \psi}{\partial \gamma_k} \middle| \frac{\partial \psi}{\partial \gamma_l} \right\rangle \quad (7.19)$$

The regularized derivative overlap is written as:

$$\left\langle \frac{\partial\psi}{\partial\gamma_k} \middle| \frac{\partial\psi}{\partial\gamma_l} \right\rangle = \langle \bar{0} \mid V_k(\gamma)^\dagger V_l(\gamma) \mid \bar{0} \rangle \quad (7.20)$$

where $V_i(\gamma)$ is the i-th derivative of the regularized state preparation unitary:

$$V_i(\gamma) = U_n(\gamma_n)U_{n-1}(\gamma_{n-1})\dots U_i(\gamma_i)P_i\dots U_1(\gamma_1)U_0(\gamma_0) \quad (7.21)$$

This overlap may be evaluated with a modified Hadamard test. The Pauli strings injected by the derivatives are carefully applied to the state register, controlled on an ancilla qubit in a superposition state. For a calculation of the real part of the metric tensor (with $k < l$), this circuit is given by:

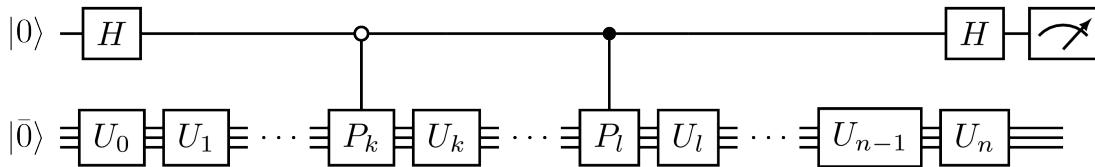


Fig. 7.16: Measurement circuit for the real part of the regularized derivative overlap: $\text{Re} \langle \partial\psi/\partial\gamma_k \mid \partial\psi/\partial\gamma_l \rangle$ generated by HadamardTestDerivativeOverlap.

As with a standard Hadamard test, the regularized derivative overlap is then given by

$$\text{Re} \left\langle \frac{\partial\psi}{\partial\gamma_k} \middle| \frac{\partial\psi}{\partial\gamma_l} \right\rangle = p(0) - p(1) \quad (7.22)$$

where $p(b)$ is the probability of measuring the ancilla qubit in state $b \in 0, 1$. Equation (7.19) then provides the derivatives in terms of the original ansatz parameters. See a simple example below for a UCCSD ansatz:

```
from inquanto.express import get_system
from inquanto.ansatzen import FermionSpaceAnsatzUCCSD
from inquanto.protocols import HadamardTestDerivativeOverlap

_, fermion_space, hf_state = get_system("h2_sto3g.h5")
ansatz = FermionSpaceAnsatzUCCSD(
    fermion_space, hf_state
)
params = ansatz.state_symbols.construct_random()
symbols = ansatz.state_symbols.symbols
print(params)

protocol = HadamardTestDerivativeOverlap(AerBackend(), shots_per_circuit=5000)
protocol.build(
    parameters=params,
    state=ansatz,
    diff_symbols=(symbols[1], symbols[1])
)
protocol.run()
protocol.evaluate_derivative_overlap()
```

```
{d0: 0.9417154046806644, s0: -1.3965781047011498, s1: -0.6797144480784211}
```

```
{(s0, s0): 1.000000000000000}
```

We specify above that the `HadamardTestDerivativeOverlap` protocol should build circuits to calculate the (s_0, s_0) element of the metric tensor. The `get_dataframe_derivative_overlap()` method provides a breakdown of the circuits required for this calculation:

```
print(protocol.get_dataframe_derivative_overlap())
```

	symbols	regularized symbols	prefactor	circuit	index
0	(s_0, s_0)	(u_{14}, u_{14})	0.2500000000000000		NaN
1	(s_0, s_0)	(u_{14}, u_{17})	-0.2500000000000000		0.0
2	(s_0, s_0)	(u_{17}, u_{14})	-0.2500000000000000		0.0
3	(s_0, s_0)	(u_{17}, u_{17})	0.2500000000000000		NaN

which shows that only one circuit was required for this calculation. It also reveals that the ansatz regularization replaced two occurrences of the symbol s_0 with the parameters u_{14} and u_{17} in the 14th and 17th gates of the state preparation unitary respectively. The prefactor column shows the Jacobian product $J_{ki}J_{lj}$. This protocol attempts to minimize the total number of circuits required by exploiting the fact that $\text{Re } \langle \partial_k \psi | \partial_l \psi \rangle = \text{Re } \langle \partial_l \psi | \partial_k \psi \rangle$, and that diagonal elements are unity. Rows with a NaN circuit index mark these diagonal terms.

7.5 Other averaging protocols

Here, we detail additional shot-based protocols available in InQuanto.

7.5.1 ProjectiveMeasurements

A state $|\Psi\rangle$ prepared on a digital quantum computer may be written as a linear combination of computational basis states:

$$|\Psi\rangle = \sum_i c_i |i\rangle \quad (7.23)$$

where i is a bit string. For example, a 2-qubit state is given by:

$$|\Psi\rangle = c_{00}|00\rangle + c_{10}|10\rangle + c_{01}|01\rangle + c_{11}|11\rangle. \quad (7.24)$$

The `ProjectiveMeasurements` protocol measures the magnitude squared of the expansion coefficients, $|c_i|^2$, by preparing the state and measuring the register in the Z basis. The probability of measuring the bitstring i is precisely $|c_i|^2$. See below for a simple example:

```
from pytket.extensions.qiskit import AerBackend
from inquanto.express import get_system
from inquanto.ansatzes import FermionSpaceAnsatzUCCSD
from inquanto.protocols import ProjectiveMeasurements

_, space, state = get_system("h2_sto3g.h5")

ansatz = FermionSpaceAnsatzUCCSD(space, state)
params = ansatz.state_symbols.construct_random()

protocol = ProjectiveMeasurements(
```

(continues on next page)

(continued from previous page)

```

backend=AerBackend(),
shots_per_circuit=1000,
)
protocol.build(params, ansatz).run(seed=0)

print(protocol.get_dataframe_basis_states(4))

```

Basis	State	Probability	Uncertainty	Count
0	0110	0.600	0.015492	600
1	0011	0.224	0.013184	224
2	1100	0.164	0.011709	164
3	1001	0.012	0.003443	12

7.6 Protocols for Phase Estimation

Quantum phase estimation (QPE) [11, 12, 13, 14] is a quantum algorithm used to estimate the phase $\phi \in [0, 1]$ of a given unitary operator U and eigenstate $|\phi\rangle$ satisfying

$$U|\phi\rangle = e^{i2\pi\phi}|\phi\rangle. \quad (7.25)$$

There are two main approaches to QPE algorithms, i.e., QPE based on the quantum Fourier transform (QFT) and QPE based on classical post-processing. The former is referred to as canonical QPE. It requires as many ancilla qubits as is necessary for representing the phase to the desired precision. The latter is called iterative QPE. The phase value is inferred by analyzing the samples obtained with the basic measurement operation with one ancilla qubit. Currently, InQuanto supports the protocol layer for the iterative QPE algorithms.

7.6.1 IterativePhaseEstimation

Iterative QPE algorithms commonly use the circuit below.

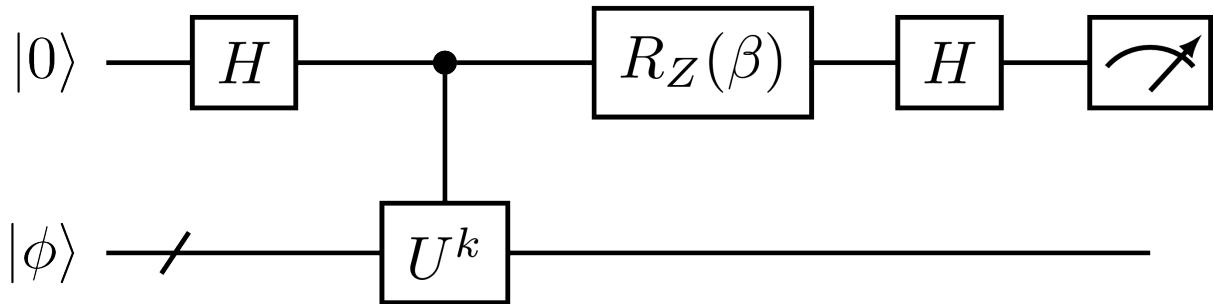


Fig. 7.17: Iterative QPE circuit parameterized by $k \in \mathbb{N}$ and $\beta \in [0, 2\pi)$.

Conceptually, `IterativePhaseEstimation` serves as a black box taking (k, β) as input to return the measurement sample(s) $m \in \{0, 1\}$ as output. The probability of measuring m from this circuit is expressed as

$$P(m|\phi, k, \beta) = \frac{1 + \cos(k\phi + \beta - m\pi)}{2} \quad (7.26)$$

The iterative QPE protocols return samples of m to be post-processed by the higher-level routine, such as the objects of the algorithm layer of InQuanto.

To demonstrate the primary usage of iterative QPE protocols, consider the example below with the following setup for example:

$$U = U_1(\alpha) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i2\pi\alpha} \end{pmatrix} \quad (7.27)$$

and the initial state is

$$|\phi\rangle = |1\rangle \quad (7.28)$$

In this case, the phase readout will be $2\pi k\alpha$. Consider the case of $\beta = -2\pi k\alpha$ to always obtain measurement outcome $m = 0$ for convenience.

To begin, we must build some objects required for the iterative QPE protocol.

```
from pytket.circuit import Circuit, OpType
from pytket.extensions.qiskit import AerBackend
from inquanto.protocols import IterativePhaseEstimation

# Eigenphase for example.
alpha = 1.0 / 8

# Protocol parameter.
n_shots = 10

# Prepare a function to return the ctrl-U circuit.
def get_ctrlu(k: int) -> Circuit:
    ctrlu = Circuit(2)
    ctrlu.add_gate(OpType.CU1, 2 * k * alpha, [0, 1])
    return ctrlu

# Prepare the circuit for the state preparation.
init_state = Circuit(1).X(0)

# pytket Backend.
backend = AerBackend()
```

Then we construct and build the `IterativePhaseEstimation` object.

```
# Initialize and build the protocol, and then set the circuit parameter.
protocol = IterativePhaseEstimation(
    backend=backend,
    n_shots=n_shots,
).build_from_circuit(
    get_ctrlu=get_ctrlu,
    state=init_state,
);

# Set the circuit parameter.
k = 4
beta = -2 * k * alpha # To ensure the measurement outcome is always 0.
protocol.update_k_and_beta(k, beta)
```

(continues on next page)

(continued from previous page)

```
# Run the protocol to get the measurement outcome
protocol.run()
protocol.get_measurement_outcome()
```

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

To make it more like a chemistry problem, one can pass the `QubitOperatorList` as the exponents to be trotterized as $U = e^{-iHt}$.

```
from inquanto.operators import QubitOperator
from inquanto.ansatzes import CircuitAnsatz

# Input for generating the CTRL-U gate.
time = 1.0
hamiltonian = QubitOperator("Z0", 0.5)
qopl = hamiltonian.trotterize()

# Ansatz circuit for the state preparation.
prep_ansatz = CircuitAnsatz(Circuit(1))

# Construct and build the protocol, then update the circuit parameter.
protocol = IterativePhaseEstimation(
    backend=backend,
    n_shots=n_shots,
).build(
    state=prep_ansatz,
    evolution_operator_exponents=qopl * time,
)

# Set the circuit parameters.
k = 10
beta = 0.56789
protocol.update_k_and_beta(k, beta)

# Run the protocol.
protocol.run()

# Get the measurement outcome.
protocol.get_measurement_outcome()
```

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

The measurement outcome may be a random sequence of 0 and 1, the probability of which is determined by (7.26).

This protocol may be helpful as a subroutine of iterative QPE algorithms. The higher-level workflow should manage the classical pre- and post-processing methods. The pre-processing here means the generation of the samples of (k, β) , whereas post-processing implies the interpretation of measurement outcome m on classical computers, which could be the evaluation of the expectation value $\langle e^{-iHt} \rangle$, the inference procedure using the likelihood (7.26), and so on, depending on the iterative QPE algorithm. Once the protocol object is built, `update_k_and_beta()`, `run()`, and `get_measurement_outcome()` are used by those higher-level algorithms without concerning the inside of the protocol object. See [Iterative QPE algorithms section](#) for more about the application.

7.6.2 IterativePhaseEstimationQuantinum

IterativePhaseEstimation is designed for a noiseless backend to be free from backend-specific options. However, such options are essential for experiments on a noisy backend such as real quantum hardware. *IterativePhaseEstimationQuantinum* is designed for a noisy simulation to get the most out of the pytket Quantinuum backend.

One of the remarkable features provided by this class is the logical qubit encoding using an error detection code [35]. Users can perform logical qubit experiments of iterative QPE algorithms, thanks to the features of Quantinuum H-Series hardware, including high-fidelity operations for all-to-all connected qubits, mid-circuit measurement, qubit reuse, and conditional logic.

Currently, the $[[n+2, n, 2]]$ error detection code (dubbed iceberg code) proposed by C. Self, M. Benedetti, D. Amaro [35] is available. The experimental demonstration of the iterative QPE for a hydrogen molecule with the iceberg code has been performed by K. Yamamoto, S. Duffield, Y. Kikuchi, and David Muñoz Ramo [20].

First let us prepare the objects to build *IterativePhaseEstimationQuantinum*

```
from pytket.circuit.display import render_circuit_jupyter
from pytket.extensions.quantinum import QuantinuumBackend
from inquanto.operators import QubitOperator

# pytket Backend.
# backend = QuantinuumBackend(device_name="H1-1E")
backend = AerBackend()

hamiltonian = QubitOperator("Z0", 0.25) + QubitOperator("X0", -0.33)
qopl = hamiltonian.trotterize()

# Ansatz circuit for the state preparation.
prep_ansatz = CircuitAnsatz(Circuit(1))

# Circuit parameters.
k = 4
beta = 0.25
```

This simple example of *IterativePhaseEstimationQuantinum* is almost compatible with *IterativePhaseEstimation* as

```
from inquanto.protocols import (
    IterativePhaseEstimationQuantinum,
    CompilationLevelQuantinum,
    CircuitEncoderQuantinum,
)
# Construct and build the protocol, then update the circuit parameter.
protocol = IterativePhaseEstimationQuantinum(
    backend=backend,
    n_shots=n_shots,
    compilation_level=CompilationLevelQuantinum.ENCODED,
).build(
    state=prep_ansatz,
    evolution_operator_exponents=qopl,
    encoding_method=CircuitEncoderQuantinum.PLAIN,
).update_k_and_beta(k, beta)

# Display the encoded circuit.
```

(continues on next page)

(continued from previous page)

```
circ = protocol.get_circuits()[0]
render_circuit_jupyter(circ)
```

```
<IPython.core.display.HTML object>
```

Note that we can control the backend-specific compilation by specifying `compilation_level`. One can activate the logical qubit encoding with the `encoding_method` option as

```
from inquanto.protocols import IcebergOptions

# Construct and build the protocol, then update the circuit parameter.
protocol = IterativePhaseEstimationQuantinum(
    backend=backend,
    n_shots=n_shots,
    compilation_level=CompilationLevelQuantinum.ENCODED,
).build(
    state=prep_ansatz,
    evolution_operator_exponents=qopl,
    encoding_method=CircuitEncoderQuantinum.ICEBERG,
    encoding_options=IcebergOptions(
        syndrome_interval=2,
        sx_insertion=True,
    )
).update_k_and_beta(k, beta)

# Display the encoded circuit.
circ = protocol.get_circuits()[0]
render_circuit_jupyter(circ)
```

```
<IPython.core.display.HTML object>
```

The `encoding_method` may be accompanied by `encoding_options` to set the encoding-method-specific options. In this case, we set the `syndrome_interval=2` to perform the syndrome measurement (error detection) [35], and `sx_insertion=True` to enable the simple dynamical decoupling for the error suppression [20]. Although the circuits look very different from each other, the key methods such as `update_k_and_beta()` and `get_measurement_outcome()` work essentially in the same manner, i.e., the encoding of circuits and the decoding of measurement outcomes are performed within the protocol layer.

Circuit optimization strategy may differ from the unencoded physical qubit implementation if we use the logical qubit encoding. To help the circuit optimization, `IterativePhaseEstimationQuantinum` has additional options, such as `terms_map` and `paulis_map`.

The `terms_map` option of the `build()` method is used for replacing a Pauli string of H with another Pauli string. Typically, we use it to introduce a “dummy” qubit [20] to enable the logical qubit encoding by the iceberg code, as the iceberg code requires a code block consisting of even number qubits. Let us consider the following qubit Hamiltonian, for example.

$$H = gX_0X_1, \quad (7.29)$$

where $g \in \mathbb{R}$ is a coefficient. We can easily convert it into $H = gI_0X_1X_2$ without reconstructing the Hamiltonian by using `terms_map` option as

```

from pytket.circuit import Qubit
from pytket.pauli import Pauli, QubitPauliString
from inquanto.operators import QubitOperator

# Prepare a Hamiltonian.
qpol = QubitOperator("X0 X1", 0.25).trotterize()

# Prepare the terms_map object.
qps_xx = QubitPauliString([Qubit(0), Qubit(1)], [Pauli.X, Pauli.X])
qps_ixx = QubitPauliString([Qubit(1), Qubit(2)], [Pauli.X, Pauli.X])
terms_map = {qps_xx: qps_ixx}

# Prepare the state.
init_state = CircuitAnsatz(Circuit(3))

```

Then, construct the protocol to generate the encoded circuit as

```

# Initialize and build the protocol.
protocol = IterativePhaseEstimationQuantinum(
    backend=None,
    compilation_level=CompilationLevelQuantinum.ENCODED,
).build(
    state=init_state,
    evolution_operator_exponents=qpol,
    encoding_method=CircuitEncoderQuantinum.ICEBERG,
    terms_map=terms_map,
).update_k_and_beta(k=1, beta=0.0)

# Display the circuit.
circ = protocol.get_circuits()[0]
render_circuit_jupyter(circ)

```

<IPython.core.display.HTML object>

The dummy qubit undergoes no logical operations in general, but we may use it for circuit optimization. For example, the X operator acting on this dummy qubit serves as a stabilizer if the dummy qubit is initialized as $|+\rangle$. We can use this additional stabilizer for the circuit optimization to obtain a Pauli string to optimize the circuit by exploiting the other stabilizers.

```

# Set the Pauli map object.
qubits = [Qubit(i) for i in range(4)]
qps_zixx = QubitPauliString(qubits, [Pauli.Z, Pauli.I, Pauli.X, Pauli.X])
qps_zxxx = QubitPauliString(qubits, [Pauli.Z, Pauli.X, Pauli.X, Pauli.X])
paulis_map = {qps_zixx: qps_zxxx}

# Initialize and build the protocol, and then set the circuit parameter.
protocol = IterativePhaseEstimationQuantinum(
    backend=None,
    compilation_level=CompilationLevelQuantinum.ENCODED,
).build(
    state=init_state,
    evolution_operator_exponents=qpol,
    encoding_method=CircuitEncoderQuantinum.ICEBERG,

```

(continues on next page)

(continued from previous page)

```

encoding_options=IcebergOptions(
    n_plus_states=2,
),
terms_map=terms_map,
paulis_map=paulis_map,
).update_k_and_beta(k=1, beta=0.0)

# Display the circuit.
circ = protocol.get_circuits()[0]
render_circuit_jupyter(circ)

```

```
<IPython.core.display.HTML object>
```

See [20] for technical details.

7.6.3 IterativePhaseEstimationStatevector

Some forms of iterative QPE (such as Kitaev's QPE) use the probability distribution of measurement outcome m , rather than the sequence of m . Exact probability distribution will be useful for quick and reliable prototyping. `IterativePhaseEstimationStatevector` has such a feature, `get_distribution()`.

```

from inquanto.protocols import IterativePhaseEstimationStatevector

# Eigenphase in the pytket convention (angle is in the unit of half turn).
phase = 1.0 / 8

# Protocol parameters.
k = 4
beta = -2 * k * phase    # To make sure the measurement outcome is always 0.
n_shots = 10

# Prepare a function to return the ctrl-U circuit.
def get_ctrlu(k: int) -> Circuit:
    ctrlu = Circuit(2)
    ctrlu.add_gate(OpType.CU1, 2 * k * phase, [0, 1])
    return ctrlu

# Prepare the circuit for the state preparation.
init_state = Circuit(1).X(0)

# pytket Backend.
backend = AerBackend()

# Initialize and build the protocol, and then set the circuit parameter.
protocol = IterativePhaseEstimationStatevector(
).build_from_circuit(
    get_ctrlu=get_ctrlu,
    state=init_state,
).update_k_and_beta(k, beta)

protocol.run()
protocol.get_distribution()

```

```
{(0,): 1.0, (1,): 0.0}
```

Internally, this class classically diagonalizes the unitary to obtain the eigenvalues and calculate the eigenstate population of the initial state to weight the likelihood functions. Note that this exact diagonalization may take a long time, even for a modest system size.

7.7 Resource estimation

Jobs submitted to quantum backends need to be designed to obtain sensible results within a realistic time. In principle, the resource requirements of an experiment can be estimated based on the circuits and the number of shots to be run. The effect of the noise may be predicted using an emulator equipped with a noise model that mimics the behavior of the real hardware. However, it is a good practice to step back and check the circuits before submitting to any quantum backends to consider the feasibility. InQuanto Protocols have methods that help the users perform such a circuit analysis for resource estimation.

The `dataframe_circuit_shot()` method is available in all Protocols with the `get_circuits()` and `get_shots()` methods. This method quickly displays a summary of circuits to be run, and is a helpful tool for quickly checking the effect of the circuit optimization techniques. Below is a simple example of resource estimation in a typical InQuanto workflow for the energy expectation value calculations.

In the example below we use pytket qiskit's `AerBackend`, but we note that costs will depend on the backend due to differences in circuit compilation (e.g., for qubit architecture).

```
# Prepare the pytket Backend object.

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

Prepare the qubit Hamiltonian and ansatz as follows, with help from the `express` module:

```
# Evaluate the Hamiltonian.
from inquanto.express import get_system
from inquanto.ansatze import FermionSpaceAnsatzUCCSD

fham, fsp, fst_hf = get_system('h2_sto3g.h5')
qham = fham.qubit_encode()
ansatz = FermionSpaceAnsatzUCCSD(fsp, fst_hf)
params = ansatz.state_symbols.construct_random()
```

Then, prepare the `Computable` and `Protocol` objects. The summary of the circuit and the number of shots is displayed as a pandas dataframe.

```
# Prepare the energy expectation value calculations with PauliAveraging.
from inquanto.computables import ExpectationValue
from inquanto.protocols import PauliAveraging

computable = ExpectationValue(ansatz, qham)
protocol = PauliAveraging(backend=backend, shots_per_circuit=8000)
protocol.build_from(params, computable)

# Use the protocol research estimation tool.
protocol.dataframe_circuit_shot()
```

	Qubits	Depth	Depth2q	DepthCX	Shots
0	4	38	21	21	8000
1	4	39	21	21	8000
2	4	39	21	21	8000
3	4	39	21	21	8000
4	4	39	21	21	8000
Sum	-	-	-	-	40000

The `build_from()` method uses pytket to optimize and compile circuits to the target backend, and `dataframe_circuit_shot()` displays basic information, such as the 2-qubit gate count. If further details are required, circuit analysis using pytket and its extensions may be performed as follows:

```
from pytket.circuit import OpType

# Get the circuits to be measured.
circuits = protocol.get_circuits()
shots = protocol.get_shots()

# Start the analysis.
circ = circuits[0]
shot = shots[0]
data = {}
data['shots'] = shot
data['depth'] = circ.depth()
data['CX count'] = circ.n_gates_of_type(OpType.CX)
data['CX depth'] = circ.depth_by_type(OpType.CX)

# Show data.
for k, v in data.items():
    print(f"{k:10s}: {v}")
```

```
shots      : 8000
depth      : 38
CX count   : 22
CX depth   : 21
```

For the NISQ algorithms, the number of two-qubit gates (e.g., CX, ZZPhase) is the primary limiting factor in obtaining sensible results. One should always check the error rate of the two-qubit gates for a backend in question. Note that any cost estimation is performed at the Protocol level, but the total cost depends on the algorithm executed. For example, VQE needs more shots to drive the feedback loop, and Bayesian QPE needs more shots with growing circuits. The number of repeats/loops needed cannot generally be predicted.

Some backends support estimation of the cost from the quota given by the backend provider. For example, the Quantinuum backend has the `cost()` method to estimate the cost in H-Series Quantum Credits (HQC):

```
from pytket.circuit import Circuit
from pytket.extensions.quantinuum import QuantinuumBackend

# Initialize the quantinuum backend.
qtnm_backend = QuantinuumBackend("H1-1E")
circ = Circuit(2)
circ.H(0).CX(0, 1).measure_all()
```

(continues on next page)

(continued from previous page)

```
# Cost estimate.
qtnm_circ = qtnm_backend.get_compiled_circuit(circ) #login needed
qtnm_backend.cost(qtnm_circ, n_shots=8000, syntax_checker="H1-1SC")

# output: 57.8
```

Circuit optimization may also be performed prior to optimization at the pytket level. Here we demonstrate such an optimization performed outside of pytket. We use an example demonstrating basic usage of the iterative quantum phase estimation (QPE) protocol with an error detection code [35]. See the *QPE protocol* manual for details. Here, we focus on showing the potential for circuit optimization; for theoretical details, see ref [20].

Let us prepare the two-qubit Hamiltonian (See *Symmetry* for the method to taper off the qubits) describing the molecular hydrogen molecule as an example:

```
# Prepare the target system.
from pytket.circuit import Circuit, PauliExpBox, Pauli
from inquanto.operators import QubitOperator
from inquanto.ansatzes import CircuitAnsatz

# Qubit operator.
# Two-qubit H2 with the equilibrium geometry.
qop = QubitOperator.from_string(
    "(-0.398, Z0), (-0.398, Z1), (-0.1809, Y0 Y1)",
)
qop_totally_commuting = QubitOperator.from_string(
    "(0.0112, Z0 Z1), (-0.3322, )",
)

# Parameters for constructing a function to return controlled unitary.
time = 0.1
n_trotter = 1
evo_ope_exp = qop.trotterize(trotter_number=n_trotter) * time
eoe_tot_com = qop_totally_commuting.trotterize(trotter_number=n_trotter) * time
k = 50
beta = 0.5
ansatz_parameter = -0.07113
```

Now we generate the circuit with the (unencoded) physical circuits for reference:

```
from inquanto.protocols import IterativePhaseEstimationQuantinuum,
    CircuitEncoderQuantinuum

# Prepare the IterativePhaseEstimationQuantinuum object.
from pytket.circuit import Qubit
from pytket.pauli import QubitPauliString

# State preparation circuit.
# Introduce the dummy qubit to satisfy the requirement of the iceberg code by the
# Hamiltonian terms mapping: P -> IP
# There is no effect for the unencoded circuits.
terms_map = {
    QubitPauliString([Qubit(0)], [Pauli.Z]): QubitPauliString([Qubit(1)], [Pauli.Z]),
    QubitPauliString([Qubit(1)], [Pauli.Z]): QubitPauliString([Qubit(2)], [Pauli.Z]),
```

(continues on next page)

(continued from previous page)

```

QubitPauliString([Qubit(0), Qubit(1)], [Pauli.Y, Pauli.Y]): QubitPauliString(
    [Qubit(1), Qubit(2)], [Pauli.Y, Pauli.Y]
),
QubitPauliString([Qubit(0), Qubit(1)], [Pauli.Z, Pauli.Z]): QubitPauliString(
    [Qubit(1), Qubit(2)], [Pauli.Z, Pauli.Z]
),
QubitPauliString(): QubitPauliString(),
}

# State preparation circuit.
state = Circuit(3)
state.add_pauliexpbox(
    PauliExpBox([Pauli.I, Pauli.Y, Pauli.X], ansatz_parameter),
    state.qubits,
)
state_prep = CircuitAnsatz(state)

# Preparing the protocol without circuit encoding.
protocol = IterativePhaseEstimationQuantinuum(
    backend=backend,
    optimisation_level=0,    # For the clear comparison
    n_shots=10,
)
protocol.build(
    state=state_prep,
    evolution_operator_exponents=evo_ope_exp,
    eoe_totally_commuting=eoe_tot_com,
    encoding_method=CircuitEncoderQuantinuum.PLAIN,    # Meaning no logical qubit
    ↪encoding is performed.
    terms_map=terms_map,
)
protocol.update_k_and_beta(k=k, beta=beta)

# Show the circuit and shot information.
protocol.dataframe_circuit_shot()

```

	Qubits	Depth	Shots	k	beta	TQ	CX	ZZPhase
0	4	560	10	50	0.5	305	102	203

Note

If the purpose is to generate an unencoded circuit for the physical qubit experiments, the general purpose `IterativePhaseEstimation` will simplify the code.

Now we generate the circuit encoded by the $[[6, 4, 2]]$ error detection code (dubbed iceberg code) [35]. The controlled unitary of the unencoded circuit includes two `CRz=RzzRiz` gates and one `CRyy=RzyyRiy` gate that linearly scales as k increases. This part requires four `ZZPhase` and two `CX` gates to represent this operation.

We introduce (still in the snippet above) a dummy qubit with no logical operation to make the system consist of an even number of qubits. The `terms_map` option may be used for introducing the dummy qubit without changing the qubit Hamiltonian defined as a logical operator. The state preparation circuit needs to take the dummy qubit into account.

Then, we construct the encoded circuit and perform the resource estimation similarly to the unencoded circuits.

```
# Prepare the IterativePhaseEstimationQuantinuum object.
protocol = IterativePhaseEstimationQuantinuum(
    backend=backend,
    optimisation_level=0,    # For the clear comparison
    n_shots=10,
)
protocol.build(
    state=state_prep,
    evolution_operator_exponents=evo_ope_exp,
    eoe_totally_commuting=eoe_tot_com,
    encoding_method=CircuitEncoderQuantinuum.ICEBERG,
    terms_map=terms_map,
)
protocol.update_k_and_beta(k=k, beta=beta)
protocol.dataframe_circuit_shot()
```

	Qubits	Depth	Shots	k	beta	TQ	CX	ZZPhase
0	8	629	10	50	0.5	523	218	305

Now, we perform some circuit optimization at the logical circuit level and then analyze the resource requirements. This optimization consists of the following:

- Basis rotation: $Y \rightarrow X$ (X is cheaper than Y in the iceberg code)
- Initialize the dummy qubit in the logical $|+\rangle$ state so that the Pauli X acting on this qubit becomes a stabilizer.
- Replace the logical $ZIXX$ with $ZXXX$ for the more efficient encoding to reduce the number of CXs.

```
from inquanto.protocols import IcebergOptions

# Change the basis: X -> Y.
terms_map_y2x = terms_map.copy()
terms_map_y2x = {
    QubitPauliString([Qubit(0), Qubit(1)], [Pauli.Y, Pauli.Y]): QubitPauliString(
        [Qubit(1), Qubit(2)], [Pauli.X, Pauli.X]
    ),
}

# State preparation circuit.
state = Circuit(3)
state.add_pauliexpbox(
    PauliExpBox([Pauli.I, Pauli.Y, Pauli.X], -0.07113),
    state.qubits,
)
state.Sdg(1)
state.Sdg(2)
state_prep = CircuitAnsatz(state)

# Use the optimization technique with the dummy qubit.
# Pauli operator mapping primary for the iceberg code for the circuit optimization.
qubits = [Qubit(i) for i in range(4)]
paulis_map = {
    QubitPauliString(qubits, [Pauli.Z, Pauli.I, Pauli.X, Pauli.X]): QubitPauliString(
        (continues on next page)
```

(continued from previous page)

```

        qubits, [Pauli.Z, Pauli.X, Pauli.X, Pauli.X]
    )
}
protocol = IterativePhaseEstimationQuantinuum(
    backend=backend,
    optimisation_level=0,    # For the clear comparison
    n_shots=10,
)
protocol.build(
    state=state_prep,
    evolution_operator_exponents=evo_ope_exp,
    eoe_totally_commuting=eoe_tot_com,
    encoding_method=CircuitEncoderQuantinuum.ICEBERG,
    encoding_options=IcebergOptions(
        n_plus_states=2,
    ),
    terms_map=terms_map_y2x,
    paulis_map=paulis_map,
)
protocol.update_k_and_beta(k=k, beta=beta)
protocol.dataframe_circuit_shot()

```

	Qubits	Depth	Shots	k	beta	TQ	CX	ZZPhase
0	8	332	10	50	0.5	326	19	307

Note that CX count is reduced significantly (from 218 to 19) from the original straightforward encoding by using the optimization tools of the iceberg code which exploits the stabilizers.

NOISE MITIGATION

Near-term quantum devices are inherently noisy: qubits can decohere and manipulation of qubits is imperfect. To combat the effects of noise a wide range of approaches are being developed, which include scalable error correction methods and various quantum error mitigation techniques. Many of these are general schemes and they are not specific to chemistry simulations, however there are also methods that are specifically designed for quantum chemistry algorithms (such as taking advantage of molecular symmetries) to calculate a more accurate final result.

General error mitigation methods can be applied to chemistry calculations in InQuanto by importing from the [Qermit package](#) [36]. This is a flexible open-source quantum error mitigation package developed by Quantinuum which can modify circuits and perform post-processing to improve results. Many schemes are available in Qermit, and they can be applied to InQuanto calculations through use of `run_mitres()` or `run_mitex()`.

Alongside Qermit support, InQuanto offers additional error mitigation schemes. In particular we highlight Partition Measurement Symmetry Verification (PMSV), which uses symmetries of the Hamiltonian to validate measurements. Another approach involves mitigating state preparation and measurement (SPAM) noise, which enhances the precision of the energy derived from quantum hardware in comparison to the precise value obtained through state-vector simulations on a classical computer. These InQuanto error mitigation methods are contained in the `protocols` module, and are applied at the point of `build()`. These classes can seamlessly integrate with Qermit workflows.

Below, we showcase the application of both Qermit and InQuanto error mitigation techniques to InQuanto's [PauliAveraging](#) protocol. To achieve this, we construct a simple 2-qubit `ansatz` and operator, and import the necessary dependencies:

```
from pytket import Circuit
from sympy import Symbol, pi

from inquanto.ansatzes import CircuitAnsatz
from inquanto.operators import QubitOperator

circ = Circuit(2)
circ.Ry(-2 * Symbol("a") / pi, 0)
circ.CX(0, 1)
circ.Rz(-2 * Symbol("b") / pi, 1)
circ.Rx(-2 * Symbol("c") / pi, 1)
circ.CX(1, 0)
circ.Ry(-2 * Symbol("d") / pi, 0)

ansatz = CircuitAnsatz(circ)
kernel = QubitOperator("X0 X1", 2) + QubitOperator("Y0 Y1", 2) + QubitOperator("Z0 Z1"
    ↪", 2)
parameters = ansatz.state_symbols.construct_from_array([0.1, 0.2, 0.3, 0.4])
```

We will calculate the expectation value of the operator `kernel` with the `ansatz` using [PauliAveraging](#), and qiskit's Aer-Backend with a basic noise model. The InQuanto `express` module offers a `get_noisy_backend()` utility function to quickly set up a simple noisy backend for demonstration purposes:

```
from inquanto.express import get_noisy_backend
from inquanto.protocols import PauliAveraging

noisy_backend = get_noisy_backend(n_qubits=2)
protocol = PauliAveraging(noisy_backend, shots_per_circuit=10000)
```

We are now ready to build the circuits and run them, applying the mitigation schemes.

8.1 Using Qermit

Qermit provides detailed [API documentation](#) and offers two types of error mitigation workflows: `MitRes` (mitigation of results) and `MitEx` (mitigation of expectation values). The `PauliAveraging` protocol supports both of these workflows. To begin, we will build the protocol without any InQuanto noise mitigation.

```
protocol.build(parameters, ansatz, kernel);
```

Typically, we can call the `run()` method of the protocol at this point. However, if we wish to use a `MitRes` or `MitEx` object, we need to call `run_mitres()` or `run_mitex()` respectively. For example, to use Qermit's State preparation and measurement scheme (SPAM), follow these steps:

```
from qermit.spam import gen_UnCorrelated_SPAM_MitRes

uc_spam_mitres = gen_UnCorrelated_SPAM_MitRes(
    backend=noisy_backend, calibration_shots=50
)
protocol.run_mitres(uc_spam_mitres, {})

energy_value = protocol.evaluate_expectation_value(ansatz, kernel)
print("MitRes (SPAM): ", energy_value)
```

```
MitRes (SPAM): 1.5564887022595482
```

Alternatively, we could use Qermit's zero-noise extrapolation (ZNE) method as such:

```
from qermit.zero_noise_extrapolation import gen_ZNE_MitEx

zne_mitex = gen_ZNE_MitEx(
    backend=noisy_backend, noise_scaling_list=[3]
)
protocol.run_mitex(zne_mitex, {})

energy_value = protocol.evaluate_expectation_value(ansatz, kernel)
print("MitEx (ZNE3): ", energy_value)
print("Exact: ", 1.5196420749021882)
```

```
MitEx (ZNE3): 1.498200000000000
Exact: 1.5196420749021882
```

In both cases, after running, we can call `evaluate_expectation_value()` to evaluate the final result.

Note that running a Qermit graph will often perform a significant amount of circuit preparation, evaluation, and post-processing, all of which must be completed synchronously.

8.2 Using InQuanto's PMSV and SPAM

An alternative to Qermit is using InQuanto's noise mitigation classes. These can be particularly useful for those who want to perform asynchronous jobs (i.e. launch/retrieve logic), enabling the separation of pre-measurement from post-measurement workflows.

A simple SPAM error mitigation can be implemented as follows:

```
from inquanto.protocols import SPAM

protocol = PauliAveraging(noisy_backend, shots_per_circuit=10000)

spam = SPAM(backend=noisy_backend).calibrate(
    calibration_shots=50, seed=0
)

protocol.build(parameters, ansatz, kernel).run(seed=0)
energy_value = protocol.evaluate_expectation_value(ansatz, kernel)
print("Raw: ", energy_value)

protocol.clear()
protocol.build(
    parameters, ansatz, kernel, noise_mitigation=spam
).run(seed=0)
energy_value = protocol.evaluate_expectation_value(ansatz, kernel)
print("NoiseMitigation (SPAM): ", energy_value)
```

```
Raw: 1.4407999999999999
NoiseMitigation (SPAM): 1.5000999800039994
```

In the next example, we use PMSV for a chemistry system (H_2). We begin by preparing the system:

```
from inquanto.express import get_system
from inquanto.ansatzes import FermionSpaceStateExpChemicallyAware

hamiltonian, space, state = get_system("h2_sto3g_symmetry.h5")
kernel = hamiltonian.qubit_encode()
exponents = space.construct_single_ucc_operators(state)
exponents += space.construct_double_ucc_operators(state)

ansatz = FermionSpaceStateExpChemicallyAware(exponents, state)
p = ansatz.state_symbols.construct_random(seed=1)

# We need a 4 qubit noisy backend
noisy_backend = get_noisy_backend(4)
```

PMSV prepares a set of expected Pauli string results and adds measurement of those Pauli strings to the circuit. If a shot does not have the expected symmetry for the set of stabilizers, it is discarded. Further details are given in the Appendix of the paper by Yamamoto et. al. [37].

```
from inquanto.protocols import PMSV

stabilizers = [
    -1 * QubitOperator("Z0 Z2"),
```

(continues on next page)

(continued from previous page)

```

-1 * QubitOperator("Z1 Z3"),
+1 * QubitOperator("Z2 Z3"),
]

protocol = PauliAveraging(noisy_backend, shots_per_circuit=10000)
pmsv = PMSV(stabilizers)
protocol.build(p, ansatz, kernel, noise_mitigation = pmsv).run(seed=0)
energy_value = protocol.evaluate_expectation_value(ansatz, kernel)
print("PMSV: ", energy_value)

```

PMSV: 0.4928363367103187

Lastly, the shot table for the Pauli strings can be examined.

```
print(protocol.dataframe_measurements())
```

	pauli_string	mean	stderr	umean	sample_size
0	Z0	0.838858	0.005620	0.839+-0.006	9383
1	Z3	-0.838858	0.005620	-0.839+-0.006	9383
2	Z2	-0.838858	0.005620	-0.839+-0.006	9383
3	Y0 Y1 X2 X3	-0.528168	0.008730	-0.528+-0.009	9461
4	X0 Y1 Y2 X3	0.507124	0.008855	0.507+-0.009	9475
5	Z1 Z3	-1.000000	0.000000	-1.0+-0	47232
6	X0 X1 Y2 Y3	-0.526716	0.008736	-0.527+-0.009	9470
7	Z2 Z3	1.000000	0.000000	1.0+-0	47232
8	Z0 Z2	-1.000000	0.000000	-1.0+-0	47232
9	Z1 Z2	-1.000000	0.000000	-1.0+-0	9383
10	Z1	0.838858	0.005620	0.839+-0.006	9383
11	Z0 Z1	1.000000	0.000000	1.0+-0	9383
12	Z0 Z3	-1.000000	0.000000	-1.0+-0	9383
13	Y0 X1 X2 Y3	0.520491	0.008787	0.520+-0.009	9443

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.

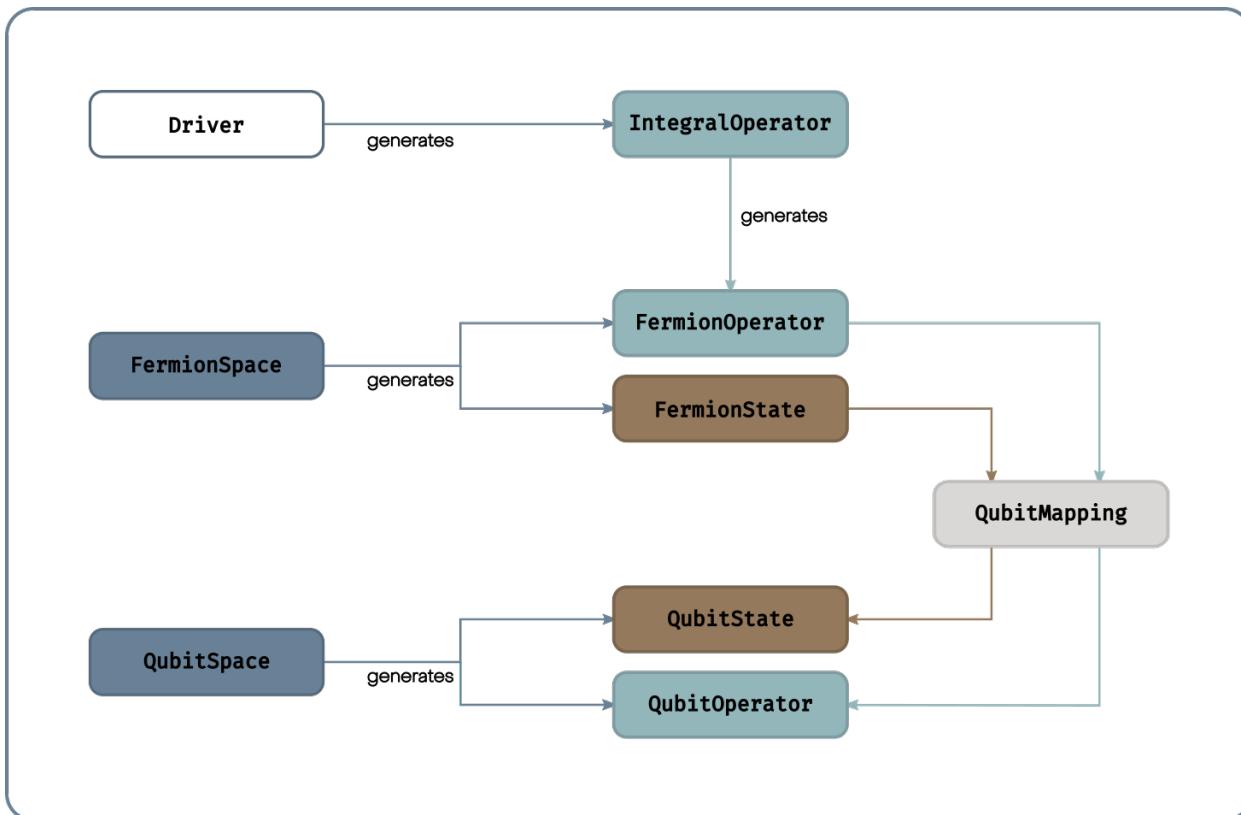


Fig. 9.1: A schematic of the vector space, state and operator classes in InQuanto. Spaces generate operators and states; fermion-to-qubit mappings transform fermionic states and operators to qubit states and operators.

We first discuss *fermionic* and *qubit* spaces and how operators and states can be *mapped* between the two. An option for parafermionic [10] spaces 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* - classes 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

$$|\boldsymbol{\eta}\rangle = |\eta_0, \eta_1, \dots, \eta_P, \dots, \eta_{Q-1}\rangle \quad (9.1)$$

where an occupation number $\eta_P = 1$ (0) if spin-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 spin-orbitals or modes. An arbitrary vector in this space corresponds to a linear combination of ON vectors

$$|\psi\rangle = \sum_{\boldsymbol{\eta}} \psi_{\boldsymbol{\eta}} |\boldsymbol{\eta}\rangle \quad (9.2)$$

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

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

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 spin-orbitals.

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

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, one- and two-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. The `FermionSpace` class represents a fermionic Hilbert space, maintaining consistent indexing of spin-orbitals. It incorporates the fermionic anti-commutation algebra in the logic of the class, allowing for the generation of various fermionic quantum operators acting on a given Hilbert space.

Various fermionic quantum operators can be constructed from the methods in this class. These include the number operators, spin operators, one- and two-body operators in various forms, and the anti-Hermitian operators needed for unitary coupled cluster ansatz generation. For example, the UCC terms for minimal basis H₂ (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 utilized 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_operators_z2()` 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 $\{\hat{A}, \hat{B}\} = \hat{A}\hat{B} + \hat{B}\hat{A}$. 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 spin-orbitals or modes that should be unoccupied

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

In InQuanto, vectors in $\mathcal{F}(Q, N)$ space are represented by `FermionState` objects. Individual basis states without coefficients are represented by `FermionStateString` objects. A `FermionStateString` functions as a dictionary mapping spin-orbital indices represented as integers, to occupation numbers (also represented as integers, of values 0 or 1). For example, for an ON vector of four spin-orbitals, with spin-orbitals indexed 0 and 1 occupied, and spin-orbitals indexed 3 and 4 unoccupied:

```

from inquanto.states import FermionStateString
on_vector = FermionStateString({0:1, 1:1, 3:0, 4:0})
print(on_vector)

```

```
{0: 1, 1: 1, 3: 0, 4: 0}
```

A simple list of occupation numbers may also be provided to the constructor, as a convenient alternative. In this case, a default register of fermionic modes will be assumed, indexed from $[0, N)$, where N is the length of the provided list (i.e. the number of spin-orbitals)

```
from inquanto.states import FermionStateString
on_vector = FermionStateString([1, 1, 0, 0])
print(on_vector)
```

```
{0: 1, 1: 1, 2: 0, 3: 0}
```

Note

The use of a dictionary with integer keys – as opposed to a simple `list` or `tuple` – allows for the representation of spin-orbitals with discontinuous indices. This may be useful when representing states which are elements of different Hilbert spaces. It also allows for a uniform interface with classes which represent states wherein the modes are represented by more complex objects, as in the case of `QubitState` below.

The `FermionState` then represents the state as a dictionary, with the `FermionStateString` objects (occupation configurations) as keys, and numerical quantities (configuration coefficients) as values. Typically, this will be constructed with a `dict` giving such a mapping:

```
from inquanto.states import FermionState, FermionStateString

on_vector = FermionStateString({0:1, 1:1, 2:0, 3:0})
state = FermionState({on_vector: 1.0})
print(state)
```

```
(1.0, {0: 1, 1: 1, 2: 0, 3: 0})
```

If a simple `FermionState` representing a single basis state with unit coefficient is required, a convenient alternative for construction is also provided. A list of integer occupation values may be directly passed to the constructor, following the logic of the `FermionStateString` above:

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

```
(1.0, {0: 1, 1: 1, 2: 0, 3: 0})
```

A `FermionState` object can also be generated as a method of `FermionSpace`, by passing the occupation list as an argument. This allows for consistent spin-orbital indexing determined by the `FermionSpace` instance.

```
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, {0: 1, 1: 1, 2: 0, 3: 0})
(1.0, {0: 0, 1: 0, 2: 1, 3: 1})
```

In addition to representing fermionic states, InQuanto has capabilities for representing operators acting within fermionic Hilbert spaces. Analogously to *FermionStateString*, the *FermionOperatorString* represents a single unweighted string of fermionic creation and annihilation operators. A *FermionOperatorString* may be created with 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. For example, consider an excitation operator in which an electron is annihilated from spin-orbital 2 and created on spin-orbital 3 ($\hat{f}_3^\dagger \hat{f}_2$):

```
from inquanto.operators import FermionOperator, FermionOperatorString

f_op_str = FermionOperatorString((3, 1), (2, 0))
print(f_op_str)
```

```
F3^ F2
```

For convenience, it is also possible to create such an object from string input:

```
f_op_str = FermionOperatorString.from_string("F3^ F2")
print(f_op_str)
```

```
F3^ F2
```

Similarly to *FermionState*, fermionic operators are then represented by *FermionOperator*. This stores the coefficients of each sequence of creation/annihilation operators as a dictionary, in which the keys are *FermionOperatorString* objects and the values are their coefficients. It may be created through providing the constructor with such a dictionary. Again, a convenient alternative is provided for the construction of an operator comprised of a single string:

```
f_op_string = FermionOperatorString((3, 1), (2, 0))
f_op1 = FermionOperator({f_op_string: 1.0})
f_op2 = FermionOperator(f_op_string, 1.0)
print(f_op1)
print(f_op2)
```

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

Additional construction methods are also available in the reference documentation for *FermionOperator*. Standard algebraic manipulation is possible for this class:

```
f_op_string = FermionOperatorString((3, 1), (2, 0))
f_op_string_conj = FermionOperatorString((2, 1), (3, 0))
f_op = FermionOperator({f_op_string: 1.0})
f_op_conj = FermionOperator({f_op_string_conj: 1.0})
f_op_sum = f_op + f_op_conj
print(f_op_sum)
```

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

Various other methods for manipulating `FermionOperator` instances are available in InQuanto. The following example (`fermion_operator`) summarizes some of the 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
op1 = FermionOperator({FermionOperatorString([(0, 0)]: 1.0})

# now multiply by its adjoint
op = op1 * op1.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})

# 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_
(continues on next page)
```

(continued from previous page)

```
 $\rightarrow Y_2), (0.875j, X_1 \cdot Y_2)$ 
 $\langle H_F | op_3 = (0)$ 
 $op_3 | H_F \rangle = (-3.5, \{0: 1, 1: 0, 2: 1, 3: 0\})$ 
```

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

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

```
a      [(1.0, F0)],
b      [(-3.5, F2^ F1 )]
```

Note

FermionOperatorList are designed to retain ordering for use in cases where operator ordering matters, such as exponentiation. In contrast, *FermionOperator* is not naturally ordered.

This class is particularly useful for containing Trotter sequences. The *FermionOperator* class contains various helper methods for generating such sequences.

```
op1 = FermionOperator({FermionOperatorString([(0, 0), (1, 1)]): 1.0})
op2 = FermionOperator({FermionOperatorString([(2, 0), (3, 1)]): 1.0})
op3 = op1 + op2
op_trotterized = op3.trotterize(trotter_number=2)
print(op_trotterized)
```

```
0.5      [(1.0, F0  F1^)],
0.5      [(1.0, F2  F3^)],
0.5      [(1.0, F0  F1^)],
0.5      [(1.0, F2  F3^)]
```

Additional information on the functionality of these classes is provided in the API reference.

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*

$$\hat{O} = \sum_i h_i P_i \quad (9.7)$$

where each Pauli string

$$P_i = \bigotimes_{n=0}^N p_n \quad (9.8)$$

is a tensor product of single qubit $p_n \in \{I, X, Y, Z\}$ 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 may also be constructed manually. Akin to the fermion operators discussed above, these comprise a map between [QubitOperatorString](#) Pauli strings and numerical or symbolic coefficients.

Whereas [FermionOperatorString](#) represents strings of creation and annihilation operators as a collection of pairs of integers (indexing modes) and 0 or 1 (annihilation/creation), pytket provides us a more sophisticated approach to identifying qubits. As such, a [QubitOperatorString](#) contains a map between pytket `qubit` objects and `Pauli` objects, indicating which Pauli operator (of $\{I, X, Y, Z\}$) is acting on which qubit. Note that this map is not necessarily exhaustive – while it may contain *explicit* identity operators, the operator is assumed to also *implicitly* act with the identity on any qubit not contained within the map.

```
from inquanto.operators import QubitOperatorString
from pytket import Qubit
from pytket.pauli import Pauli

pauli_string = QubitOperatorString({Qubit(0):Pauli.X, Qubit(1):Pauli.X})
print(pauli_string.map)
```

```
{q[0]: <Pauli.X: 1>, q[1]: <Pauli.X: 1>}
```

Much like [FermionOperator](#), a [QubitOperator](#) then stores a linear combination of Pauli strings as a map between these [QubitOperatorString](#) objects and numerical or symbolic coefficients.

```
from inquanto.operators import QubitOperator
pauli_string = QubitOperatorString({Qubit(0):Pauli.X, Qubit(1):Pauli.X})
op = QubitOperator({pauli_string:1.0})
print(op)
```

```
(1.0, x0 x1)
```

As with their fermionic counterparts, a variety of convenient alternative methods to construct the operator are provided, detailed in the API reference for [QubitOperator](#). 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: <Compressed Sparse Column sparse matrix of dtype 'complex128'>
    with 2 stored elements and shape (2, 2)
    Coords      Values
    (1, 0)      (1+0j)
    (0, 1)      (1+0j)
Identity: (1.0, )
```

Note

These examples have used an alternative string construction method for the `QubitOperator`, which is useful for simple examples such as this.

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

As mentioned above, a `QubitOperator` consists of a mapping between `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 an $(M \times 2N)$ 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 additionally provides a class for dealing with sets of Pauli operators that are not linearly combined, akin to the `FermionOperatorList` – the `QubitOperatorList`. This provides similar functionality to that of its fermionic counterpart.

```
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)]
```

Trotterization functionality is also available for the `QubitOperator`.

```
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)],
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, {q[0]: 1, q[1]: 1, q[2]: 0, q[3]: 0})
```

These can be converted to state vector representations:

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

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

Most importantly for the purposes of analyzing 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:

$$\{\hat{a}_i^\dagger, \hat{a}_j^\dagger\} = 0, \{\hat{a}_i, \hat{a}_j\} = 0, \{\hat{a}_i^\dagger, \hat{a}_j\} = \delta_{i,j} \quad (9.9)$$

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:

$$\begin{aligned} |1\rangle \langle 0|_i &= X_i - iY_i, \\ |0\rangle \langle 1|_i &= X_i + iY_i \\ \{X_i - iY_i, X_j + iY_j\} &\neq \delta_{i,j} \end{aligned} \quad (9.10)$$

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)])
```

(continues on next page)

(continued from previous page)

```
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]]
<Compressed Sparse Column sparse matrix of dtype 'int64'
 with 1 stored elements and shape (16, 1)>
Coords      Values
(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 [38]. 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:

$$\begin{aligned} a_i^\dagger &\rightarrow \frac{1}{2}(X_{U(i)}X_iZ_{P(i)} - iX_{U(i)}Y_iZ_{\rho(i)}) \\ a_i &\rightarrow \frac{1}{2}(X_{U(i)}X_iZ_{P(i)} + iX_{U(i)}Y_iZ_{\rho(i)}) \end{aligned} \quad (9.11)$$

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

$$\begin{aligned} a_i^\dagger &\rightarrow \frac{1}{2}(X_i - iY_i) \otimes Z^{\otimes i} \\ a_i &\rightarrow \frac{1}{2}(X_i + iY_i) \otimes Z^{\otimes i} \end{aligned} \quad (9.12)$$

Comparing this against (9.11) 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, since 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 \uparrow and \downarrow 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 ($\uparrow\uparrow\uparrow\uparrow$, $\downarrow\downarrow\downarrow\downarrow$, $\uparrow\uparrow\downarrow\downarrow$ and $\downarrow\downarrow\uparrow\uparrow$).

The `inquito-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:

```
import pandas as pd
from inquito.express import load_h5

pd.options.display.max_rows = 15
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.df())
```

	Coefficient	Term
0	-4.107196	\

(continues on next page)

(continued from previous page)

```

1      -0.396888          Z11
2      -0.396888          Z10
3       0.114044          Z10 Z11
4      -0.228984          Z9
...
626     -0.011161  Z0 X1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 X11
627      0.029058          Z0 Y1 Z2 Y3
628      0.034437          Z0 Y1 Z2 Z3 Z4 Y5
629     -0.011161  Z0 Y1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9 Z10 Y11
630      0.414556          Z0 Z1

          Coefficient Type
0    <class 'numpy.float64'>
1    <class 'numpy.float64'>
2    <class 'numpy.float64'>
3    <class 'numpy.float64'>
4    <class 'numpy.float64'>
...
626   <class 'numpy.float64'>
627   <class 'numpy.float64'>
628   <class 'numpy.float64'>
629   <class 'numpy.float64'>
630   <class 'numpy.float64'>

[631 rows x 3 columns]

```

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	-2.200996	F0^ F0
2	-1.093774	F0^ F2
3	0.530605	F0^ F4
4	-0.239791	F0^ F6
...
4460	-0.034129	F11^ F10^ F9 F10
4461	-0.034129	F11^ F10^ F9 F10
4462	0.193769	F11^ F10^ F10 F11
4463	0.193769	F11^ F10^ F10 F11
4464	-1.535127	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:

-1.5140974066187696

Effective potential a:

```
[[ 0.996 -0.      0.226]
 [-0.      0.924  0.      ]
 [ 0.226  0.      1.632]]
```

Effective potential b:

```
[[ 1.163 -0.      0.069]
 [-0.      1.541  -0.      ]
 [ 0.069  0.      1.712]]
```

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([[-0.316, -0.949],
       [-0.949,  0.316]])
```

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([[-0.267, -1.309],
       [-0.802,  1.068]])
```

One can achieve the same goal of orthonormalization using the `orthonormalize()` method, which finds the closest orthonormal set by the symmetric transformation.

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

```
array([[-0.887,  0.999],
       [ 1.336,  0.001]])
```

The `OrbitalTransformer` object also defines 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. 1.]
 [1. 0.]]
```

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 optimization 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 optimization returns only the optimized orbitals. This is for compatibility with the `transf` functionality in some extensions.

After the optimization, a report can be generated with the `generate_report()` method.

9.6 Double Factorization

An important restriction in near-term chemistry applications of quantum computing is the size of the Hamiltonian operator, particularly the two-body interaction term:

$$\hat{H}_2 = \frac{1}{2} \sum_{pqrs} h_{pqrs} a_p^\dagger a_r^\dagger a_s a_q, \quad (9.13)$$

which has $\mathcal{O}(N^4)$ terms where N is the number of orbitals. Double factorization is a two-step, tensor decomposition of the two-body integrals h_{pqrs} , which provides a systematic approach for truncating terms in two-body Hamiltonians. Below, we provide an introduction to the essential equations, and an example of using this decomposition strategy with InQuanto.

The first decomposition is given by:

$$h_{pqrs} = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t, \quad (9.14)$$

where $N_\gamma = N^2$ for an exact decomposition. And the second:

$$V_{pq}^t = \sum_u^{N_\lambda^t} U_{pu}^t \lambda_u^t U_{qu}^t. \quad (9.15)$$

for each t , where $N_\lambda^t = N$ for an exact decomposition.

In InQuanto, the first factorization can be done either by eigenvalue decomposition or a pivoted, incomplete Cholesky decomposition [39, 40]. The second factorization step is always an eigenvalue decomposition, since the fully factorized operator must be expressed in terms of unitary matrices to enable circuit-level rotations as shown below.

Each of the decompositions may be truncated to better control the size and scaling of number of terms in the Hamiltonian. The magnitudes of the factors γ^t and λ_u^t determine the importance of their corresponding terms in the decompositions and so terms can be systematically discarded based on their values.

The sums are truncated by discarding terms in ascending order of the factor magnitudes starting from the smallest such that the sum of the *discarded* factors does not exceed some threshold [41]:

$$\sum_{t=N_\gamma+1}^{N^2} |\gamma^t| < \varepsilon_1, \quad \sum_{u=N_\lambda^t+1}^N |\lambda_u^t| < \varepsilon_2. \quad (9.16)$$

In practice, the first decomposition can be truncated to $N_\gamma \sim \mathcal{O}(N)$, and the second decomposition to $N_\lambda^t < N$, helping to control the scaling of the number of terms in the two-body Hamiltonian.

Inserting the double factorized integrals into the two-body Hamiltonian (9.13), after some work we arrive at the diagonalized expression [41]:

$$\hat{H}_2 = \hat{S} + \frac{1}{2} \sum_t^{N_\gamma} \gamma^t \hat{R}(\mathbf{U}_t) \left(\sum_u^{N_\lambda^t} \lambda_u^t a_u^\dagger a_u \right)^2 \hat{R}(\mathbf{U}_t)^\dagger. \quad (9.17)$$

Here, \hat{S} is a one-body offset term which comes from rearranging the fermion operators:

$$\hat{S} = -\frac{1}{2} \sum_{pq} \left(\sum_r h_{prrq} \right) a_p^\dagger a_q, \quad (9.18)$$

and $\hat{R}(\mathbf{U}_t)$ are Fock-space basis rotation operators, corresponding to the single particle basis rotation matrices \mathbf{U}_t (the eigenvector matrices of the second decomposition (9.15)). These operators are given by the Thouless theorem [42]:

$$\hat{R}(\mathbf{A}) = \exp \left[\sum_{ij} [\ln \mathbf{A}]_{ij} a_i^\dagger a_j \right], \quad (9.19)$$

which are implemented at the circuit level with a Givens QR decomposition [43]. See below for an example, and [here](#) for more details.

We may similarly diagonalize the one-body integrals and \hat{S} operator:

$$\hat{H}'_1 = \hat{H}_1 + \hat{S} = \sum_{pq} h'_{pq} a_p^\dagger a_q = \hat{R}(\mathbf{W}) \left(\sum_r \omega_r a_r^\dagger a_r \right) \hat{R}(\mathbf{W})^\dagger \quad (9.20)$$

where all one-body-like terms have been consolidated into an effective one-body Hamiltonian H'_1 . The full Hamiltonian can then be written in this diagonal form as:

$$\begin{aligned} \hat{H} = & \hat{H}_0 + \hat{R}(\mathbf{W}) \left(\sum_r \omega_r a_r^\dagger a_r \right) \hat{R}(\mathbf{W})^\dagger \\ & + \frac{1}{2} \sum_t^{N_\gamma} \gamma^t \hat{R}(\mathbf{U}_t) \left(\sum_u^{N_\lambda^t} \lambda_u^t a_u^\dagger a_u \right)^2 \hat{R}(\mathbf{U}_t)^\dagger. \end{aligned} \quad (9.21)$$

In InQuanto, a `ChemistryRestrictedIntegralOperator` object may be transformed into this representation, a `DoubleFactorizedHamiltonian` object, using the `double_factorize()` method. We demonstrate this below for H₂O in the STO-3G basis:

```
import numpy as np
from inquanto.express import load_h5
from inquanto.operators import ChemistryRestrictedIntegralOperator

ham = load_h5("h2o_sto3g.h5", as_tuple=True).hamiltonian_operator
df_ham = ham.double_factorize(
    tol1=1e-3,
)

gammas = df_ham.two_body.outer_params
lambdas = df_ham.two_body.inner_params
N_gamma = len(gammas)
N_lambda = [len(l) for l in lambdas]

df_ham.n_orb, N_gamma, N_lambda
```

(7, 23, [7, 7, 4, 7, 2, 7, 4, 7, 2, 2, 7, 4, 4, 7, 2, 7, 4, 7, 2, 7, 2, 4, 7])

where we show the extent of the truncation (recall equation (9.16)). Default behavior is to consolidate and diagonalize all one-body terms, so `df_ham` represents a Hamiltonian of the form (9.21).

Note

`double_factorize()` performs the decomposition on the molecular orbital (MO) integrals. In classical literature, similar methods have been used to reduce the memory storage requirements of two-body atomic orbital (AO) integrals [44]. In InQuanto, the purpose of this approach is to truncate the Hamiltonian for quantum simulation, so it should not be expected to reduce classical memory requirements.

The `items()` method returns an iterator which generates the *FermionOperator* terms (the number operator sums in brackets of (9.21)) alongside their corresponding rotation matrices (\mathbf{W} and \mathbf{U}_t). For example, the one-body and the first two-body terms are:

```
terms = list(df_ham.items())
print("One body operator: ", terms[1][0])
print("One body rotation: ", terms[1][1])
print("\nFirst two body operator: ", terms[2][0])
print("First two body rotation: ", terms[2][1])
```

```
One body operator:  (-35.15627858132068, F0^ F0 ), (-35.15627858132068, F1^ F1 ), (-8.
-99140515493988, F2^ F2 ), (-8.99140515493988, F3^ F3 ), (-8.30673410272335, F4^ F4
), (-8.30673410272335, F5^ F5 ), (-8.057439359193435, F6^ F6 ), (-8.057439359193435,
F7^ F7 ), (-8.028943982634134, F8^ F8 ), (-8.028943982634134, F9^ F9 ), (-4.
621602713403975, F10^ F10 ), (-4.621602713403975, F11^ F11 ), (-4.593693566394931,
F12^ F12 ), (-4.593693566394931, F13^ F13 )
One body rotation:  [[-0.999 -0.031 -0. -0.011 -0. 0. 0.006]
 [ 0.028 -0.897 -0. -0.22 -0. -0. -0.382]
 [-0. 0. -0.775 0. 0. -0.632 0. 0.416]
 [ 0.011 0.046 -0. -0.908 -0. 0. 0.416]
 [ 0. -0. 0. -0. 1. 0. 0. ]
 [ 0.014 -0.438 -0. 0.355 0. 0. 0.825]
 [-0. 0. -0.632 0. 0. 0.775 -0. ]]

First two body operator:  (2.0462384054524048, F0^ F0 F0^ F0 ), (2.0462384054524048,
F0^ F0 F1^ F1 ), (0.816537144036992, F0^ F0 F2^ F2 ), (0.816537144036992, F0^ F0
F3^ F3 ), (0.8055041237052254, F0^ F0 F4^ F4 ), (0.8055041237052254, F0^ F0 F5^
F5 ), (0.7673891230355185, F0^ F0 F6^ F6 ), (0.7673891230355185, F0^ F0 F7^ F7 ),
(0.7479034551648778, F0^ F0 F8^ F8 ), (0.7479034551648778, F0^ F0 F9^ F9 ), (0.
41264550253745824, F0^ F0 F10^ F10 ), (0.41264550253745824, F0^ F0 F11^ F11 ), (0.
40355738951112213, F0^ F0 F12^ F12 ), (0.40355738951112213, F0^ F0 F13^ F13 ), (2.
0462384054524048, F1^ F1 F0^ F0 ), (2.0462384054524048, F1^ F1 F1^ F1 ), (0.
816537144036992, F1^ F1 F2^ F2 ), (0.816537144036992, F1^ F1 F3^ F3 ), (0.
8055041237052254, F1^ F1 F4^ F4 ), (0.8055041237052254, F1^ F1 F5^ F5 ), (0.
7673891230355185, F1^ F1 F6^ F6 ), (0.7673891230355185, F1^ F1 F7^ F7 ), (0.
7479034551648778, F1^ F1 F8^ F8 ), (0.7479034551648778, F1^ F1 F9^ F9 ), (0.
41264550253745824, F1^ F1 F10^ F10 ), (0.41264550253745824, F1^ F1 F11^ F11 ), (0.
40355738951112213, F1^ F1 F12^ F12 ), (0.40355738951112213, F1^ F1 F13^ F13 ), (0.
816537144036992, F2^ F2 F0^ F0 ), (0.816537144036992, F2^ F2 F1^ F1 ), (0.
3258334443413395, F2^ F2 F2^ F2 ), (0.3258334443413395, F2^ F2 F3^ F3 ), (0.
3214307946364968, F2^ F2 F4^ F4 ), (0.3214307946364968, F2^ F2 F5^ F5 ), (0.
30622126982800824, F2^ F2 F6^ F6 ), (0.30622126982800824, F2^ F2 F7^ F7 ), (0.
29844565015908275, F2^ F2 F8^ F8 ), (0.29844565015908275, F2^ F2 F9^ F9 ), (0.
16466330572421792, F2^ F2 F10^ F10 ), (0.16466330572421792, F2^ F2 F11^ F11 ), (0.
16103675769568104, F2^ F2 F12^ F12 ), (0.16103675769568104, F2^ F2 F13^ F13 ), (0.
```

(continues on next page)

(continued from previous page)

↳ 816537144036992, F3[^] F3 F0[^] F0), (0.816537144036992, F3[^] F3 F1[^] F1), (0.
 ↳ 3258334443413395, F3[^] F3 F2[^] F2), (0.3258334443413395, F3[^] F3 F3[^] F3), (0.
 ↳ 3214307946364968, F3[^] F3 F4[^] F4), (0.3214307946364968, F3[^] F3 F5[^] F5), (0.
 ↳ 30622126982800824, F3[^] F3 F6[^] F6), (0.30622126982800824, F3[^] F3 F7[^] F7), (0.
 ↳ 29844565015908275, F3[^] F3 F8[^] F8), (0.29844565015908275, F3[^] F3 F9[^] F9), (0.
 ↳ 16466330572421792, F3[^] F3 F10[^] F10), (0.16466330572421792, F3[^] F3 F11[^] F11), (0.
 ↳ 16103675769568104, F3[^] F3 F12[^] F12), (0.16103675769568104, F3[^] F3 F13[^] F13), (0.
 ↳ 8055041237052254, F4[^] F4 F0[^] F0), (0.8055041237052254, F4[^] F4 F1[^] F1), (0.
 ↳ 3214307946364968, F4[^] F4 F2[^] F2), (0.3214307946364968, F4[^] F4 F3[^] F3), (0.
 ↳ 3170876333750911, F4[^] F4 F4[^] F4), (0.3170876333750911, F4[^] F4 F5[^] F5), (0.
 ↳ 3020836191152334, F4[^] F4 F6[^] F6), (0.3020836191152334, F4[^] F4 F7[^] F7), (0.
 ↳ 2944130633377988, F4[^] F4 F8[^] F8), (0.2944130633377988, F4[^] F4 F9[^] F9), (0.
 ↳ 162438381098047, F4[^] F4 F10[^] F10), (0.162438381098047, F4[^] F4 F11[^] F11), (0.
 ↳ 15886083485519145, F4[^] F4 F12[^] F12), (0.15886083485519145, F4[^] F4 F13[^] F13), (0.
 ↳ 8055041237052254, F5[^] F5 F0[^] F0), (0.8055041237052254, F5[^] F5 F1[^] F1), (0.
 ↳ 3214307946364968, F5[^] F5 F2[^] F2), (0.3214307946364968, F5[^] F5 F3[^] F3), (0.
 ↳ 3170876333750911, F5[^] F5 F4[^] F4), (0.3170876333750911, F5[^] F5 F5[^] F5), (0.
 ↳ 3020836191152334, F5[^] F5 F6[^] F6), (0.3020836191152334, F5[^] F5 F7[^] F7), (0.
 ↳ 2944130633377988, F5[^] F5 F8[^] F8), (0.2944130633377988, F5[^] F5 F9[^] F9), (0.
 ↳ 162438381098047, F5[^] F5 F10[^] F10), (0.162438381098047, F5[^] F5 F11[^] F11), (0.
 ↳ 15886083485519145, F5[^] F5 F12[^] F12), (0.15886083485519145, F5[^] F5 F13[^] F13), (0.
 ↳ 7673891230355185, F6[^] F6 F0[^] F0), (0.7673891230355185, F6[^] F6 F1[^] F1), (0.
 ↳ 30622126982800824, F6[^] F6 F2[^] F2), (0.30622126982800824, F6[^] F6 F3[^] F3), (0.
 ↳ 3020836191152334, F6[^] F6 F4[^] F4), (0.3020836191152334, F6[^] F6 F5[^] F5), (0.
 ↳ 28778956771805136, F6[^] F6 F6[^] F6), (0.28778956771805136, F6[^] F6 F7[^] F7), (0.
 ↳ 28048196878961346, F6[^] F6 F8[^] F8), (0.28048196878961346, F6[^] F6 F9[^] F9), (0.
 ↳ 15475209021246006, F6[^] F6 F10[^] F10), (0.15475209021246006, F6[^] F6 F11[^] F11), (0.
 ↳ 15134382699799565, F6[^] F6 F12[^] F12), (0.15134382699799565, F6[^] F6 F13[^] F13), (0.
 ↳ 7673891230355185, F7[^] F7 F0[^] F0), (0.7673891230355185, F7[^] F7 F1[^] F1), (0.
 ↳ 30622126982800824, F7[^] F7 F2[^] F2), (0.30622126982800824, F7[^] F7 F3[^] F3), (0.
 ↳ 3020836191152334, F7[^] F7 F4[^] F4), (0.3020836191152334, F7[^] F7 F5[^] F5), (0.
 ↳ 28778956771805136, F7[^] F7 F6[^] F6), (0.28778956771805136, F7[^] F7 F7[^] F7), (0.
 ↳ 28048196878961346, F7[^] F7 F8[^] F8), (0.28048196878961346, F7[^] F7 F9[^] F9), (0.
 ↳ 15475209021246006, F7[^] F7 F10[^] F10), (0.15475209021246006, F7[^] F7 F11[^] F11), (0.
 ↳ 15134382699799565, F7[^] F7 F12[^] F12), (0.15134382699799565, F7[^] F7 F13[^] F13), (0.
 ↳ 7479034551648778, F8[^] F8 F0[^] F0), (0.7479034551648778, F8[^] F8 F1[^] F1), (0.
 ↳ 29844565015908275, F8[^] F8 F2[^] F2), (0.29844565015908275, F8[^] F8 F3[^] F3), (0.
 ↳ 2944130633377988, F8[^] F8 F4[^] F4), (0.2944130633377988, F8[^] F8 F5[^] F5), (0.
 ↳ 28048196878961346, F8[^] F8 F6[^] F6), (0.28048196878961346, F8[^] F8 F7[^] F7), (0.
 ↳ 2733599255869177, F8[^] F8 F8[^] F8), (0.2733599255869177, F8[^] F8 F9[^] F9), (0.
 ↳ 15082260028140734, F8[^] F8 F10[^] F10), (0.15082260028140734, F8[^] F8 F11[^] F11), (0.
 ↳ 147500880233922, F8[^] F8 F12[^] F12), (0.147500880233922, F8[^] F8 F13[^] F13), (0.
 ↳ 7479034551648778, F9[^] F9 F0[^] F0), (0.7479034551648778, F9[^] F9 F1[^] F1), (0.
 ↳ 29844565015908275, F9[^] F9 F2[^] F2), (0.29844565015908275, F9[^] F9 F3[^] F3), (0.
 ↳ 2944130633377988, F9[^] F9 F4[^] F4), (0.2944130633377988, F9[^] F9 F5[^] F5), (0.
 ↳ 28048196878961346, F9[^] F9 F6[^] F6), (0.28048196878961346, F9[^] F9 F7[^] F7), (0.
 ↳ 2733599255869177, F9[^] F9 F8[^] F8), (0.2733599255869177, F9[^] F9 F9[^] F9), (0.
 ↳ 15082260028140734, F9[^] F9 F10[^] F10), (0.15082260028140734, F9[^] F9 F11[^] F11), (0.
 ↳ 147500880233922, F9[^] F9 F12[^] F12), (0.147500880233922, F9[^] F9 F13[^] F13), (0.
 ↳ 41264550253745824, F10[^] F10 F0[^] F0), (0.41264550253745824, F10[^] F10 F1[^] F1), (0.
 ↳ 16466330572421792, F10[^] F10 F2[^] F2), (0.16466330572421792, F10[^] F10 F3[^] F3), (0.
 ↳ 162438381098047, F10[^] F10 F4[^] F4), (0.162438381098047, F10[^] F10 F5[^] F5), (0.

(continues on next page)

(continued from previous page)

```

→15475209021246006, F10^ F10 F6^ F6 ), (0.15475209021246006, F10^ F10 F7^ F7 ), (0.
→15082260028140734, F10^ F10 F8^ F8 ), (0.15082260028140734, F10^ F10 F9^ F9 ), (0.
→08321430695009605, F10^ F10 F10^ F10 ), (0.08321430695009605, F10^ F10 F11^ F11 ), (0.
→ (0.08138159334405826, F10^ F10 F12^ F12 ), (0.08138159334405826, F10^ F10 F13^ F13 ), (0.41264550253745824, F11^ F11 F0^ F0 ), (0.41264550253745824, F11^ F11 F1^ F1 ), (0.16466330572421792, F11^ F11 F2^ F2 ), (0.16466330572421792, F11^ F11 F3^ F3 ), (0.162438381098047, F11^ F11 F4^ F4 ), (0.162438381098047, F11^ F11 F5^ F5 ), (0.15475209021246006, F11^ F11 F6^ F6 ), (0.15475209021246006, F11^ F11 F7^ F7 ), (0.15082260028140734, F11^ F11 F8^ F8 ), (0.15082260028140734, F11^ F11 F9^ F9 ), (0.08321430695009605, F11^ F11 F10^ F10 ), (0.08321430695009605, F11^ F11 F11^ F11 ), (0.08138159334405826, F11^ F11 F12^ F12 ), (0.08138159334405826, F11^ F11 F13^ F13 ), (0.40355738951112213, F12^ F12 F0^ F0 ), (0.40355738951112213, F12^ F12 F1^ F1 ), (0.16103675769568104, F12^ F12 F2^ F2 ), (0.16103675769568104, F12^ F12 F3^ F3 ), (0.15886083485519145, F12^ F12 F4^ F4 ), (0.15886083485519145, F12^ F12 F5^ F5 ), (0.15134382699799565, F12^ F12 F6^ F6 ), (0.15134382699799565, F12^ F12 F7^ F7 ), (0.147500880233922, F12^ F12 F8^ F8 ), (0.147500880233922, F12^ F12 F9^ F9 ), (0.08138159334405826, F12^ F12 F10^ F10 ), (0.08138159334405826, F12^ F12 F11^ F11 ), (0.07958924346013584, F12^ F12 F12^ F12 ), (0.07958924346013584, F12^ F12 F13^ F13 ), (0.40355738951112213, F13^ F13 F0^ F0 ), (0.40355738951112213, F13^ F13 F1^ F1 ), (0.16103675769568104, F13^ F13 F2^ F2 ), (0.16103675769568104, F13^ F13 F3^ F3 ), (0.15886083485519145, F13^ F13 F4^ F4 ), (0.15886083485519145, F13^ F13 F5^ F5 ), (0.15134382699799565, F13^ F13 F6^ F6 ), (0.15134382699799565, F13^ F13 F7^ F7 ), (0.147500880233922, F13^ F13 F8^ F8 ), (0.147500880233922, F13^ F13 F9^ F9 ), (0.08138159334405826, F13^ F13 F10^ F10 ), (0.08138159334405826, F13^ F13 F11^ F11 ), (0.07958924346013584, F13^ F13 F12^ F12 ), (0.07958924346013584, F13^ F13 F13^ F13 )
First two body rotation: [[-0.991 -0.09 -0. -0. -0.094 0. -0.01 ]
 [ 0.109 -0.641 -0. -0. -0.587 -0. 0.483]
 [-0. 0. -0.694 0. 0. -0.72 -0. ]
 [ 0.044 0.399 0. -0. -0.798 0. -0.449]
 [ 0. -0. -0. -1. 0. 0. -0. ]
 [ 0.057 -0.65 -0. 0. 0.101 0. -0.751]
 [-0. 0. -0.72 0. -0. 0.694 0. ]]

```

Note

The corresponding rotation matrix for the constant energy term is the identity.

Circuit representations of the rotation operators may be computed using the `inquanto.ansatzes.restricted_basis_rotation_to_circuit()` method, which uses the `RealRestrictedBasisRotationAnsatz` to encode the rotations in the Jordan-Wigner picture (see [here](#) for more information). For example, below we construct a computable for the expectation value of a single term in (9.21) with a UCCSD ansatz:

```

from inquanto.mappings import QubitMappingJordanWigner
from inquanto.ansatzes import FermionSpaceAnsatzUCCSD, CircuitAnsatz, ComposedAnsatz, restricted_basis_rotation_to_circuit
from inquanto.states import FermionState
from inquanto.computables import ExpectationValue

jw = QubitMappingJordanWigner()

```

(continues on next page)

(continued from previous page)

```

operator, rotation = terms[4]  # Selecting the 4th term as an example

ansatz = FermionSpaceAnsatzUCCSD( # |psi>
    fermion_space=7*2,
    fermion_state=FermionState([1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0]),
    qubit_mapping=jw
)
ansatz_rotator = CircuitAnsatz(restricted_basis_rotation_to_circuit(rotation.T))  # ↳
    ↳ R(U) ^
rotated_ansatz = ComposedAnsatz(ansatz_rotator, ansatz)  # R(U) ^ |psi>
term_computable = ExpectationValue(rotated_ansatz, jw.operator_map(operator))  # ↳
    ↳ <psi| R(U) X R(U) ^ |psi>

```

Note the transpose of the rotation matrix: $\hat{R}(\mathbf{U}_t)^\dagger = \hat{R}(\mathbf{U}_t^T)$, which is true for a real-valued rotation matrix.

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-generalized *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 essentially perform Configuration Interaction at the quantum circuit level: prepare a circuit in a specified multi-configurational state 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.

10.1 Trotter based

10.1.1 The UCC Family

The following unitary coupled cluster (UCC) ansatzes are available in InQuanto:

- *FermionSpaceAnsatzUCCSD*
- *FermionSpaceAnsatzUCCD*
- *FermionSpaceAnsatzUCCGSD*
- *FermionSpaceAnsatzUCCGD*
- *FermionSpaceAnsatzkUpCCGSD*

These are discussed in further detail in the following subsections.

Unitary Coupled Cluster

This ansatz corresponds to a variant of the (non-unitary) coupled cluster method [45], in which the operator acting on a reference state becomes unitary [46]

$$\begin{aligned}\hat{U}_{\text{UCC}}(\boldsymbol{\theta}) &= e^{\hat{T}-\hat{T}^\dagger} = e^{\sum_{\lambda} \theta_{\lambda} (\hat{T}_{\lambda} - \hat{T}_{\lambda}^\dagger)} \\ |\Psi(\boldsymbol{\theta})\rangle &= \hat{U}_{\text{UCC}}(\boldsymbol{\theta})|\text{HF}\rangle.\end{aligned}\quad (10.1)$$

Here, the excitation operator \hat{T} has the same form as in coupled cluster theory. Under the exponent the excitation operator self-adjoint is subtracted, rendering the exponent anti-Hermitian, and hence the operator \hat{U}_{UCC} is unitary. The unitarity of \hat{U}_{UCC} makes its implementation on a quantum circuit more natural. By parameterizing the excitations, this ansatz is suitable for *variational algorithms* in which the parameters $\boldsymbol{\theta} = \{\theta_1, \theta_2, \dots, \theta_{\lambda}, \dots\}$ 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}_{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 H₂ 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'],
    report_uccd['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: [d0, s0, s1]

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*). Symbols corresponding to the ansatz 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. A pytket *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 *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 minimal basis H₂. 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. 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 is expressed in a trotterized form (facilitated by the *FermionOperatorList* internal structure), i.e. approximated as

$$e^{\hat{T} - \hat{T}^\dagger} \approx \prod_{\lambda} e^{\theta_{\lambda} (\hat{T}_{\lambda} - \hat{T}_{\lambda}^\dagger)} \quad (10.2)$$

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 trotterization. To implement higher-order approximations, one needs to use their base *FermionSpaceStateExp* class described in the corresponding *section*.

Generalized 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. [27] 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 generalized transitions is referred

to as UCCG(S)D. Here

$$\begin{aligned}\hat{T}^{(1G)} &= \sum_{p,q} t_p^q \hat{f}_q^\dagger \hat{f}_p \\ \hat{T}^{(2G)} &= \sum_{p,q,r,s} t_{p,q}^{r,s} \hat{f}_r^\dagger \hat{f}_s^\dagger \hat{f}_p \hat{f}_q \\ \hat{T}_{\text{GSD}} &= \hat{T}^{(1G)} + \hat{T}^{(2G)} \\ \hat{U}_{\text{UCCGSD}}(\boldsymbol{\theta}) &= e^{\hat{T}_{\text{GSD}} - \hat{T}_{\text{GSD}}^\dagger},\end{aligned}\tag{10.3}$$

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. $\boldsymbol{\theta} = \{\theta_1, \theta_2, \dots, \theta_\lambda, \dots\} \mapsto \{t_p^q, \dots, t_{p,q}^{r,s}, \dots\}$. 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 generalized 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)

$$\hat{T}^{(2pG)} = \sum_{(p\alpha, p\beta), (q\alpha, q\beta)} t_{p\alpha, p\beta}^{q\alpha, q\beta} \hat{f}_{q\alpha}^\dagger \hat{f}_{q\beta}^\dagger \hat{f}_{p\alpha} \hat{f}_{p\beta}\tag{10.4}$$

where α, β label spins, p, q labels spatial orbitals, and the singles are as in $T^{(1G)}$. This ansatz, referred to as k -UpCCGSD [27] (`FermionSpaceAnsatzkUpCCGSD`), is expressed as a product of k factors of cluster operators, each one with an independent set of parameters

$$\begin{aligned}\hat{T}^{(k)} &= \hat{T}^{(1G)} + \hat{T}^{(2pG)} \\ \hat{U}_{k\text{UpCCGSD}}(\boldsymbol{\theta}) &= \prod_k e^{\hat{T}^{(k)} - \hat{T}^{(k)\dagger}}.\end{aligned}\tag{10.5}$$

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
)

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

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

```
UCCGSD parameters: [gd0, gs0, gs1]
kUpCCGSD parameters: [gd0k0, gd0k1, gs0k0, gs0k1, gs1k0, gs1k1]
```

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.

10.1.2 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. This also comes with an overhead of 2 CX-gates per spatial orbital for all spatial orbitals used across all spatial-to-spatial excitations.
- Encoding spin-orbitals to qubits with Jordan–Wigner mapping.
- Exploiting commuting sets of excitations in the Ansatz circuit synthesis.

For more details on the method please refer to [47].

The ansatz can be used similarly to other ansatzes in the UCC family. Once instances of *FermionSpace* and *FermionState* (the occupation state corresponding to the reference) 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 `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. The H₂ molecule in the 6-31g basis set can be used to exemplify the benefit of a symmetry aware *FermionSpace*:

```
from inquanto.symmetry import PointGroup
```

(continues on next page)

(continued from previous page)

```

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.

10.1.3 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:

$$|\text{TrotterAnsatz}\rangle = \prod_k^{\leftarrow} e^{ip_k \sum_{l_k} \lambda_{l_k}^{(k)} \hat{P}_{l_k}^{(k)}} |\text{Ref}\rangle = \prod_k^{\leftarrow} \prod_{l_k} e^{ip_k \lambda_{l_k}^{(k)} \hat{P}_{l_k}^{(k)}} |\text{Ref}\rangle, \quad (10.6)$$

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:

$$\prod_k^{\leftarrow} \hat{O}_k = \dots \hat{O}_{k+1} \hat{O}_k \hat{O}_{k-1} \dots \hat{O}_0. \quad (10.7)$$

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

$$\text{PauliExpBox}(\hat{P}, t) = e^{-\frac{i\pi}{2} t \hat{P}}. \quad (10.8)$$

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

$$|\text{TrotterAnsatz}\rangle = \prod_k^{\leftarrow} \prod_{l_k} \text{PauliExpBox}(\hat{P}_{l_k}^{(k)}, -\frac{2}{\pi} p_k \lambda_{l_k}^{(k)}) |\text{Ref}\rangle. \quad (10.9)$$

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

$$\begin{aligned} \text{QubitOperatorList} = & [\dots, (p_{k-1}, \{ \dots, (i\lambda_{l_{k-1}}^{(k-1)}, \hat{P}_{l_{k-1}}^{(k-1)}), \dots \}), \\ & (p_k, \{ \dots, (i\lambda_{l_k}^{(k)}, \hat{P}_{l_k}^{(k)}), \dots \}), (p_{k+1}, \{ \dots, (i\lambda_{l_{k+1}}^{(k+1)}, \hat{P}_{l_{k+1}}^{(k+1)}), \dots \}), \dots]. \end{aligned} \quad (10.10)$$

One should keep in mind that only the 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 the InQuanto ansatz class hierarchy, `TrotterAnsatz` is a base class of `FermionSpaceStateExp`, which in turn forms the basis of all `UCC` ansatz classes. However, it can be used all by itself, in order to define a custom ansatz in terms of `QubitOperator` objects. To do this, one needs to provide bare-bones Pauli string terms, “wrapped” into `QubitOperator` objects, and then construct a `QubitOperatorList` object out of them. As described above, one should think of it as of a product of exponents of the provided Pauli strings, with each string multiplied by a symbolic expression. The 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]))
```

10.1.4 Fermionic Exponentiated Ansatz

The `FermionSpaceStateExp` class is essentially a thin wrapper over `TrotterAnsatz` (see previous section), and serves as the basis for all the `Unitary Coupled Cluster` ansatzes. It accepts a `FermionOperatorList` object as an input, maps it to a corresponding `QubitOperatorList` object, and provides the latter as an input to `TrotterAnsatz`.

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))
    ]
)
```

(continues on next page)

(continued from previous page)

```
)
print(f"Fermion state: {state}")
print(f"Fermion operator list: \n{term_list}")
```

```
Fermion state: (1.0, {0: 1, 1: 1, 2: 0, 3: 0, 4: 0, 5: 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` constructor, together with the `FermionState` object and the fermion-to-qubit mapping class of choice:

```
from inquanto.ansatzen 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())
)
```

The ansatz object thus constructed can generate a circuit and be used with any of the algorithms or computable objects of InQuanto.

10.2 Basic ansatz

10.2.1 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 pytket `Circuit` object to a `CircuitAnsatz` constructor.

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

```
from pytket import circuit
from sympy import Symbol
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)

# Add custom block, of redefined Rz
theta = Symbol('theta')
```

(continues on next page)

(continued from previous page)

```

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)

```

10.2.2 Composed Ansatz

In this section we show to merge two InQuanto ansatzes into a single ansatz object. This can be done 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 generalized UCC calculation for a system of two electrons in four spin-orbitals, one could consider employing [MultiConfigurationAnsatz](#) 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.mappings import QubitMappingJordanWigner
from inquanto.ansatzes import (
    MultiConfigurationAnsatz,
    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)})

qubit_mapping = QubitMappingJordanWigner()
fstate_multiconf = qubit_mapping.state_map(fstate_multiconf)

ansatz_givens = MultiConfigurationAnsatz(fstate_multiconf.terms)

```

(continues on next page)

(continued from previous page)

```
ansatz_uccgd = FermionSpaceAnsatzUCCGD(space, FermionState([0, 0, 0, 0]))
ansatz_multiref = ComposedAnsatz(ansatz_uccgd, ansatz_givens)
print(ansatz_multiref.state_symbols)

[gd0, theta0, theta1]
```

The Givens rotations of `MultiConfigurationAnsatz` will be applied to the circuit first, then the unitaries of the `FermionSpaceAnsatzUCCGD` part will be appended at the end. In this particular case the resulting multireference ansatz will be identical to a UCCGSD ansatz constructed from the HF reference state $|1100\rangle$ and in the same Fock space as above. 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.

10.3 Other Ansatz

10.3.1 Hardware Efficient Ansatz

The hardware efficient ansatz (HEA, `HardwareEfficientAnsatz`) [48] 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:

$$\hat{U}_{\text{HEA}}(\boldsymbol{\theta}) = \left(\prod_i^{N_q} \hat{U}_{R_x}(\theta_{i,N_L}) \right) \hat{U}_{\text{Ent}} \left(\prod_i^{N_q} \hat{U}_{R_x}(\theta_{i,N_{L-1}}) \right) \hat{U}_{\text{Ent}} \dots \times \dots \times \left(\prod_i^{N_q} \hat{U}_{R_x}(\theta_{i,l}) \right) \hat{U}_{\text{Ent}} \dots \left(\prod_i^{N_q} \hat{U}_{R_x}(\theta_{i,1}) \right) \hat{U}_{\text{Ent}} \left(\prod_i^{N_q} \hat{U}_{R_x}(\theta_{i,\text{cap}}) \right) \quad (10.11)$$

where N_q is the number of qubits, N_L is the number of layers, and index l labels the layers (indexing is in reverse order). 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 pytket.circuit import OpType

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

(continues on next page)

(continued from previous page)

```

)
report_heo = ansatz.generate_report()

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

heo_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.

10.3.2 Multiconfiguration States Using Givens Rotations

In InQuanto, Givens rotations can be used 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 [QubitState](#). The resulting object can be used as i) an ansatz itself, or ii) a multiconfigurational generalization 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 [49]. 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

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

which corresponds to a rotation between the $|10\rangle$ and $|01\rangle$ basis states. This has a straight-forward generalization to n qubits [49]. 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 generalization of G_1 , with appropriate lexicographical ordering)

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

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 normalized to unity. In InQuanto, the gate decompositions

previously reported in the literature for particle-number preserving forms of the G_1 and G_2 [49, 50] are used in the Ansatz classes `MultiConfigurationState` and `MultiConfigurationAnsatz`.

The following example shows how to use InQuanto to prepare a circuit in a quantum state corresponding to an equal mixing of 2 occupation configurations:

```
from inquanto.ansatzes import MultiConfigurationState
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 = MultiConfigurationState(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, all rotations must end up in the desired Fock space sector. This conserves spin and particle numbers, and prevents unwanted configurations from entering the full expansion of the rotated state vector. One way to accomplish this is to apply Givens rotations in a sequence corresponding to the order of input configurations (i.e. the order of `QubitStateString` objects used to instantiate `QubitState`). A Givens rotation at the j^{th} configuration in the sequence is then controlled on certain qubits only if necessary; the necessity depending on whether it can also rotate any of the $i < j$ configurations. In InQuanto, this is implemented in a general way that guarantees the state of the circuit will be the user-defined linear combination of configurations (with user-defined coefficients for `MultiConfigurationState`). However, the size of the circuits can increase significantly when these extra controls are present. Consider adding a third configuration $|1001\rangle$ to the example above, such that the second Givens rotation to mix $|1100\rangle$ and $|1001\rangle$ can also act on the basis state $|0011\rangle$. This more than doubles the circuit size since that single excitation (second Givens rotation) needs to be controlled on a qubit of the first configuration:

```
from inquanto.ansatzes import MultiConfigurationState
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 = MultiConfigurationState(qubit_states)
```

(continues on next page)

(continued from previous page)

```
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: 51
Number of gates: 76
```

However, adding a configuration with an associated Givens rotation that can not act on previous configurations (in the sense that the two Givens rotations act on disjoint qubit subspaces) leads to only a linear (with respect to the number of configurations) increase in circuit size. By keeping this in mind, the same state vector as above can be obtained with less gates by changing the order so that $|1001\rangle$ is “before” $|0011\rangle$, since no extra controls of the Givens rotations are needed in this in order:

```
from inquanto.ansatzes import MultiConfigurationState
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([1, 0, 0, 1]), 1/sqrt(3)
config3, c_3 = QubitStateString([0, 0, 1, 1]), 1/sqrt(3)
qubit_states = QubitState({config1: c_1, config2: c_2, config3: c_3,})

multi_state = MultiConfigurationState(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: 25
Number of gates: 40
```

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 G_1 rotation [49] is utilized. This multicontrolled “SWAP+rotation ladder” can represent a general chemical excitation to any order. However, again due to the use of extra controls, a large increase in circuit depth is observed when comparing a 2-body mixing to e.g. 3-body mixing. 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 MultiConfigurationState
from inquanto.states import QubitState, QubitStateString

from pytket.extensions.qiskit import AerStateBackend

from math import sqrt

backend = AerStateBackend()
```

(continues on next page)

(continued from previous page)

```

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 = MultiConfigurationState(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 = MultiConfigurationState(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: 321
Number of gates, configs separated by 3-body excitation: 347

```

How much the circuit size increases due to the presence of (multi)controls 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 [51]. Currently, pytket utilizes a linear depth scaling approach for decomposing multicontrols [52] during circuit compilation for intermediate ($6 \leq n \leq 50$) numbers of control qubits. However, the user should in general be cautious of large circuits when mixing of a large number of basis configurations and/or mixing configurations that are separated by ($N > 2$)-body rotations.

Symbolic Multiconfiguration Ansatz

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. For this we use the InQuanto ansatz class `MultiConfigurationAnsatz`. An instance of this class can 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 MultiConfigurationAnsatz
from inquanto.states import QubitStateString

qss_ref = QubitStateString([1, 1, 0, 0])
qss_2 = QubitStateString([0, 0, 1, 1])
qss_list = [qss_ref, qss_2]
ansatz = MultiConfigurationAnsatz(qss_list)
# |psi> = a|1100> + b|0011>, b = SQRT(1 - |a|^2) => only 1 parameter in VQE

```

(continues on next page)

(continued from previous page)

```
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 optimization of their coefficients.

Note that `MultiConfigurationAnsatz` takes a list of `QubitStateString` objects (representing Slater determinants) as its input argument. This differs from `MultiConfigurationState` which needs a `QubitState` object containing occupations states and their coefficients (so that gate angles are calculated for a given set of coefficients during instantiation of `MultiConfigurationState`). On the other hand, `MultiConfigurationAnsatz` only needs occupation states to construct the variational ansatz (in this case the rotation angles of the object returned by `MultiConfigurationAnsatz` are symbolic). However, the symbols representing these coefficients are related by normalization, hence in this example there is only 1 variational parameter.

Multiconfiguration states without Givens rotations

In addition to `MultiConfigurationState` which uses Givens rotations, a user can also prepare arbitrary multiconfigurational states with fixed coefficients by utilizing pytket's `StatePreparationBox` via InQuanto. This can be done using the InQuanto Ansatz class `MultiConfigurationStateBox`, which serves as a wrapper of `StatePreparationBox`. The API is similar to `MultiConfigurationState`. The major differences between `MultiConfigurationState` and `MultiConfigurationStateBox` is that the latter does not use Given's rotations specifically, but rather multiplexed R_y and R_z gates which prepare a circuit from a statevector provided in the computational basis (see [53] for more details on the circuit synthesis method). This means that, unlike in `MultiConfigurationState`, for `MultiConfigurationStateBox` the input `QubitState` (which specifies the chemical occupations) has to be converted to a computational statevector in `numpy.ndarray` format, *before* the circuit is built. This is akin to first solving an exponentially scaling classical problem before preparing the state circuit. Hence, `MultiConfigurationStateBox` is considered a tool for exploring and debugging state preparation schemes, rather than a scalable state preparation method. An example showing its usage is given below.

```
from inquanto.ansatzes import MultiConfigurationStateBox
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 = MultiConfigurationStateBox(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: 25
Number of gates: 28
```

Note that in `MultiConfigurationStateBox` coefficients must be fixed and non-symbolic, hence the use of this class to variationally optimize the configuration coefficients is currently not supported.

10.3.3 Real Basis Rotation Ansatzes

An orbital basis transformation can be represented on a quantum circuit with Givens rotation gates [43]. This circuit representation allows us to define the set of classes:

- `RealGeneralizedBasisRotationAnsatz`
- `RealRestrictedBasisRotationAnsatz`
- `RealUnrestrictedBasisRotationAnsatz`

These classes apply a unitary transformation to a reference state, in the form:

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

where the relationship between a real basis transformation matrix \mathbf{R} and the variational parameters is $\theta_{ij} = [\ln \mathbf{R}]_{ij}$, according to the Thouless theorem [42]. This ansatz can be used in variational 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 RealGeneralizedBasisRotationAnsatz
from inquanto.express import load_h5, run_vqe
from inquanto.states import QubitState

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

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

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): -1.1175058842043306
<REF|H|REF>: -0.12440620192256882
# TIMER BLOCK-0 BEGINS AT 2024-10-30 10:11:10.741639
```

```
# TIMER BLOCK-0 ENDS - DURATION (s): 7.2835802 [0:00:07.283580]
VQE energy: -1.1175058838338097
```

Given a real unitary matrix \mathbf{R} , 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:
{phi_0: 0.0, phi_1: 0.0, phi_2: 0.0, phi_3: 3.141592653589793, theta_1_0: -1.
 ↪5263485631292077, theta_2_0: -1.717901058621577, theta_2_1: -0.31117806797812514,
 ↪theta_3_0: 0.027836442642734206, theta_3_1: -1.0105744336026965, theta_3_2: 0.
 ↪862400917785053}

```

The `RealGeneralizedBasisRotationAnsatz` supports *generalized* real rotations, which may couple spin channels. For restricted rotations i.e. both α and β spins are rotated by the same matrix independently of one another, one should use the `RealRestrictedBasisRotationAnsatz`. For unrestricted rotations, where α and β spins are rotated by *different* matrices, one should use the `RealUnrestrictedBasisRotationAnsatz`.

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

10.3.4 Parameters

As discussed in previous sections, 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:

```

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

```

```
{d1, d0}
```

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

```
from sympy import Symbol
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:

```
from inquanto.ansatzes import RealRestrictedBasisRotationAnsatz
my_ansatz = RealRestrictedBasisRotationAnsatz(2)
random_param = my_ansatz.state_symbols.construct_random()
print(random_param)
```

```
{phi_0: 0.9417154046806644, phi_1: -1.3965781047011498, theta_1_0: -0.
↪6797144480784211}
```

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}
```

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. Class method `is_symmetry_of()` can verify (by commutation) that a given symmetry operator is indeed a symmetry of a given operator within the same space. Class method `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, long execution times, or both.

For computational purposes, it is common to express fermionic symmetry operators as products of operators, rather than linear combinations. For example, the fermionic parity operator $(-1)^{\sum_i \hat{N}_i}$ may be expressed as a product $\prod_i 1 - 2\hat{N}_i$. The product form can be efficiently stored, whereas expanding out the product into a single linear combination may invoke computational difficulty. It is particularly useful when converting to a qubit representation, where individual multiplicands may be converted to single Pauli strings, thus avoiding any exponential growth in the number of terms.

In order to allow for symmetry operators in this form to be used in InQuanto, a `inquanto.operators.SymmetryOperatorFermionicFactorised` class is provided. These are a subclass of `FermionOperatorList`,

and provide similar `is_symmetry_of()` and `symmetry_sector()` functionality to the main symmetry classes. Conversion is also possible with the `to_symmetry_operator_fermionic()` method. In all cases, care should again be taken to ensure excessive computational difficulty is not incurred.

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 electron number parity and spin parity only.

⚠ 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`, as discussed above.

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. [54]

```
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, Z0 Z1 I2 I3)
(1.0, Z0 I1 Z2 I3)
(1.0, Z0 I1 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 by using the `inquanto.symmetry.PointGroup` class. For example, 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 [54] 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)
```

```
(-0.29264467227722657, I0), (0.8248612119271037, Z0), (0.17966867956301552, X0)
```

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 behavior can be changed using the `tapering_exponent_check_behaviour` parameter, which is described in the API reference documentation for `FermionSpaceStateExp`, the base class of `FermionSpaceAnsatzUCCSD`.

 **Tip**

Note that in the example above, `taperer` and `tapering_exponent_check_behaviour` are passed as `**kwargs` to `FermionSpaceAnsatzUCCSD` as it doesn't have them as positional parameters. These are forwarded to its parent `FermionSpaceStateExp` class which handles the tapering.

CHAPTER TWELVE

GEOMETRY

The first step of any quantum chemistry calculation is specification of the system geometry. The InQuanto `GeometryMolecular` and `GeometryPeriodic` classes exist to standardize geometry formats and provide convenient functionality for building and manipulating molecular and periodic structures.

12.1 Molecular Systems

12.1.1 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 `dataframe` 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. See below:

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

	Element	X	Y	Z
id				
0	H	0.281779	-0.436955	0.412635

 **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", [0.0, 0.0, 0.127161],],
    ["H", [0.0, 0.758082, -0.508642],],
    ["H", [0.0, -0.758082, -0.508642]],
]
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 using the `load_csv()` and `save_csv()` methods.
- .json files using the `load_json()` and `save_json()` methods.
- Z-matrix files in the Gaussian format using the `load_zmatrix()` and `save_zmatrix()` methods.

12.1.2 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]],
    ["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.

12.1.3 Calculating and Modifying Properties By Group

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
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	3.349405e-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`.

12.1.4 Generating Many Structures

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])
```

```
[100.0, 101.0, 103.0, 103.99999999999999, 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).

12.2 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
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
```

(continues on next page)

(continued from previous page)

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.

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.ansatze import FermionSpaceAnsatzUCCSD
from inquanto.computables import ExpectationValue, ExpectationValueDerivative
from inquanto.core import dict_to_vector
from inquanto.protocols import SparseStatevectorProtocol
from pytket.extensions.qiskit import AerStateBackend
import numpy as np

ansatz = FermionSpaceAnsatzUCCSD(space, state)
parameters = ansatz.state_symbols.construct_zeros()
sv_protocol = SparseStatevectorProtocol(AerStateBackend())
```

(continues on next page)

(continued from previous page)

```

ev = ExpectationValue(kernel=qubit_hamiltonian, state=ansatz)
evg = ExpectationValueDerivative(ansatz, qubit_hamiltonian, ansatz.free_symbols_
↪ordered())

def vqe_objective(variables):
    parameters = ansatz.state_symbols.construct_from_array(variables)
    sv_evaluator = sv_protocol.get_evaluator(parameters)
    return ev.evaluate(sv_evaluator).real

def vqe_gradient(variables):
    parameters = ansatz.state_symbols.construct_from_array(variables)
    sv_evaluator = sv_protocol.get_evaluator(parameters)
    gradient_dict = evg.evaluate(sv_evaluator)
    return dict_to_vector(ansatz.free_symbols_ordered(), gradient_dict)

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]: -1.1175058842043306
Gradients of parameters at [0, 0, 0]: [0.359 0. 0.]
```

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
```

```
(-1.1368465754720538, array([-0.107, -0. , -0. ]))
```

```

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
```

(continues on next page)

(continued from previous page)

```

Function evaluations: 5
Gradient evaluations: 5
Objective function minimum is -1.1368465754720538, located at [-0.107 0.      0.      ]

```

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 [55], 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(protocol_objective=SparseStatevectorProtocol(AerStateBackend()))
vqe.run()
print("VQE Energy:", vqe.generate_report()["final_value"])

```

```

# TIMER BLOCK-0 BEGINS AT 2024-10-30 10:12:38.343578

ROTORsolver - A gradient-free optimizer for parametric circuits

```

```

Iteration 1
fun = -1.1368465754720538          variance = 0.011499023526666223      p-norm =_
˓→0.10723350002059162

```

```

Iteration 2
fun = -1.1368465754720538          variance = 0.0          p-norm = 0.
˓→10723350002059162

Optimizer Converged
nit = 2          nfun = 19
final fun = -1.1368465754720538      final variance = 0.0      final p-norm =_
˓→0.10723350002059162

# TIMER BLOCK-0 ENDS - DURATION (s): 0.7162587 [0:00:00.716259]
VQE Energy: -1.1368465754720538

```

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 [56].

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(
    protocol_objective=SparseStatevectorProtocol(AerStateBackend()),
    protocol_gradient=SparseStatevectorProtocol(AerStateBackend()),
)
vqe.run()
print("VQE Energy:", vqe.generate_report()["final_value"])
```

```
# TIMER BLOCK-1 BEGINS AT 2024-10-30 10:12:39.067260
```

```
Optimizer Stochastic Gradient Descent
```

```
Iteration 0
```

```
fun = -1.1175058842043306
p-norm = 0.0
g-norm = 0.35933735912603115
```

```
Iteration 1
```

```
fun = -1.1321535146012704
p-norm = 0.054487281372526786
g-norm = 0.17778365523875433
```

```
Iteration 2
```

```
fun = -1.1357239648371469
p-norm = 0.08144509079704805
g-norm = 0.08704389270781199
```

```
Iteration 3
```

```
fun = -1.136578976183475
p-norm = 0.09464378821405427
g-norm = 0.0425085425788404
```

(continues on next page)

(continued from previous page)

```
Iteration 4
```

```
fun = -1.136782840579148
p-norm = 0.10108947180749593
g-norm = 0.020746679121849215
```

```
Iteration 5
```

```
fun = -1.136831398578587
p-norm = 0.10423534605115131
g-norm = 0.010124128478002536
```

```
Iteration 6
```

```
fun = -1.1368429616408458
p-norm = 0.10577049463234589
g-norm = 0.004940280683155496
```

```
Iteration 7
```

```
fun = -1.136845714977897
p-norm = 0.10651960255782586
g-norm = 0.0024106935679380492
```

```
Iteration 8
```

```
fun = -1.1368463705791978
p-norm = 0.10688514244785748
g-norm = 0.0011763364084182715
```

```
Iteration 9
```

```
fun = -1.1368465266849062
p-norm = 0.10706351347231796
g-norm = 0.0005740118592297838
```

```
Optimizer Converged
```

```
nit = 9           nfun = 9           njac = 9
```

```
final fun = -1.1368465266849062
final p-norm = 0.10715055242023334
final g-norm = 0.0005740118592297838
```

```
# TIMER BLOCK-1 ENDS - DURATION (s): 0.9145364 [0:00:00.914536]
VQE Energy: -1.1368465266849062
```

13.4 Minimizer SPSA

Simultaneous Perturbation Stochastic Approximation (SPSA) [57] is available in the `MinimizerSPSA` class. This minimizer is especially efficient in high dimensional problems. SPSA approximates function gradients using only two objective function measurements and is robust to noisy measurements of this objective function.

A short example using `MinimizerSPSA` is shown below.

```
from inquanto.minimizers import MinimizerSPSA

minimizer=MinimizerSPSA(max_iterations=5, disp=True)

vqe = AlgorithmVQE(
    objective_expression=ev,
    minimizer=minimizer,
    initial_parameters=ansatz.state_symbols.construct_zeros()
)
vqe.build(protocol_objective=SparseStatevectorProtocol(AerStateBackend()))
vqe.run()

print("VQE Energy:", vqe.generate_report()["final_value"])
```

```
# TIMER BLOCK-2 BEGINS AT 2024-10-30 10:12:39.988636
Starting SPSA minimization.
```

```
Result at iteration 0: [0. 0. 0.]
```

```
Result at iteration 1: [-0.063 -0.063 0. ].
```

```
Result at iteration 2: [-0.063 0.021 0.084].
```

```
Result at iteration 3: [-0.153 -0.068 -0.005].
```

```
Result at iteration 4: [-0.126 -0.068 0.022].
Finishing SPSA minimization.
# TIMER BLOCK-2 ENDS - DURATION (s): 0.9211855 [0:00:00.921186]
VQE Energy: -1.1324899392974357
```

CHAPTER
FOURTEEN

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 [58, 59]. 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 i) impurity DMET, which divides the total system into a single fragment and its environment; ii) one-shot DMET, which only optimizes the chemical potential; and iii) full DMET, which utilizes a chemical potential and an arbitrary parameterization of the one-body correlation potential.

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 the 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 *inquito-pyscf* extension has a number of fragment solvers for the user to combine.

14.1 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 basis of the new orbitals becomes diagonal. The Schmidt decomposition and DMET guarantee that the number of fractional diagonal elements in the environment block will be bounded by 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 (fermionic) form of:

$$\begin{aligned}\hat{H}_{\text{emb}} = & \sum_{pq \in A \cup B} (h_{pq} + \sum_{rs \in E} ((pq|rs) - (ps|rq))\Gamma_{rs}^{\text{env}}) \hat{a}_p^\dagger \hat{a}_q \\ & + \frac{1}{2} \sum_{pqrs \in A \cup B} (pq|rs)\hat{a}_p^\dagger \hat{a}_r^\dagger \hat{a}_s \hat{a}_q\end{aligned}\tag{14.1}$$

where A is the set of fragment orbitals, 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 fermionic creation and annihilation operators in the fragment basis, respectively, and Γ_{rs}^{env} is the 1-RDM constructed from the fully occupied environment orbitals only.

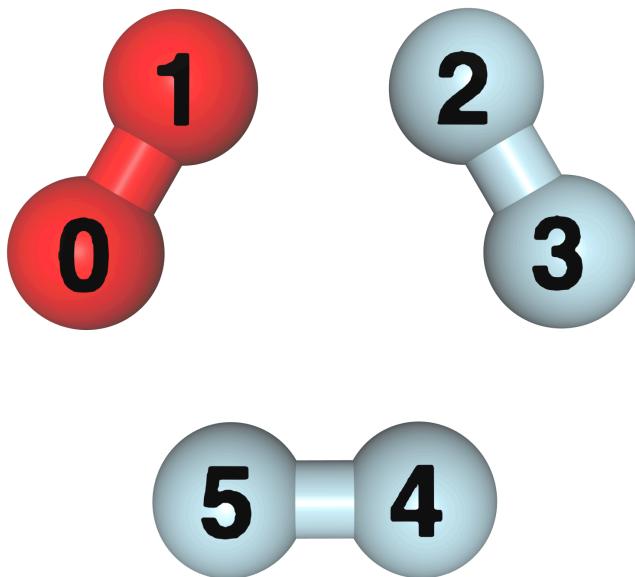


Fig. 14.1: Three H_2 molecules in a ring arrangement, the H_2 molecule in red color is selected as the fragment.

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. 14.1. The Hamiltonian and the Hartree-Fock (HF) 1-RDM are computed with the STO3G basis set and with the Löwdin transformation. This results 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 diagonalization 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.

14.2 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

$$\hat{H}_{\text{emb}}(\mu) = \hat{H}_{\text{emb}} - \mu \sum_{p \in A} \hat{a}_p^\dagger \hat{a}_p \quad (14.2)$$

where μ is the introduced global chemical potential. The value of μ is determined by the global constraint

$$\sum_f \langle \Psi_f(\mu) | \hat{N}_f | \Psi_f(\mu) \rangle = N_{\text{tot}}, \quad (14.3)$$

where $\Psi_f(\mu)$ is the ground state of fragment f , calculated by a high-level method, $\hat{N}_f = \sum_{p \in A_f} \hat{a}_p^\dagger \hat{a}_p$ 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 [60] 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:

$$\begin{aligned} \hat{E}_{\text{emb}} = & \sum_{p \in A} \left(\sum_{q \in A \cup B} \left(h_{pq} + \frac{\sum_{rs \in E} [(pq|rs) - (ps|rq)] \Gamma_{rs}^{\text{env}}}{2} \right) \hat{a}_p^\dagger \hat{a}_q \right. \\ & \left. + \frac{1}{2} \sum_{qrs \in A \cup B} (pq|rs) \hat{a}_p^\dagger \hat{a}_r^\dagger \hat{a}_s \hat{a}_q \right). \end{aligned} \quad (14.4)$$

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 $\hat{E}_{\text{emb},f}$ 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.one_body_rdm_hf_
    ↪lowdin
)

```

The fragments are similarly defined as in the impurity DMET case. In this particular example, we decompose the total system into three H₂ fragments and use a black-box statevector (i.e. ideal classical simulation) fragment solver *DMETRHF-FragmentUCCSDVQE* which utilizes the *UCCSD* ansatz and optimises the ansatz parameters using *VQE*:

```

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)
```

14.3 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:

$$\hat{H}_{\text{emb}}(\mu, C) = \hat{H}_{\text{emb}}(\mu) + V_{\text{emb}} \left(\sum_{p,q \notin A} C_{pq} \hat{a}_p^\dagger \hat{a}_q \right) \quad (14.5)$$

By computing the ground state of this Hamiltonian with a high-level method, an accurate 1-RDM, Γ_{pq}^A , is obtained for the fragment. In turn, the full DMET method finds the effective one-body correlation for the fragment, that is, $C_{pq} \hat{a}_p^\dagger \hat{a}_q$ for $p, q \in A$ such that a mean-field (MF) solution 1-RDM, $\Gamma_{pq}^{A,\text{MF}}$, of

$$\hat{H}_{\text{emb}}(\mu, C)' = \hat{H}_{\text{emb}}(\mu) + V_{\text{emb}} \left(\sum_{p,q \notin A} C_{pq} \hat{a}_p^\dagger \hat{a}_q \right) + \sum_{p,q \in A} C_{pq} \hat{a}_p^\dagger \hat{a}_q \quad (14.6)$$

minimally differs from Γ_{pq}^A , that is, $|\Gamma_{pq}^A - \Gamma_{pq}^{A,\text{MF}}|$ 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 one-shot DMET is that we also need to input a parameterization for the correlation potential C_{pq} . The [DMETRHF](#) class can take this parameterized 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:

```
import numpy

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 parameterization; 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)
```

14.4 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 simulation. 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 [DMETRHF-FragmentActive](#), 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 of the ground state:

```
class MyFragment(DMETRHFFragmentActive):

    def solve_active(
        self,
        hamiltonian_operator: ChemistryRestrictedIntegralOperator,
        fragment_energy_operator: ChemistryRestrictedIntegralOperator,
        fermion_space: FermionSpace,
        fermion_state: FermionState,
    ):
```

(continues on next page)

(continued from previous page)

```
# ... 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 H₂ 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/embeddings` folder.

14.5 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
FIFTEEN

EXPRESS

The `express` module contains a database of pre-computed, example data sets (*see below*) generated with classical compute methods as well as helpful methods and model hamiltonians for use in testing and benchmarking.

15.1 Express methods

Each example 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', 'beh2_purvis_A_purvis_cas44_symmetry.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', 'beh2_purvis_A_purvis_symmetry.h5', 'h3_chain_sto3g.h5', 'beh2_
↳ purvis_A_purvis_cas22_symmetry.h5', 'beh2_purvis_E_purvis_cas44_symmetry.h5', 'h2_5_
↳ pbc_sto3g.h5', 'heh_sto3g_u.h5', 'h2_631g.h5', 'beh2_sto3g_symmetry.h5', 'hehp_
↳ 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',
↳ 'beh2_purvis_E_purvis_symmetry.h5', 'h2o_sto3g_symmetry.h5', 'h2_3_ring_sto3g.h5',
↳ 'h3p_chain_sto3g.h5', 'ch4_sto3g_symmetry.h5', 'beh2_purvis_E_purvis_cas22_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 with bond length 0.7122Å, calculated with the sto3g basis set in a spin-
↳ restricted formalism.

Hamiltonian operator:

(continues on next page)

(continued from previous page)

```
(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.
-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 )
```

H2 STO-3G CCSD Energy:
-1.1368465754747636 Ha

A list of the available data sets is given *below*, or can be queried with the `list_h5()` function.

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: -1.117505884204331

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 2024-10-30 10:16:20.297995

TIMER BLOCK-0 ENDS - DURATION (s): 0.3149335 [0:00:00.314934]
VQE Energy: -1.13684658

When preparing to process circuits on noisy quantum hardware it can be useful to first employ simple noise models in simulations. The `get_noisy_backend()` function returns a shot-based AerBackend simulator class which includes only CNOT depolarizing errors and readout error.

```
from inquanto.express import get_noisy_backend

noisy_aer = get_noisy_backend(n_qubits=4, cx_err=0.01, ro_err=0.001)
type(noisy_aer)
```

```
pytket.extensions.qiskit.backends.aer.AerBackend
```

This can be used as a direct replacement when AerBackend is used in examples and tutorials to study the effect of noise.

15.2 Model hamiltonians

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 )
```

15.3 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_purvis_A_purvis_cas22_symmetry.h5	BeH ₂ molecule geometry A from G. D. Purvis III, R. Shepard, F. B. Brown and R. J. Bartlett, Int. J. Quantum Chem., 1983, 23, 835–845.
beh2_purvis_A_purvis_cas44_symmetry.h5	BeH ₂ molecule geometry A from G. D. Purvis III, R. Shepard, F. B. Brown and R. J. Bartlett, Int. J. Quantum Chem., 1983, 23, 835–845.
beh2_purvis_A_purvis_symmetry.h5	BeH ₂ molecule geometry A from G. D. Purvis III, R. Shepard, F. B. Brown and R. J. Bartlett, Int. J. Quantum Chem., 1983, 23, 835–845.
beh2_purvis_E_purvis_cas22_symmetry.h5	BeH ₂ molecule geometry E from G. D. Purvis III, R. Shepard, F. B. Brown and R. J. Bartlett, Int. J. Quantum Chem., 1983, 23, 835–845.
beh2_purvis_E_purvis_cas44_symmetry.h5	BeH ₂ molecule geometry E from G. D. Purvis III, R. Shepard, F. B. Brown and R. J. Bartlett, Int. J. Quantum Chem., 1983, 23, 835–845.
beh2_purvis_E_purvis_symmetry.h5	BeH ₂ molecule geometry E from G. D. Purvis III, R. Shepard, F. B. Brown and R. J. Bartlett, Int. J. Quantum Chem., 1983, 23, 835–845.
beh2_sto3g.h5	BeH ₂ molecule, calculated with the sto3g basis set, with a spin-restricted formalism.
beh2_sto3g_symmetry.h5	BeH ₂ molecule, calculated with point group symmetries in the sto3g basis, with a spin-restricted formalism.
ch2_sto3g_m3.h5	CH ₂ molecule with spin multiplicity 3. Calculated with the sto3g basis set in a spin-restricted formalism.
ch2_sto3g_m3_u.h5	CH ₂ molecule with spin multiplicity 3. Calculated with the sto3g basis set in a spin-unrestricted formalism.

continues on next page

Table 15.1 – continued from previous page

File name	Description
ch4_sto3g.h5	CH4 molecule, calculated with the sto3g basis set in a spin-restricted formalism.
ch4_sto3g_symmetry.h5	CH4 molecule, calculated with point group symmetries in the sto3g basis, with a spin-restricted formalism.
h2_1_pbc_631g.h5	H2 molecule in a tetragonal unit cell with periodic boundary conditions. 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 conditions. Calculated with the sto3g basis set in a spin-restricted formalism.
h2_1_ring_631g.h5	A single H2 molecule, calculated with the 631g basis in a spin-restricted formalism.
h2_1_ring_sto3g.h5	A single H2 molecule, calculated with the sto3g 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	2 H2 molecules in a ring geometry, calculated with the 631g basis in a spin-restricted formalism.
h2_2_ring_sto3g.h5	2 H2 molecules in a ring geometry, calculated with the sto3g 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	3 H2 molecules in a ring geometry, calculated with the sto3g 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	4 H2 molecules in a ring geometry, calculated with the sto3g 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	5 H2 molecules in a ring geometry, calculated with the sto3g basis in a spin-restricted formalism.
h2_631g.h5	H2 molecule with bond length 0.73Å, calculated with the 631g basis set in a spin-restricted formalism.
h2_631g_symmetry.h5	H2 molecule, calculated with point group symmetries in the 631g basis, with a spin-restricted formalism.
h2_sto3g.h5	H2 molecule with bond length 0.7122Å, calculated with the sto3g basis set in a spin-restricted formalism.
h2_sto3g_long.h5	H2 molecule with a longer bond length of 0.735Å compared to h2_sto3g.h5.
h2_sto3g_symmetry.h5	H2 molecule, calculated with point group symmetries in the sto3g basis, with a spin-restricted formalism.

continues on next page

Table 15.1 – continued from previous page

File name	Description
h2o_sto3g.h5	H2O molecule, calculated with the sto3g basis set in a spin-restricted formalism.
h2o_sto3g_symmetry.h5	H2O molecule, calculated with point group symmetries in the sto3g basis, with a spin-restricted formalism.
h3_chain_sto3g.h5	Linear chain of three H atoms with separation 0.9Å, calculated with the sto3g basis in a spin-restricted formalism.
h3_sto3g_m2.h5	Linear chain of three H atoms with spin multiplicity 2, calculated with the sto3g basis in a spin-restricted formalism.
h3_sto3g_m2_u.h5	Linear chain of three H atoms with spin multiplicity 2, calculated with the sto3g basis in a spin-unrestricted formalism.
h3p_chain_sto3g.h5	Linear H3+ chain with separation 0.9Å, calculated with the sto3g basis in a spin-restricted formalism.
h3p_sto3g_c2v.h5	H3+ chain with C2v symmetry, calculated with point group symmetries in the sto3g 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 sto3g basis in a spin-restricted formalism
h5_sto3g_m2.h5	H5 molecule with spin multiplicity 2. Calculated with the sto3g basis set in a spin-restricted formalism.
h5_sto3g_m2_u.h5	H5 molecule with spin multiplicity 2. Calculated with the sto3g basis set in a spin-unrestricted formalism.
heh_sto3g_u.h5	HeH molecule with bond length 0.772Å and spin multiplicity 2. Calculated with the sto3g basis set in a spin-unrestricted formalism.
hehp_sto3g.h5	HeH+ molecule with bond length 0.772Å. Calculated with the sto3g basis set in the spin-restricted formalism.
hehp_sto3g_symmetry.h5	HeH+ molecule with bond length 0.772Å. Calculated with point group symmetries in the sto3g basis set in the spin-restricted formalism.
lih_631g.h5	LiH molecule with bond length 1.64Å, calculated with the 631g basis in a spin-restricted formalism.
lih_sto3g.h5	LiH molecule with bond length 1.511Å, calculated with the sto3g basis in a spin-restricted formalism.
lih_sto3g_symmetry.h5	LiH molecule with bond length 1.511Å, calculated with point group symmetries in the sto3g basis, with a spin-restricted formalism.
nh3_sto3g.h5	NH3 molecule, calculated in the sto3g basis set with a spin-restricted formalism.

continues on next page

Table 15.1 – continued from previous page

File name	Description
nh3_sto3g_symmetry.h5	NH3 molecule, calculated with point group symmetries in the sto3g basis, with a spin-restricted formalism.

CHAPTER
SIXTEEN

TUTORIALS

These tutorial notebooks contain more guided examples of using the functionality in InQuanto and its extensions to simulate quantum chemistry using quantum computers. The notebooks are available to download (right-click and ‘Save as’ on download link) or viewable in browser. InQuanto also includes many other *examples* of specific functionality, mostly in the form of scripts that can be adapted. Examples using the inquanto-extensions can be found [here](#). If there isn’t a downloadable tutorial for your use case then take a look at the relevant manual pages as they contain extensive example code-blocks.

CHAPTER
SEVENTEEN

CORE TUTORIALS

These tutorials cover some of the main methods, demonstrating how a user can put together their own calculation and perform analysis using the modules in InQuanto.

VQE	A basic VQE simulation	download
Extended VQE	Extended VQE	download
VQD	Variational Quantum Deflation for excited states	download
NGLView	Visualization with inquanto-nglview	download

17.1 A basic VQE simulation

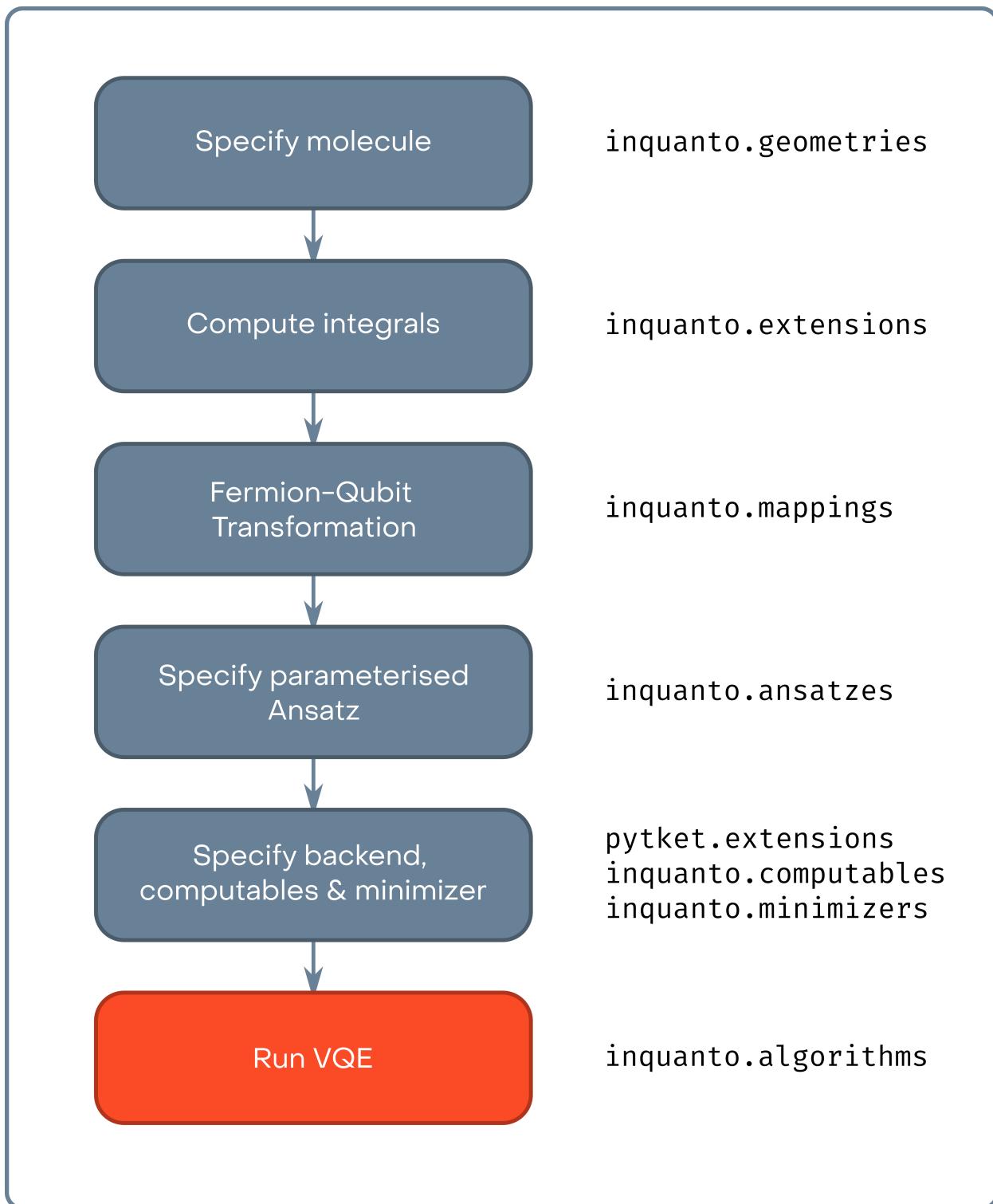
This file will introduce the basic methodology of performing quantum chemistry calculations on quantum computers using InQuanto. We focus here on a practical guide for computation using InQuanto. A discussion of the theory of [VQE](#) is available in the [InQuanto docs](#). For this example, we will 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:

$$\hat{H} = \sum_{i,j=0}^N h_{ij} \hat{a}_i^\dagger \hat{a}_j + \frac{1}{2} \sum_{i,j,k,l=0}^N h_{ijkl} \hat{a}_i^\dagger \hat{a}_j^\dagger \hat{a}_k \hat{a}_l \quad (17.1)$$

where N is the number of fermionic spin-orbitals, \hat{a}_i^\dagger and \hat{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, `inquito.extensions` streamlines this process.

Here we will utilize `inquito-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 `inquito.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 `inquito.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 inquito.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: {}'.format(hartree_fock_energy))

/lib/python3.10/site-packages/pyscf/dft/libxc.py:771: UserWarning: Since PySCF-2.3, B3LYP (and B3P86) are changed to the VWN-RPA variant, corresponding to the original definition by Stephens et al. (issue 1480) and the same as the B3LYP functional in Gaussian. To restore the VWN5 definition, you can put the setting "B3LYP_WITH_VWN5 = True" in pyscf_conf.py
  warnings.warn('Since PySCF-2.3, B3LYP (and B3P86) are changed to the VWN-RPA variant, '
```

HARTREE FOCK ENERGY: -1.1175058842043306

The `inquito.extensions.pyscf.ChemistryDriverPySCFMolecularRHF.get_system` method uses PySCF to run the restricted Hartree-Fock calculation, returning the integrals in the electronic Hamiltonian as a `inquito.operators.ChemistryRestrictedIntegralOperator`, a `inquito.spaces.FermionSpace` object describing the fermionic Fock space, and the Hartree-Fock fermionic reference state.

In the cell below, we show how the `inquito.operators.ChemistryRestrictedIntegralOperator`, which internally stores the h_{ij} and h_{ijkl} integrals, can be converted to a `inquito.operators.FermionOperator` that represents the electronic Hamiltonian as a sum of terms. These terms can be viewed in a data frame table or printed. The `inquito.operators.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.
-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
```

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')
```

(continues on next page)

(continued from previous page)

```

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

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: {}'.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, ), (-1.270292724390438, 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.
    ↪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 )

HARTREE FOCK ENERGY: -1.1175058842043306

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 `inquito.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 `inquito.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 inquito.mappings import QubitMappingJordanWigner

jw_map = QubitMappingJordanWigner()
```

(continues on next page)

(continued from previous page)

```

qubit_hamiltonian = jw_map.operator_map(fermionic_hamiltonian)
print('QUBIT HAMILTONIAN:\n{}'.format(qubit_hamiltonian))

QUBIT HAMILTONIAN:
(-0.05962058276034754, ), (0.17575942918319665, Z0), (-0.23667117678035543, Z2), (0.
↔17575942918319665, Z1), (-0.23667117678035543, Z3), (0.12222714936261832, Z0 Z2), (0.
↔(0.12222714936261832, Z1 Z3), (0.17001546439603182, Z0 Z1), (0.1671443192533722, Z1
↔Z2), (0.044917169890753894, X0 X1 X2 Y3), (-0.044917169890753894, X0 X1 Y2 Y3), (-0.
↔044917169890753894, Y0 Y1 X2 X3), (0.044917169890753894, X0 Y1 Y2 X3), (0.
↔1671443192533722, Z0 Z3), (0.1757033833190701, Z2 Z3)

```

Here, we use the `inquanto.mappings.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. The `inquanto.mappings` module contains several fermion-qubit mappings, which all utilize the `.operator_map()` method to map fermionic Fermion-Operators 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 parameterized 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]; S q[3]; Sdg q[0]; S q[2]; H q[3]; Vdg q[0]; CX q[1], q[2]; S
↳ q[3]; Sdg q[2]; V 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];
↳ S q[0]; S q[1]; Sdg q[2]; H q[0]; H q[1]; Vdg q[2]; S q[0]; S q[1]; S q[2]; V q[0];
↳ V q[1]; H q[2]; S q[0]; Sdg q[1]; S q[2]; H q[0]; Vdg q[1]; V q[2]; S q[0]; CX q[2],
↳ q[3]; V 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]; S q[1]; S q[2];
↳ Sdg q[3]; H q[1]; H q[2]; Vdg q[3]; S q[1]; S q[2]; S q[3]; V q[1]; V q[2]; H q[3];
↳ CX q[0], q[1]; S q[3]; CX q[0], q[2]; V 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]; S q[3]; CX
↳ q[0], q[1]; S q[2]; H q[3]; S q[0]; H q[2]; S q[3]; H q[0]; H q[1]; S q[2];
↳ V q[3]; S q[0]; S q[1]; V q[2]; V q[0]; V 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 states 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 #
```

(continues on next page)

(continued from previous page)

```

# ##### #

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 2023-11-29 09:48:03.249868
# TIMER BLOCK-0 ENDS - DURATION (s): 0.3071886 [0:00:00.307189]

VQE ENERGY: -1.1368465754720527

VQE REPORT:

{'minimizer': {'final_value': -1.1368465754720527,
   'final_parameters': array([-0.107,  0.,  0.,  1.]), 'final_value': -1.1368465754720527,
   'initial_parameters': [{('ordering': 0, 'symbol': 'd0', 'value': 0.0),
    ('ordering': 1, 'symbol': 's0', 'value': 0.0),
    ('ordering': 2, 'symbol': 's1', 'value': 0.0)}, {'('ordering': 0,
   'symbol': 'd0',
   'value': -0.10723347230091572),
    ('ordering': 1, 'symbol': 's0', 'value': 0.0),
    ('ordering': 2, 'symbol': 's1', 'value': 0.0}]}

```

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 Hartrees – a big improvement over the Hartree-Fock energy of -1.1175 Ha! 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](#).

17.2 Extended VQE

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 `inquito-pyscf` driver. Finally, we look at one optimization strategy in InQuanto - Ansatz parameter reduction by point group symmetry.

```
[ ]: from pytket.extensions.qiskit import AerStateBackend
from inquito.express import run_vqe
from inquito.minimizers import MinimizerScipy
from inquito.ansatzes import FermionSpaceAnsatzUCCSD
from inquito.mappings import QubitMappingJordanWigner
from inquito.extensions.pyscf import ChemistryDriverPySCFMolecularRHF
import datetime
import matplotlib.pyplot as plt
import numpy as np
```

17.2.1 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", disp=False)
    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 2023-11-29 10:03:18.932717
# TIMER BLOCK-0 ENDS - DURATION (s): 0.2546752 [0:00:00.254675]
(-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 2023-11-29 10:03:19.444925
# TIMER BLOCK-1 ENDS - DURATION (s): 0.2311640 [0:00:00.231164]
# TIMER BLOCK-2 BEGINS AT 2023-11-29 10:03:19.750268
# TIMER BLOCK-2 ENDS - DURATION (s): 0.3098021 [0:00:00.309802]
# TIMER BLOCK-3 BEGINS AT 2023-11-29 10:03:20.149157
# TIMER BLOCK-3 ENDS - DURATION (s): 0.2040837 [0:00:00.204084]
# TIMER BLOCK-4 BEGINS AT 2023-11-29 10:03:20.414429
# TIMER BLOCK-4 ENDS - DURATION (s): 0.2530793 [0:00:00.253079]
# TIMER BLOCK-5 BEGINS AT 2023-11-29 10:03:20.898705
# TIMER BLOCK-5 ENDS - DURATION (s): 0.2698489 [0:00:00.269849]
# TIMER BLOCK-6 BEGINS AT 2023-11-29 10:03:21.238844
# TIMER BLOCK-6 ENDS - DURATION (s): 0.2238931 [0:00:00.223893]
# TIMER BLOCK-7 BEGINS AT 2023-11-29 10:03:21.529253
# TIMER BLOCK-7 ENDS - DURATION (s): 0.2436052 [0:00:00.243605]
# TIMER BLOCK-8 BEGINS AT 2023-11-29 10:03:21.839684
# TIMER BLOCK-8 ENDS - DURATION (s): 0.1830366 [0:00:00.183037]
# TIMER BLOCK-9 BEGINS AT 2023-11-29 10:03:22.185276
# TIMER BLOCK-9 ENDS - DURATION (s): 0.2403583 [0:00:00.240358]
# TIMER BLOCK-10 BEGINS AT 2023-11-29 10:03:22.511148
# TIMER BLOCK-10 ENDS - DURATION (s): 0.2173594 [0:00:00.217359]
# TIMER BLOCK-11 BEGINS AT 2023-11-29 10:03:22.793701
# TIMER BLOCK-11 ENDS - DURATION (s): 0.2096147 [0:00:00.209615]
# TIMER BLOCK-12 BEGINS AT 2023-11-29 10:03:23.086062
# TIMER BLOCK-12 ENDS - DURATION (s): 0.1866202 [0:00:00.186620]
# TIMER BLOCK-13 BEGINS AT 2023-11-29 10:03:23.546918
# TIMER BLOCK-13 ENDS - DURATION (s): 0.2001028 [0:00:00.200103]
# TIMER BLOCK-14 BEGINS AT 2023-11-29 10:03:23.911464
# TIMER BLOCK-14 ENDS - DURATION (s): 0.2476650 [0:00:00.247665]
# TIMER BLOCK-15 BEGINS AT 2023-11-29 10:03:24.631068
# TIMER BLOCK-15 ENDS - DURATION (s): 0.1854311 [0:00:00.185431]
# TIMER BLOCK-16 BEGINS AT 2023-11-29 10:03:24.928800
# TIMER BLOCK-16 ENDS - DURATION (s): 0.2353134 [0:00:00.235313]
# TIMER BLOCK-17 BEGINS AT 2023-11-29 10:03:25.681557
# TIMER BLOCK-17 ENDS - DURATION (s): 0.2013109 [0:00:00.201311]
# TIMER BLOCK-18 BEGINS AT 2023-11-29 10:03:26.197478
# TIMER BLOCK-18 ENDS - DURATION (s): 0.3231277 [0:00:00.323128]
# TIMER BLOCK-19 BEGINS AT 2023-11-29 10:03:26.705219
# TIMER BLOCK-19 ENDS - DURATION (s): 0.2864726 [0:00:00.286473]
# TIMER BLOCK-20 BEGINS AT 2023-11-29 10:03:27.174827
# TIMER BLOCK-20 ENDS - DURATION (s): 0.2154477 [0:00:00.215448]
[(-0.9141497046270836, -0.9043613941635398), (-1.0396441933684182, -1.
˓→0278952240485912), (-1.102723451217337, -1.088582110334757), (-1.1303984654811357, -1.
˓→-1.1133931546411708), (-1.1373027360323475, -1.1169158055488677), (-1.
˓→1320031440770353, -1.1076586023375483), (-1.119647652656638, -1.0906923776851998), -1.
˓→(-1.1033573201316123, -1.0690432214496197), (-1.0850885605499416, -1.
˓→04456492118589), (-1.0661536707290789, -1.0184750668009017), (-1.047492324194345, -0.9916426659510071), (-1.0297900705318137, -0.9647203151980028), (-1.
```

(continues on next page)

(continued from previous page)

```

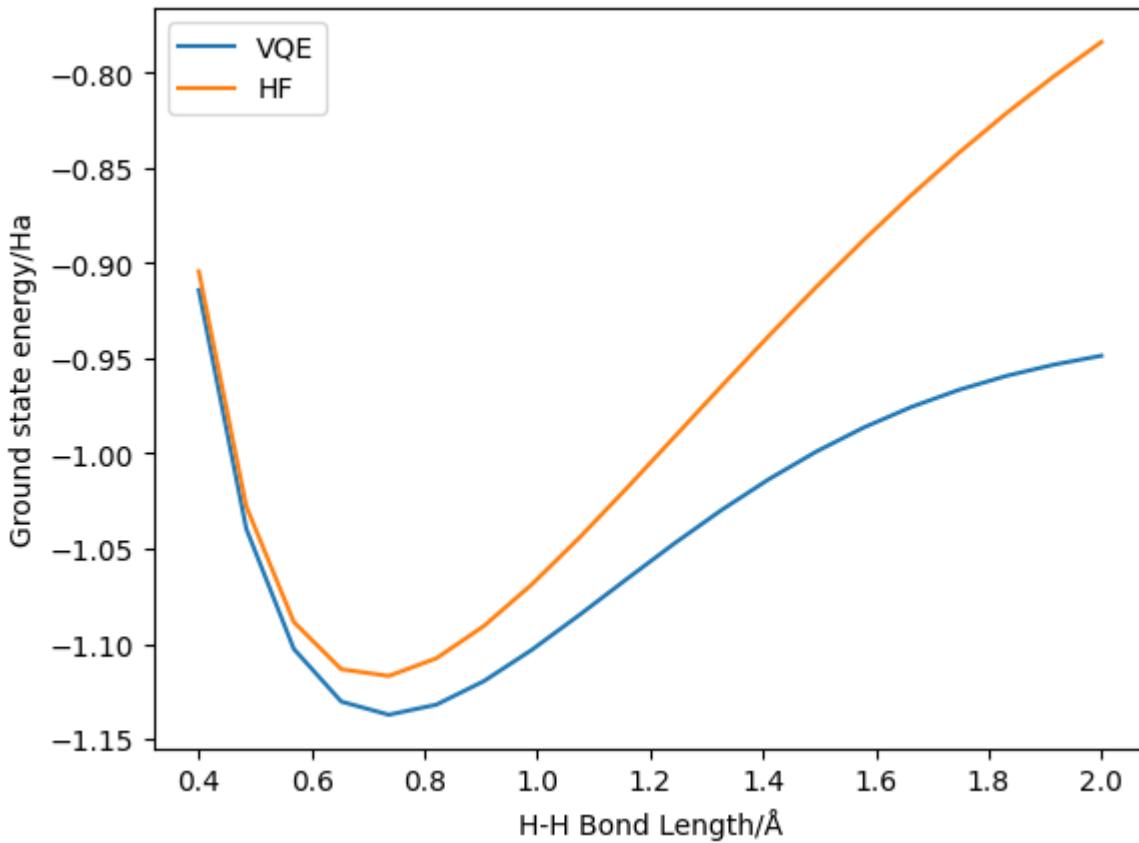
↪ 0135255910407293, -0.9382014307919533), (-0.9989959202496171, -0.9124510069368059),
↪ (-0.9863403143934912, -0.887729636381618), (-0.9755678355211699, -0.
↪ 8642149983652552), (-0.9665878877681977, -0.8420201240601859), (-0.9592413973101768,
↪ -0.8212077601664906), (-0.9533300339756414, -0.8018012103709642), (-0.
↪ 9486411121756362, -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 0x7fa7606c9930>
```



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.

17.2.2 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]

    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", disp=False)
    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 2023-11-29 10:03:30.842856
# TIMER BLOCK-21 ENDS - DURATION (s): 27.7681518 [0:00:27.768152]
(-75.01966834467395, -74.96466253913081)
```

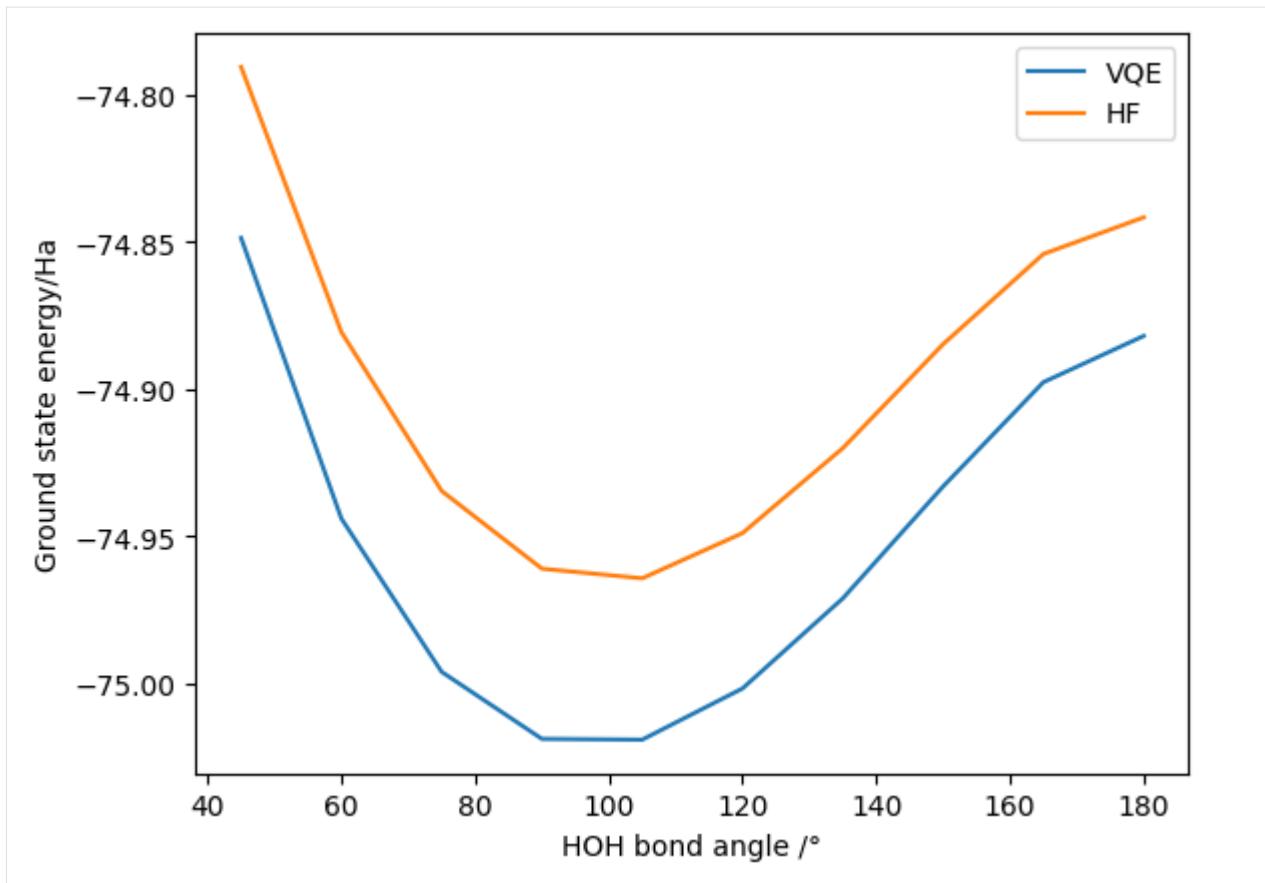
This block may take up to a minute to run, as the system is a bit bigger than molecular hydrogen. Here, we have asked the driver to freeze the lowest energy spatial orbital (i.e. the core electrons). Note that frozen orbitals are specified as a list of indices of spatial orbitals, not spin-orbitals - 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 2023-11-29 10:04:00.885648
# TIMER BLOCK-22 ENDS - DURATION (s): 29.3601284 [0:00:29.360128]
# TIMER BLOCK-23 BEGINS AT 2023-11-29 10:04:32.660531
# TIMER BLOCK-23 ENDS - DURATION (s): 27.9207343 [0:00:27.920734]
# TIMER BLOCK-24 BEGINS AT 2023-11-29 10:05:02.711111
# TIMER BLOCK-24 ENDS - DURATION (s): 26.3334000 [0:00:26.333400]
# TIMER BLOCK-25 BEGINS AT 2023-11-29 10:05:31.376028
# TIMER BLOCK-25 ENDS - DURATION (s): 24.9143963 [0:00:24.914396]
# TIMER BLOCK-26 BEGINS AT 2023-11-29 10:05:58.777418
# TIMER BLOCK-26 ENDS - DURATION (s): 27.3104123 [0:00:27.310412]
# TIMER BLOCK-27 BEGINS AT 2023-11-29 10:06:28.389795
# TIMER BLOCK-27 ENDS - DURATION (s): 27.1425993 [0:00:27.142599]
# TIMER BLOCK-28 BEGINS AT 2023-11-29 10:06:58.433549
# TIMER BLOCK-28 ENDS - DURATION (s): 28.5798548 [0:00:28.579855]
# TIMER BLOCK-29 BEGINS AT 2023-11-29 10:07:29.744182
# TIMER BLOCK-29 ENDS - DURATION (s): 31.6896270 [0:00:31.689627]
# TIMER BLOCK-30 BEGINS AT 2023-11-29 10:08:03.574060
# TIMER BLOCK-30 ENDS - DURATION (s): 34.3928248 [0:00:34.392825]
# TIMER BLOCK-31 BEGINS AT 2023-11-29 10:08:39.667253
# TIMER BLOCK-31 ENDS - DURATION (s): 21.5196126 [0:00:21.519613]
```

```
[ ]: h2o_angle_vqe_results,h2o_angle_hf_results = zip(*h2o_bending_results)
plt.plot(h2o_bond_angles,h2o_angle_vqe_results,label='VQE')
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 0x7fa76061b3a0>
```



17.2.3 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)
```

(continues on next page)

(continued from previous page)

```

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", disp=False)
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 2023-11-29 10:09:03.058718
# TIMER BLOCK-32 ENDS - DURATION (s): 13.7481062 [0:00:13.748106]
(-75.01969733754343, -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 2023-11-29 10:09:18.425163
# TIMER BLOCK-33 ENDS - DURATION (s): 13.3292246 [0:00:13.329225]
# TIMER BLOCK-34 BEGINS AT 2023-11-29 10:09:34.646286
# TIMER BLOCK-34 ENDS - DURATION (s): 13.9985131 [0:00:13.998513]
# TIMER BLOCK-35 BEGINS AT 2023-11-29 10:09:52.028530
# TIMER BLOCK-35 ENDS - DURATION (s): 13.5315848 [0:00:13.531585]
# TIMER BLOCK-36 BEGINS AT 2023-11-29 10:10:08.392973
# TIMER BLOCK-36 ENDS - DURATION (s): 12.2572551 [0:00:12.257255]
# TIMER BLOCK-37 BEGINS AT 2023-11-29 10:10:23.751241
# TIMER BLOCK-37 ENDS - DURATION (s): 14.3392197 [0:00:14.339220]
# TIMER BLOCK-38 BEGINS AT 2023-11-29 10:10:40.594766
# TIMER BLOCK-38 ENDS - DURATION (s): 13.9344880 [0:00:13.934488]
# TIMER BLOCK-39 BEGINS AT 2023-11-29 10:10:57.463713
# TIMER BLOCK-39 ENDS - DURATION (s): 12.4128931 [0:00:12.412893]
# TIMER BLOCK-40 BEGINS AT 2023-11-29 10:11:13.271035
# TIMER BLOCK-40 ENDS - DURATION (s): 16.7570208 [0:00:16.757021]
# TIMER BLOCK-41 BEGINS AT 2023-11-29 10:11:33.619949
# TIMER BLOCK-41 ENDS - DURATION (s): 21.5077045 [0:00:21.507704]
# TIMER BLOCK-42 BEGINS AT 2023-11-29 10:11:58.588764

```

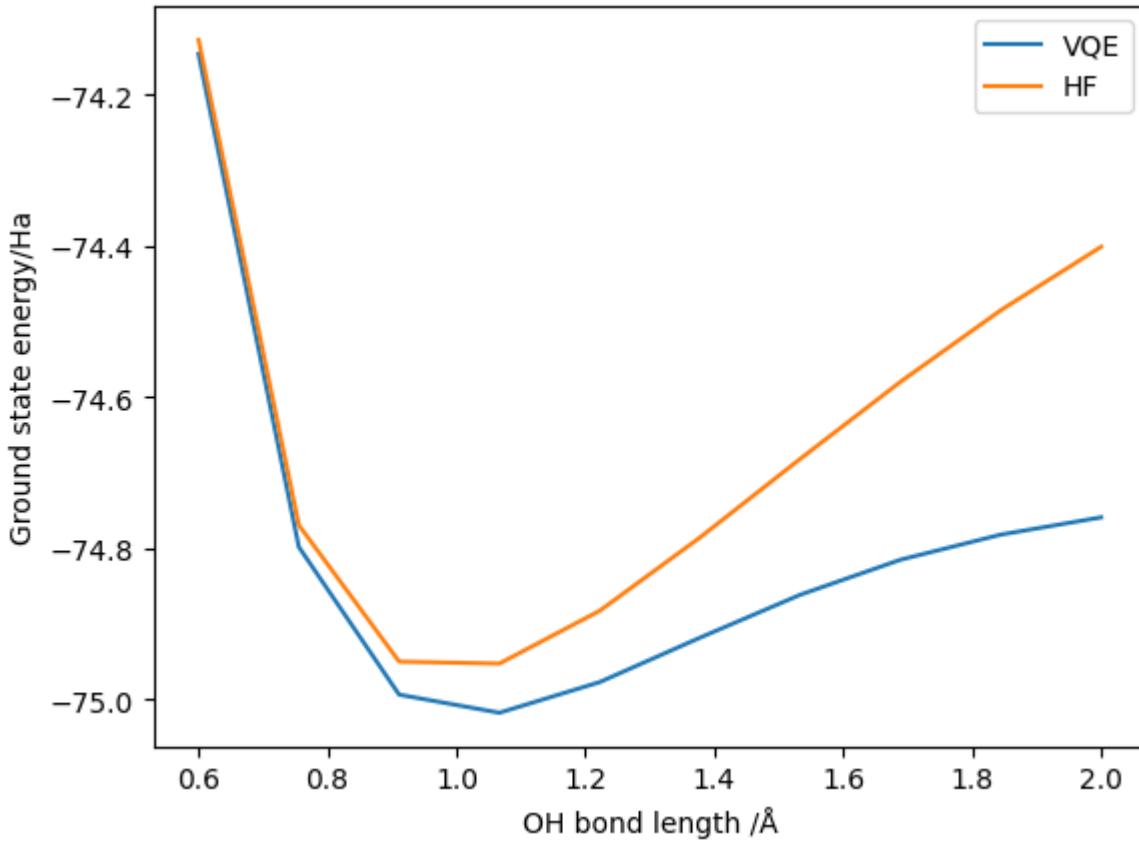
(continues on next page)

(continued from previous page)

TIMER BLOCK-42 ENDS - DURATION (s): 22.3463954 [0:00:22.346395]

```
[ ]: 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 0x7fa766242590>
```



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]
```

(continues on next page)

(continued from previous page)

```

driver_with_symmetry = ChemistryDriverPySCFMolecularRHF(basis=basis,
    ↪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}

```

17.3 Variational Quantum Deflation for excited states

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 Higgott, Wang and Brierley (2019) and more technical details, please go [here](#).

In VQE, one finds a minimum of the energy by classically optimizing 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 $|\psi(\lambda_0)\rangle, \dots, |\psi(\lambda_{k-1})\rangle$ 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 `inquanto.spaces.FermionSpace` and `inquanto.states.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.df())

```

	Coefficients	Terms
0	0.743018	
1	-1.270293	$F0^ F0$
2	0.340031	$F1^ F0^ F0 F1$
3	0.340031	$F1^ F0^ F0 F1$
4	0.089834	$F1^ F0^ F2 F3$
5	0.089834	$F1^ F0^ F2 F3$
6	-1.270293	$F1^ F1$
7	-0.089834	$F2^ F0^ F0 F2$
8	-0.089834	$F2^ F0^ F0 F2$
9	0.334289	$F2^ F0^ F0 F2$
10	0.334289	$F2^ F0^ F0 F2$
11	-0.089834	$F2^ F1^ F0 F3$
12	-0.089834	$F2^ F1^ F0 F3$
13	0.334289	$F2^ F1^ F1 F2$
14	0.334289	$F2^ F1^ F1 F2$
15	-0.456807	$F2^ F2$

(continues on next page)

(continued from previous page)

16	0.334289	F3^ F0^ F0	F3
17	0.334289	F3^ F0^ F0	F3
18	-0.089834	F3^ F0^ F1	F2
19	-0.089834	F3^ F0^ F1	F2
20	-0.089834	F3^ F1^ F1	F3
21	-0.089834	F3^ F1^ F1	F3
22	0.334289	F3^ F1^ F1	F3
23	0.334289	F3^ F1^ F1	F3
24	0.089834	F3^ F2^ F0	F1
25	0.089834	F3^ F2^ F0	F1
26	0.351407	F3^ F2^ F2	F3
27	0.351407	F3^ F2^ F2	F3
28	-0.456807		F3^ F3

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 `inquanto.ansatzes.FermionSpaceAnsatzkUpCCGSD` ansatz and the COBYLA minimizer available through the `inquanto.minimizers.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 `inquanto.protocols.PauliAveraging` – and the number of shots we wish to simulate in each iteration. Then, we build the algorithm object and execute.

```
[ ]: from inquanto.protocols import PauliAveraging

vqe.build(protocol_objective=PauliAveraging(AerBackend(), shots_per_circuit=10000))

vqe.run()

# VQE Energy:      -1.1354204303965678
# VQE Parameters: [ 1.301 -1.538  0.287  1.699 -1.339 -0.344]
```

(continues on next page)

(continued from previous page)

```

print("VQE Energy:    ", vqe.final_value)
print("VQE Parameters:", vqe.final_parameters.to_array())

# TIMER BLOCK-0 BEGINS AT 2023-11-29 10:09:26.090366
# TIMER BLOCK-0 ENDS - DURATION (s): 68.4655346 [0:01:08.465535]
VQE Energy:    -1.137201520635425
VQE Parameters: [ 1.222 -1.471  0.321  1.804 -1.145 -0.413]

```

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("{\lambda}_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 `inquanto.protocols.ComputeUncompute` 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 `inquanto.algorithms.AlgorithmVQD` constructor in the `n_vectors` argument.

```
[ ]: from inquanto.algorithms import AlgorithmVQD
from inquanto.protocols import ComputeUncompute
# 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,
)
# build object
backend=AerBackend()
```

(continues on next page)

(continued from previous page)

```
vqd.build(
    #small number of shots for demonstration purposes leads to large stochastic error
    objective_protocol=PauliAveraging(backend, shots_per_circuit=1000),
    weight_protocol=PauliAveraging(backend, shots_per_circuit=1000),
    overlap_protocol=ComputeUncompute(backend, n_shots=1000),
)

# execute
vqd.run()

# print results
# VQD excited state energies: [-1.1354204303965678, -0.4949467734066755, -0.
#                                ↪11981345040527547, 0.5478565335949009]
print("VQD excited state energies: ", vqd.final_values)

# TIMER BLOCK-1 BEGINS AT 2023-11-29 10:10:34.636622
# TIMER BLOCK-1 ENDS - DURATION (s): 38.7515074 [0:00:38.751507]
# TIMER BLOCK-2 BEGINS AT 2023-11-29 10:11:13.388266
# TIMER BLOCK-2 ENDS - DURATION (s): 65.3394235 [0:01:05.339423]
# TIMER BLOCK-3 BEGINS AT 2023-11-29 10:12:18.727942
# TIMER BLOCK-3 ENDS - DURATION (s): 79.3116225 [0:01:19.311622]
VQD excited state energies: [-1.137201520635425, -0.49272484310954634, -0.
#                                ↪1269800246167035, 0.5069123816017171]
```

17.4 Visualization with inquanto-nglview

It is often useful to visualize certain molecular data. For this purpose, InQuanto uses an extension to interface with the [NGLview](#) nglview package. The [inquanto-nglview](#) extension is focused 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.

17.4.1 Visualizing Structures

VisualizerNGL is initialized by an InQuanto Geometry object specifying molecular, or unit cell, geometry. 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 inquanto-nglview --index https://...
```

```
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]],
```

(continues on next page)

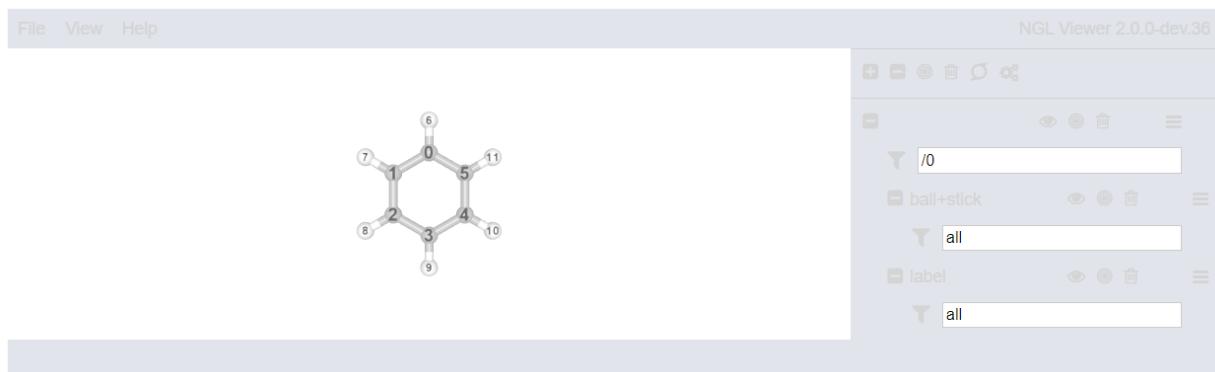
(continued from previous page)

```

['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]]
]
g = GeometryMolecular(xyz)
visualizer = VisualizerNGL(g)

```

```
[ ]: #visualizer.visualize_molecule(atom_labels="index").display(gui=True)
#markdown cell below is a presaved image
```

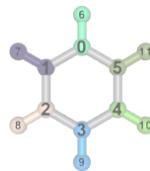


17.4.2 Visualizing Fragments

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")
```



17.4.3 Visualizing Orbitals

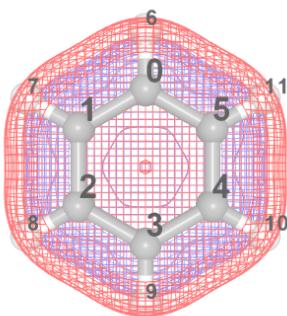
To aid in active space selection, it is also possible to view the molecular orbitals of a system if the appropriate `.cube` strings are available.

```
[ ]: from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF

driver = ChemistryDriverPySCFMolecularRHF(geometry=g.xyz, basis="sto3g")

# runs HF and returns orbitals
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]
```



CHAPTER
EIGHTEEN

BACKEND TUTORIALS

These tutorials detail how to best run calculations on different pytket backends, which provide access to simulators, emulators, and quantum hardware. We also highlight the stochastic nature of hamiltonian averaging (shots and sampling), as well as examine the role of noise and error mitigation methods.

<i>Aer Simulator</i>	Hamiltonian averaging with the Aer simulator (shots)	download
<i>H2 H-Series</i>	Using pytket-quantinuum to access the H-Series	download
<i>H-Series async</i>	Running asynchronous experiments on the H-Series	download
<i>H-Series QSE</i>	Quantum Subspace Expansion with the H-Series	download
<i>H-series setup</i>	Further details on accessing H series	
<i>IBMQ setup</i>	How to set up IBM Quantum credentials and backends	PDF version

18.1 Running on the Aer simulator

In this two-part tutorial, we demonstrate the process of executing a simple quantum chemical computation using ‘shot’ based sampling and on noisy quantum emulators and hardware.

Since this tutorial is focused on practical quantum computation, we will perform a straightforward calculation: the single-point evaluation of the total energy for the H₂ molecule using the Unitary Coupled Cluster (UCC) ansatz, without optimization or parameter variance.

This is the first part of this tutorial, and for the second part we examine the use of a *Quantinuum hardware emulator*. The use of IBM hardware (and emulator) is also possible, but note that this requires an IBM Quantum account and the

appropriate credentials to run on a machine with at least four qubits for a brief duration. Please refer to [this page](#) for instructions on how to set up calculations on IBM Quantum's devices.

The outlined steps are as follows:

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 (QuantinuumBackend emulator + machine noise profile)
- Demonstrate error mitigation methods on emulated hardware (PMSV)

18.1.1 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.

```
[ ]: from inquanto.express import load_h5

h2 = load_h5("h2_sto3g.h5", as_tuple=True)
hamiltonian = h2.hamiltonian_operator
```

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_{∞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
```

(continues on next page)

(continued from previous page)

```
exponents += space.construct_double_ucc_operators(state)
ansatz = FermionSpaceStateExpChemicallyAware(exponents, state)
```

The ansatz circuit structure is well-defined, but the guess for the wave function parameters (weights of the exponentiated determinants / angles of rotation gates) has not been established. These parameters are initialized with a single value denoted as ‘`p`’, detailed below.

Once the parameters are defined, the `Computable` class is employed to specify the quantity of interest: we seek to evaluate the expectation value of the Hamiltonian for a given parameter value θ , $E = \langle \Psi(\theta) | \hat{H} | \Psi(\theta) \rangle$.

For demonstration purposes, we fix the random seed on the initialization 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())

from inquanto.computables import ExpectationValue

expectation0 = ExpectationValue(ansatz, hamiltonian.qubit_encode())
          ordering symbol      value
0            0      d0  0.499676
```

With the circuit and parameters now established, it becomes possible to display and analyze the circuit. This particular circuit comprises 4 qubits and consists of 31 gates. Notably, among these gates, the multi-qubit CNOT gates, are expected to contribute the most noise. In this circuit, there are a total of 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 better comprehend the impact of noise on computing this circuit, in the following cell, we establish a set of “shots” to scan over, along with a random seed number and various other parameters. To illustrate the varying degrees of stochasticity and quantum noise across different backends, we will present convergence plots. These plots will demonstrate how the average energy converges towards the expectation value with increasing sampling.

```
[ ]: import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (12, 4)
```

(continues on next page)

(continued from previous page)

```
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
```

18.1.2 1. Noiseless Statevector simulation

The evaluation of a quantum measurement can be carried out directly by state vector methods, which perform 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. Note that statevector backends do not take number of shots as an argument. The needed resultant probability amplitudes are 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
from inquanto.computables import ExpectationValue
from inquanto.protocols import BackendStatevectorProtocol

backend = AerStateBackend()

expval_expression = ExpectationValue(ansatz, hamiltonian.qubit_encode())
protocol = BackendStatevectorProtocol(backend)
evaluator = protocol.get_evaluator(p)
statevector_energy = expval_expression.evaluate(evaluator=evaluator)

shotless_energies = [statevector_energy] * len(
    set_shots
) # for plotting later

print("Statevector energy is " + str(np.real(statevector_energy)) + "Ha")
Statevector energy is -0.5876463677224998Ha
```

18.1.3 2. Noiseless simulation

Subsequently, we proceed to simulate “shots”, representing the runs of the quantum computer, using the `AerBackend`. This simulation, devoid of quantum noise such as decoherence during processing, does incorporate the stochastic nature of measuring the system, culminating in a probabilistic collapse into a computational basis state upon simulation completion. Each individual shot yields a measurement outcome of 0 or 1 on each qubit. By accumulating numerous shots, one can estimate the expectation value of each Pauli string within the Hamiltonian. Higher shot counts lead to increased precision in these expectation values, ultimately resulting in enhanced precision of the overall Hamiltonian expectation value.

We generate a plot to illustrate the convergence of energy concerning the number of shots. This plot reveals that with approximately 1000 shots, the averaging of the system reaches a sufficient level, closely resembling the results obtained from the `AerStateBackend` (within a margin of 0.01 Ha).

We recommend changing the seed value in `protocol.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 PauliAveraging
from pytket.partition import PauliPartitionStrat

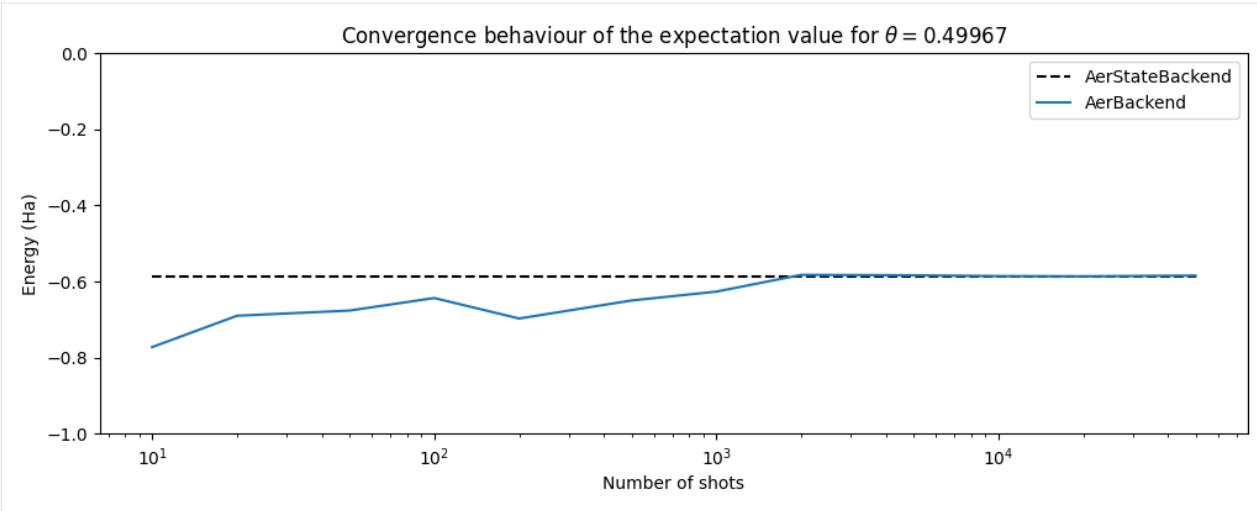
backend = []
backend = AerBackend()

protocol_template = PauliAveraging(
    backend,
    shots_per_circuit=10,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
)
protocol_template.build(p, ansatz, hamiltonian.qubit_encode())
protocol_pickle = protocol_template.dumps()

[ ]: noiseless_energies = []
for i in set_shots:
    protocol = PauliAveraging.loads(protocol_pickle, backend)
    protocol.shots_per_circuit = i
    protocol.run(seed=set_seed)
    aer_expectation = protocol.evaluate_expectation_value(
        ansatz, hamiltonian.qubit_encode()
    )
    noiseless_energies.append(aer_expectation)

[ ]: 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, 0])
plt.xlabel("Number of shots")
plt.ylabel("Energy (Ha)")
plt.title(
    "Convergence behavior of the expectation value for "
    + r"$\theta=$"
    + str(p.df().iloc[0, 2])[0:7]
)
plt.legend()

print(noiseless_energies[-1])
-0.5848319000310379
```



18.1.4 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 constructed by adding depolarising error to CNOT gates. This can be a reasonable first approximation to multi-qubit operations which 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 the readout error, which represents incorrectly measuring 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_aer.noise import NoiseModel # , ReadoutError
import qiskit_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
```

(continues on next page)

(continued from previous page)

```

) # this defaults to no noise, which is the same as NoiseModel.is_ideal
noisy_energies = []

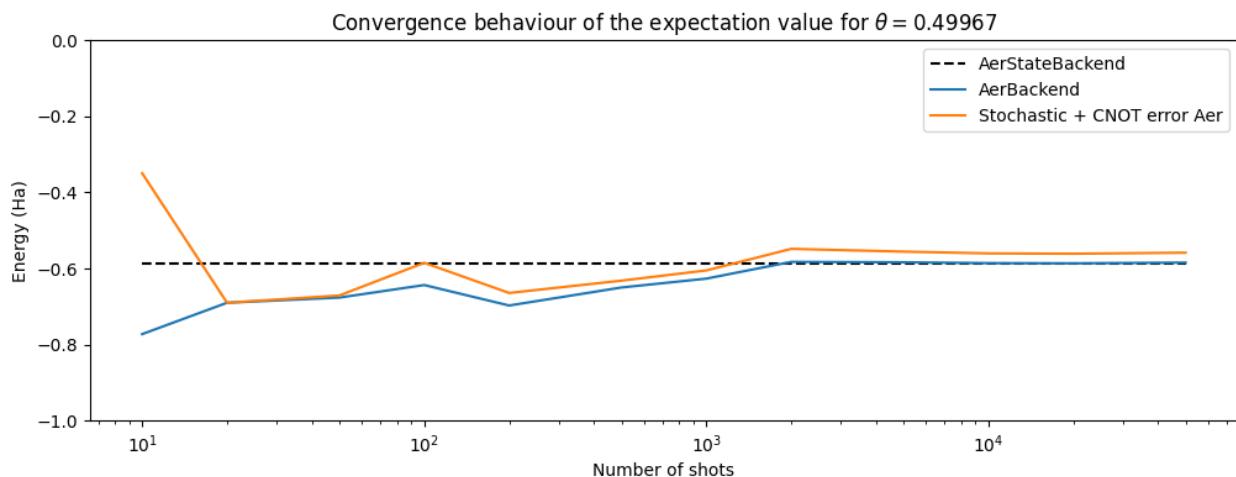
# rebuild protocol/circuits for new noisy backend
protocol_template = PauliAveraging(
    noisy_backend,
    shots_per_circuit=10,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
)
protocol_template.build(p, ansatz, hamiltonian.qubit_encode())
protocol_pickle = protocol_template.dumps()

for i in set_shots:
    protocol = PauliAveraging.loads(protocol_pickle, noisy_backend)
    protocol.shots_per_circuit = i
    protocol.run(seed=set_seed)
    aer_expectation = protocol.evaluate_expectation_value(
        ansatz, hamiltonian.qubit_encode()
    )
    noisy_energies.append(aer_expectation)

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, 0])
plt.xlabel("Number of shots")
plt.ylabel("Energy (Ha)")
plt.title(
    "Convergence behavior of the expectation value for "
    + r"$\theta=$"
    + str(p.df().iloc[0, 2])[0:7]
)
plt.legend()

```

<matplotlib.legend.Legend at 0x7f022a071010>



The final plot shows that noisy operations lead to a shift in the total expectation value due to biasing. Increasing the number of shots should increase precision but not accuracy without the use of error mitigation methods.

This tutorial has demonstrated the use of freely available Qiskit backends to evaluate a simple quantum system. The final example adds a simple model of quantum noise.

This is the first part of this tutorial, and for the second part we examine the use of a *Quantinuum hardware emulator* involving a more complex and realistic noise model, and the use of noise mitigation.

18.2 Running on Quantinuum H-Series

In this tutorial, we demonstrate how to perform a simple quantum chemical calculation on Quantinuum H-Series.

Since this tutorial focuses on practical quantum computation, we will perform a simple calculation: the single point (i.e. not optimized/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.

This tutorial will require that you have access to Quantinuum backends, which can be obtained through your InQuanto administrator or contacting Quantinuum support. You will need credentials to run on a machine for a short time. For more information on Quantinuum H-Series and instructions on granting access (for administrators), see [Quantinuum H-Series page](#).

The steps below are such:

- Define the system
- Perform computation with emulated hardware noise (QuantinuumBackend emulator + machine noise profile)
- Demonstrate error mitigation methods on emulated hardware (PMSV)

18.2.1 1. Define the system

```
[ ]: # Preload the Hamiltonian for H2
# see the Aer tutorial for details
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

# Define fermion space, state, and map the fermionic operator to qubits
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)
# Construct an efficient ansatz
```

(continues on next page)

(continued from previous page)

```

ansatz = FermionSpaceStateExpChemicallyAware(exponents, state)

p = ansatz.state_symbols.construct_from_array([0.4996755931358105])
print(p.df())

# Import an InQuanto Computable for measuring an expectation value.
# The operator is the qubit Hamiltonian, and the wavefunction is the ansatz.
from inquanto.computables import ExpectationValue

print(hamiltonian)
expectation0 = ExpectationValue(ansatz, hamiltonian.qubit_encode())

# Analyze the ansatz circuit
from pytket.circuit.display import render_circuit_jupyter
from pytket import Circuit, OpType

render_circuit_jupyter(ansatz.get_circuit(p)) # this is the uncompiled ansatz circuit

print("\nCNOT GATES: {}".format(ansatz.state_circuit.n_gates_of_type(OpType.CX)))
print(ansatz.state_circuit)

    ordering symbol      value
0          0      d0  0.499676
<inquanto.operators._chemistry_integral_operator.ChemistryRestrictedIntegralOperator_
 →object at 0x7fde25afa010>
<IPython.core.display.HTML object>

CNOT GATES: 4
<tket::Circuit, qubits=4, gates=31>

```

18.2.2 2. Machine emulation for quantum noise

Running emulator experiments before hardware experiments is a crucial step in the development and optimization of quantum algorithms and applications. Emulators provide a controlled environment where one can fine-tune algorithms, explore error mitigation strategies, and gain valuable insights about the behavior of quantum circuits without some constraints of physical hardware.

Below, we provide instructions for conducting experiments utilizing a Quantinuum H-series emulator. To utilize hardware instead of the emulator one only needs to change their choice of device when instantiating the `QuantinuumBackend`.

Note that we use the `pytket-quantinuum` extension to access a Quantinuum backend.

`QuantinuumBackend` is a `pytket` backend that calls an H-series device (“H1-1”, “H1-2”) or its emulator with the corresponding noise profile (“H1-1E”, “H2-1E”). The emulators are run remotely on a server. Accessing the backend retrieves information from your Quantinuum account. More information can be found on the [pytket-quantinuum](#) page.

For comparison in the figure below, we have also plotted the exact energy (-0.5876463677224993 Ha) for H_2 and the user should also compare these results to result from the first part of this tutorial.

```

[ ]: from pytket.extensions.quantinuum import QuantinuumBackend

# Initialize the backend, make sure to login with your credentials.
# Change the machine name and add the label and the group if necessary.

```

(continues on next page)

(continued from previous page)

```

machine = "H1-1E"
backend = QuantinuumBackend(device_name=machine, group="")

# The QuantinuumBackend has additional arguments for accessing resources and
# labelling circuits
# label (Optional[str], optional) - Job labels used if Circuits have no name,
# defaults to "job"
# group (Optional[str], optional) - string identifier of a collection of jobs, can be
# used for usage tracking.

# Running the next line (device_state) will require logging into your
# quantinuum account. This can also be called with backend.login()
print(machine, "status:", QuantinuumBackend.device_state(device_name=machine))

H1-1E status: online

```

```

[ ]: from inquanto.protocols import PauliAveraging
      from pytket.partition import PauliPartitionStrat

      # here we demonstrate building the
      protocol = PauliAveraging(
          backend,
          shots_per_circuit=10,
          pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
      )
      protocol.build(p, ansatz, hamiltonian.qubit_encode())

      # you can inspect compiled measurement circuits contained in the protocol that was
      # run on the backend
      # note that the gateset of this circuit is different to the ansatz

      render_circuit_jupyter(protocol.get_circuits()[1])

<IPython.core.display.HTML object>

```

```

[ ]: # now we loop over different numbers of shots to examine convergence
      # the protocols are built, run, and the expectation values collected
      set_shots = [10, 50, 100, 500, 1000, 5000, 10000]
      noisy_H1_1E_energies = []

      for i in set_shots:
          protocol = PauliAveraging(
              backend,
              shots_per_circuit=i,
              pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
          )
          protocol.build(p, ansatz, hamiltonian.qubit_encode())
          protocol.run()  # no seeding H series, returns results to the protocol
          noisy_H1_1E_expectation = protocol.evaluate_expectation_value(
              ansatz, hamiltonian.qubit_encode()
          )
          noisy_H1_1E_energies.append(noisy_H1_1E_expectation)

```

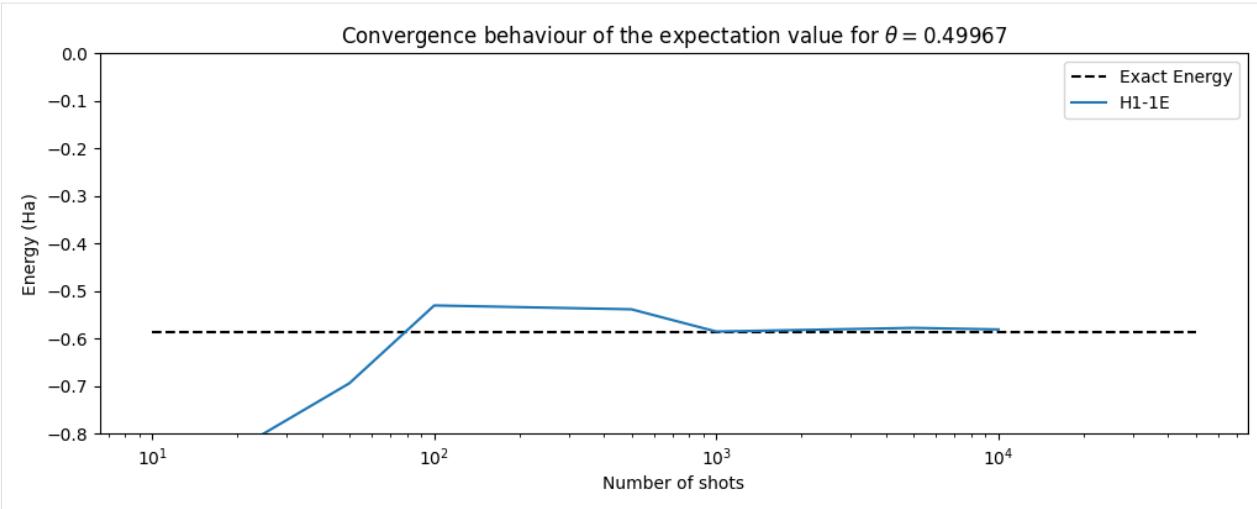
The Quantinuum Portal can be used to inspect your submitted job progress and other details as illustrated below.

Status	JobID	Name	MA-CHINF	Group	Submit Date	Start Date	Result Date	Shot:	Cost
queued	3049eb4cf05746ffb2	GQ4WEZBL	H1-1E	User-Group	29/11/2021 11:42			1000	251
queued	474e5da741a34c5b9	GQ4WEZBL	H1-1E	User-Group	29/11/2021 11:42			1000	185
completed	7dd6b36ea21a43c8b	GQ4WEZBL	H1-1E	User-Group	29/11/2021 11:40	29/11/2021 11:42	29/11/2021 11:42	5000	5000
completed	df15c1a8e2f442a694	GQ4WEZBL	H1-1E	User-Group	29/11/2021 11:40	29/11/2021 11:42	29/11/2021 11:42	5000	5000
								95	128

```
[ ]: import matplotlib.pyplot as plt
# we plot the results of our sampling
plt.rcParams["figure.figsize"] = (12, 4)

plt.hlines(
    y=-0.5876463677224993,
    xmin=10,
    xmax=50000,
    ls="--",
    colors="black",
    label="Exact Energy",
)
plt.plot(set_shots, noisy_H1_1E_energies, label="H1-1E")
plt.xscale("log")
plt.ylim([-0.8, 0])
plt.xlabel("Number of shots")
plt.ylabel("Energy (Ha)")
plt.title(
    "Convergence behavior of the expectation value for "
    + r"$\theta=$"
    + str(p.df().iloc[0, 2])[0:7]
)
plt.legend()

<matplotlib.legend.Legend at 0x7f605017bc70>
```



18.2.3 3. Noise mitigation methods in Quantinuum emulation

We can use noise mitigation techniques to reduce the impact of noise in our expectation value. In this case we will ‘purify’ results by discarding a shot if it has a certain error. There are many other mitigation methods.

Specifically, we will define the symmetries of the Qubit Operators in the system to use PMSV (Partition Measurement Symmetry Verification). As a result, noise mitigation improves the accuracy of the energy obtained from the quantum hardware compared to the unmitigated results and the exact energy of H2.

State Preparation and Measurement (SPAM) correction, which calibrates the calculation for system noise, can also be used.

```
[ ]: from inquanto.protocols.averaging._mitigation import PMSV
from inquanto.mappings import QubitMappingJordanWigner

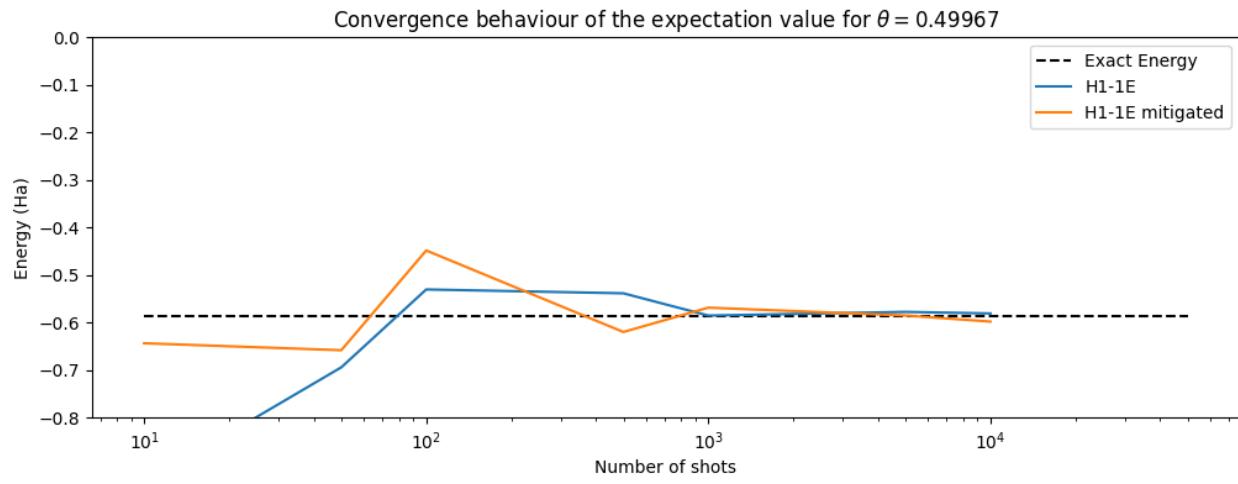
backend = QuantinuumBackend(device_name="", label="", group="")

stabilizers = QubitMappingJordanWigner().operator_map(
    space.symmetry_operators_z2_in_sector(state)
)

mitms_pmsv = PMSV(stabilizers)

miti_H1_1E_energies = []
for i in set_shots:
    protocol = PauliAveraging(
        backend,
        shots_per_circuit=i,
        pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
    )
    protocol.build(
        p, ansatz, hamiltonian.qubit_encode(), noise_mitigation=mitms_pmsv
    ).run()
    miti_H1_1E_expectation = protocol.evaluate_expectation_value(
        ansatz, hamiltonian.qubit_encode()
    )
    miti_H1_1E_energies.append(miti_H1_1E_expectation)
```

```
[ ]: #statevector
plt.hlines(
    y=-0.5876463677224993,
    xmin=10,
    xmax=50000,
    ls="--",
    colors="black",
    label="Exact Energy",
)
plt.plot(set_shots, noisy_H1_1E_energies, label="H1-1E")
plt.plot(set_shots, miti_H1_1E_energies, label="H1-1E mitigated")
plt.xscale("log")
plt.ylim([-0.8, 0])
plt.xlabel("Number of shots")
plt.ylabel("Energy (Ha)")
plt.title(
    "Convergence behavior of the expectation value for "
    + r"$\theta=$"
    + str(p.df().iloc[0, 2])[0:7]
)
plt.legend()
<matplotlib.legend.Legend at 0x7f6049d55ae0>
```



One may also explore the impact of SPAM on enhancing the convergence behavior of the expectation value. In the case of this simple system with a shallow circuit, the energy converges with slightly improved efficiency and/or heightened precision upon employing noise mitigation techniques.

Transitioning to hardware experiments is straightforward. This is facilitated by the `pytket.extensions.quantinuum` module. The user simply changes the device name (e.g., from “H1-1E” to “H1-1”), and the circuit will be run on the physical quantum device provided sufficient credits and circuit syntax.

18.3 Quantinuum H-Series - Launching circuits and retrieving results

In this tutorial, we will demonstrate the process of initiating circuits on the backend and obtaining the corresponding result handles using the Quantinuum H-Series. To illustrate this, we will conduct a Hamiltonian averaging experiment for a 6-qubit active-space of the Methane chemical system.

This tutorial will require that your InQuanto administrator has granted you access to the Quantinuum systems. You will also need credentials to run on a 6 qubit machine for a short time.

The steps below are such:

- Configuring the Quantinuum backend
- Define the system
- Find optimal VQE parameters on a noiseless simulator
- Define protocol and build InQuanto computables
- Submit circuits to backend
- Retrieve results from backend
- Evaluate the expectation value of the system

The express module is used to load in the converged mean-field (Hartree-Fock) spin-orbitals, and Hamiltonian from a H₂ computation using the STO-3G basis set.

```
[ ]: import warnings
warnings.filterwarnings('ignore')

from inquanto.express import load_h5

h2 = load_h5("h2_sto3g.h5", as_tuple=True)
hamiltonian = h2.hamiltonian_operator
```

The Fermionic space (`FermionSpace`) is defined with 4 spin-orbitals (which matches the full H₂ STO-3G space) and the D_{2h} point group is employed. 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 Fermionic state (`FermionState`) is then determined by the ground state occupations [1,1,0,0] and the Hamiltonian is encoded from the Hartree-fock integrals. The `qubit_encode` function carries out qubit encoding, utilizing the mapping class associated with the current integral operator. The default mapping approach is the Jordan-Wigner method.

```
[ ]: from inquanto.spaces import FermionSpace
from inquanto.states import FermionState
from inquanto.symmetry import PointGroup

space = FermionSpace(
    4, point_group=PointGroup("D2h"), orb_irreps=["Ag", "Ag", "B1u", "B1u"]
)

state = FermionState([1, 1, 0, 0])
qubit_hamiltonian = hamiltonian.qubit_encode()
```

To construct our ansatz for the specified fermion space and fermion state, we have employed the Chemically Aware Unitary Coupled Cluster method with singles and doubles excitations (UCCSD). The circuit is synthesized using Jordan-Wigner encoding.

```
[ ]: from inquanto.ansatzes import FermionSpaceAnsatzChemicallyAwareUCCSD

ansatz = FermionSpaceAnsatzChemicallyAwareUCCSD(space, state)
```

Here, we perform a simple experiment using a Variational Quantum Eigensolver (VQE) on a statevector backend to identify the optimal parameters that result in the ground state energy of our system. This enables us to carry out experiments

on both quantum hardware and emulators using these pre-optimized parameters. For a more comprehensive guide on performing VQE calculations using InQuanto on quantum computers, we recommend referring to the [VQE tutorial](#).

```
[ ]: from inquanto.express import run_vqe
from pytket.extensions.qulacs import QulacsBackend

state_backend = QulacsBackend()

vqe = run_vqe(ansatz, hamiltonian, backend=state_backend, with_gradient=False)

parameters = vqe.final_parameters

# TIMER BLOCK-0 BEGINS AT 2024-02-19 09:59:11.452381
# TIMER BLOCK-0 ENDS - DURATION (s): 0.3955611 [0:00:00.395561]
```

To reduce errors and inaccuracies caused by quantum noise and imperfections in the Quantinuum device, we can employ noise mitigation techniques. In this case, we will define the Qubit Operator symmetries within the system, enabling us to utilize [PMSV](#) (Partition Measurement Symmetry Verification). PMSV is an efficient technique for symmetry-verified quantum calculations. It represents molecular symmetries using Pauli strings, including mirror planes (Z_2) and electron-number conservation (U_1). For systems with Abelian point group symmetry, qubit tapering methods can be applied. PMSV uses commutation between Pauli symmetries and Hamiltonian terms for symmetry verification. It groups them into sets of commuting Pauli strings. If each string in a set commutes with the symmetry operator, measurement circuits for that set can be verified for symmetry without additional quantum resources, discarding measurements violating system point group symmetries.

Parameters used:

`stabilisers` – List of state stabilizers as `QubitOperators` with only single pauli strings in them.

The `InQuanto symmetry_operators_z2_in_sector` function is employed to retrieve a list of symmetry operators applicable to our system. These symmetry operators are associated with the point group, spin parity, and particle number parity Z_2 symmetries that uphold a specific symmetry sector. You can find additional details regarding this in the linked page.

```
[ ]: from inquanto.protocols import PMSV
from inquanto.mappings import QubitMappingJordanWigner

stabilizers = QubitMappingJordanWigner().operator_map(
    space.symmetry_operators_z2_in_sector(state)
)

mitms_pmsv = PMSV(stabilizers)
```

To simulate the specific noise profiles of machines, we can load and apply them to our simulations using the `QuantinuumBackend`, which retrieves information from your Quantinuum account. The `QuantinuumBackend` offers a range of available emulators, such as H1-1E and H1-2E. These are device-specific emulators for the corresponding hardware devices. These emulators run only remotely on a server. Additional information about the `pytket-quantinuum` extension can be found in the [link](#).

Parameters used:

`device_name` – Name of device, e.g. “H1-1E”

`label` – Job labels used if Circuits have no name, defaults to “job”

`group` – string identifier of a collection of jobs, can be used for usage tracking.

```
[ ]: from pytket.extensions.quantinuum import QuantinuumBackend
backend = QuantinuumBackend(device_name="H1-1E", label="test", group ="Default - UK")
```

To compute the expectation value of a Hermitian operator through operator averaging on the system register, we employ the PauliAveraging protocol. This protocol effectively implements the procedure outlined in ‘Operator Averaging’.

```
[ ]: from inquanto.protocols import PauliAveraging
from pytket.partition import PauliPartitionStrat

protocol = PauliAveraging(
    backend,
    shots_per_circuit=5000,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
)
```

A protocol has been constructed to process a computable dataset and calculate the expected value.

```
[ ]: # requires Quantinuum credentials
protocol.build(parameters, ansatz, qubit_hamiltonian, noise_mitigation=mitms_pmsv)
<inquanto.protocols.averaging._pauli_averaging.PauliAveraging at 0x7faf11e91c00>
```

You can also display a Pandas DataFrame using `dataframe_measurements` containing columns ‘pauli_string,’ ‘mean,’ and ‘stderr.’ Each row corresponds to a distinct Pauli string and its respective mean and standard error. Moreover, the `dataframe_circuit_shot` function generates a Pandas DataFrame containing circuit, shot, and depth details.

```
[ ]: print(protocol.dataframe_measurements())
print('')
print(protocol.dataframe_circuit_shot())

  pauli_string  mean  stderr  umean  sample_size
0            Z0  None   None   None        None
1            Z3  None   None   None        None
2            Z2  None   None   None        None
3      Y0 Y1 X2 X3  None   None   None        None
4      X0 Y1 Y2 X3  None   None   None        None
5            Z1 Z3  None   None   None        None
6      X0 X1 Y2 Y3  None   None   None        None
7            Z2 Z3  None   None   None        None
8            Z0 Z2  None   None   None        None
9            Z1 Z2  None   None   None        None
10           Z1  None   None   None        None
11           Z0 Z1  None   None   None        None
12           Z0 Z3  None   None   None        None
13      Y0 X1 X2 Y3  None   None   None        None

  Qubits  Depth  Depth2q  DepthCX  Shots
0        4     11       5       0    5000
1        4      8       3       0    5000
Sum      -      -      -      -   10000
```

The `dumps` function allows you to pickle protocols for later reloading using `loads`. Additionally, you have the option to clear internal protocol data using `clear`.

```
[ ]: pickled_data = protocol.dumps()
new_protocol = PauliAveraging.loads(pickled_data, backend)

protocol.clear()

<inquanto.protocols.averaging._pauli_averaging.PauliAveraging at 0x7faf11e91c00>
```

Running an experiment involves launching the circuits to the backend using the `launch` function. This approach handles all the circuits related to the expectation value calculations and provides a list of `ResultHandle` objects, each representing a handle for the results. Alternatively, an experiment can be initiated by employing the `run` function, which automatically executes the launch and retrieve methods. Typically, the `run` method is more useful for statevector calculations where you will receive your results from the backend immediately. On the other hand, `launch` and `retrieve` are more suitable for situations in which you expect a delay in receiving the results.

You could attempt both methods and print out the computational details to verify that you obtain the same results.

```
[ ]: handles = new_protocol.launch()
```

We can pickle these `ResultHandle` objects so we can retrieve the results once there are ready.

```
[ ]: import pickle

with open("handles.pickle", "wb") as handle:
    pickle.dump(handles, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

You can monitor the progress of your experiments on the [Quantinuum page](#) using the same credentials you used to run the experiments.

After our experiments have finished, we can obtain the results by utilizing the `retrieve` function, which retrieves distributions from the backend for the specified source. The expectation value of a kernel for a specified quantum state is calculated by using the `evaluate_expectation_value` function. In addition, we have employed the `evaluate_expectation_uvalue` function, which calculates the expectation value of the Hermitian kernel while considering linear error propagation theory.

```
[ ]: with open("handles.pickle", "rb") as handle:
    new_handles = pickle.load(handle)
print(new_handles)

[ResultHandle('50106a91ceb048e3bba7bcab26de7320', 'null', 4, '[["c", 0], ["c", 1], ["c", 2], ["c", 3]]'), ResultHandle('483171224e0b43469e22fb7467939d0a', 'null', 4, '[["c", 0], ["c", 1], ["c", 2], ["c", 3]])]
```

We can use the backend to simply query the job to see if it has completed. If all the circuits have run we can collect and process the results.

```
[ ]: completion_check=backend.circuit_status(new_handles[-1])[0].value #n-1
print(completion_check)

Circuit is queued.
```

Below we evaluate the expectation value and its uncertainty due to noise and sampling.

```
[ ]: if completion_check=='Circuit has completed. Results are ready.':
    new_protocol.retrieve(new_handles)

    energy_value = new_protocol.evaluate_expectation_value(ansatz, qubit_hamiltonian)
```

(continues on next page)

(continued from previous page)

```

print("Energy Value:\n{}".format(energy_value))

error = new_protocol.evaluate_expectation_uvalue(ansatz, qubit_hamiltonian)
print("Energy with error:\n{}".format(error))
else:
    print('Results not yet complete. ')

```

Energy Value:
-1.1364071606946333
Energy with error:
-1.1364+/-0.0018

18.4 Quantinuum H-Series - Quantum Subspace Expansion

This tutorial illustrates the computation of molecular excited states through the application of the Quantum Subspace Expansion (QSE) technique on the Quantinuum H-series remote emulator. The QSE method leverages a precise calculation of the ground state energy for a specific molecular configuration to approximate the energies of corresponding excited states. Consider a stable molecule, denoted by a Hamiltonian H , and let $|\Psi_0\rangle$ represent the outcome of the Variational Quantum Eigensolver ([VQE](#)) algorithm employed to estimate the ground state energy of H . More information on the QSE method can be found in the publication available at this [link](#).

The QSE technique constructs a subspace of state vectors $|\Psi_j^k\rangle$ formed by one-electron excitations of the ground state wavefunction:

$$|\Psi_j^k\rangle = c_k^\dagger c_j |\Psi_0\rangle. \quad (18.1)$$

where c_k^\dagger, c_j are the fermionic creation and annihilation operators over spin orbitals k and j , respectively. That is, these vectors are formed by reducing the occupation of spin orbital j by one, and increasing the occupation of spin orbital k by one. The vectors are not in general orthogonal to Ψ_0 hence we will need to calculate an overlap matrix.

Within this subspace, we solve a generalized eigenvalue problem. Consider the operator \hat{H} with matrix elements given by

$$(H)_{jk}^{lm} = \langle \Psi_j^l | \hat{H} | \Psi_k^m \rangle, \quad (18.2)$$

and define an overlap matrix S whose matrix elements are given by

$$S_{jk}^{lm} = \langle \Psi_j^l | \Psi_k^m \rangle. \quad (18.3)$$

The generalized eigenvalue equation to be solved is

$$HC = SCE, \quad (18.4)$$

where C is the matrix of eigenvectors, and E is the vector of eigenvalues. Crucially, the energy eigenvalues E provide an estimate of the excited state energies of H as well as a refined value of the ground state energy.

Notice that the solution to the generalized eigenvalue equation can be done on a classical computer, provided H and S have been calculated. The matrix elements of both of these matrices can be constructed using a quantum computer, in the following way. First, re-write the matrix elements in terms of $|\Psi_0\rangle$:

$$(H)_{jk}^{lm} = \langle \Psi_j^l | \hat{H} | \Psi_k^m \rangle = \langle \Psi_0 | c_j^\dagger c_l \hat{H} c_m^\dagger c_k | \Psi_0 \rangle \quad (18.5)$$

$$S_{jk}^{lm} = \langle \Psi_j^l | \Psi_k^m \rangle = \langle \Psi_0 | c_j^\dagger c_l c_m^\dagger c_k | \Psi_0 \rangle. \quad (18.6)$$

The matrix elements can be calculated using a quantum computer or simulator. By transforming the operators $c_j^\dagger c_l c_k^\dagger c_m$ and $c_l^\dagger c_j \hat{H} c_m^\dagger c_k$ to a set of Pauli quantum gates according to an appropriate scheme such as Jordan-Wigner or Bravyi-Kitaev, apply this gate set to the ground state wavefunction (constructed with the coefficients obtained from the VQE calculation) and perform a measurement to obtain the expected value. By measuring these expectation values, we obtain estimates of S_{jk}^{lm} and H_{jk}^{lm} , respectively.

This tutorial will guide you through the process of performing QSE and will show how to collate the resulting eigen-energies and eigen-vectors. To demonstrate these concepts, we will utilize the methane (CH4) chemical system. Note that to follow this tutorial, you must have been granted access to Quantinuum systems by your InQuanto administrator.

The steps below are such:

- Define the system
- Prepare the approximate ground state obtained by the VQE algorithm
- Configure the Quantinuum backend
- Establish the PMSV error mitigation
- Define protocol for measuring QSE matrix elements
- Build InQuanto computables
- Submit and retrieve InQuanto computables
- The black-box approach using the AlgorithmQSE

After defining the Z-matrix for Methane, the Hamiltonian object is created using a restricted Hartree-Fock (RHF) InQuanto-PySCF driver.

The `get_system` function is responsible for computing the fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

```
[ ]: import warnings
warnings.filterwarnings('ignore')

from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF

zmatrix = """
C
H      1    1.083000
H      1    1.083000      2   109.471000
H      1    1.083000      2   109.471000      3   120.000000
H      1    1.083000      2   109.471000      4   120.000000
"""

driver = ChemistryDriverPySCFMolecularRHF(
    zmatrix=zmatrix,
    charge=0,
    frozen=[0, 1, 2, 3, 7, 8],
    basis="STO-3G",
    point_group_symmetry=True,
)

hamiltonian, space, state = driver.get_system()
```

The `qubit_encode` function carries out qubit encoding, utilizing the mapping class associated with the current integral operator. The default mapping approach is the Jordan-Wigner method. Additionally, we employ the `compress` method,

which eliminates Hamiltonian terms with coefficients less than `abs_tol`, chosen as 10^{-6} here. This step is a means of reducing the number of measurement circuits/quantum computational resources needed (which means less time to run which is preferred for a tutorial), and the threshold we use is fairly reasonable compared to the scale of errors induced by noisy quantum simulation.

	Coefficient	Term	Coefficient	Type
0	-37.822198		<class 'numpy.float64'>	
1	-0.496463	Z5	<class 'numpy.float64'>	
2	-0.496463	Z4	<class 'numpy.float64'>	
3	0.128963	Z4 Z5	<class 'numpy.float64'>	
4	-0.496463	Z3	<class 'numpy.float64'>	
5	0.110247	Z3 Z5	<class 'numpy.float64'>	
6	0.123222	Z3 Z4	<class 'numpy.float64'>	
7	-0.012975	X2 X3 Y4 Y5	<class 'numpy.float64'>	
8	0.012975	X2 Y3 Y4 X5	<class 'numpy.float64'>	
9	0.012975	Y2 X3 X4 Y5	<class 'numpy.float64'>	
10	-0.012975	Y2 Y3 X4 X5	<class 'numpy.float64'>	
11	-0.496463	Z2	<class 'numpy.float64'>	
12	0.123222	Z2 Z5	<class 'numpy.float64'>	
13	0.110247	Z2 Z4	<class 'numpy.float64'>	
14	0.128963	Z2 Z3	<class 'numpy.float64'>	
15	-0.067741	Z1	<class 'numpy.float64'>	
16	0.106870	Z1 Z5	<class 'numpy.float64'>	
17	0.118715	Z1 Z4	<class 'numpy.float64'>	
18	0.106870	Z1 Z3	<class 'numpy.float64'>	
19	0.118715	Z1 Z2	<class 'numpy.float64'>	
20	-0.011845	X0 X1 Y4 Y5	<class 'numpy.float64'>	
21	-0.011845	X0 X1 Y2 Y3	<class 'numpy.float64'>	
22	0.011845	X0 Y1 Y4 X5	<class 'numpy.float64'>	
23	0.011845	X0 Y1 Y2 X3	<class 'numpy.float64'>	
24	0.011845	Y0 X1 X4 Y5	<class 'numpy.float64'>	
25	0.011845	Y0 X1 X2 Y3	<class 'numpy.float64'>	
26	-0.011845	Y0 Y1 X4 X5	<class 'numpy.float64'>	
27	-0.011845	Y0 Y1 X2 X3	<class 'numpy.float64'>	
28	-0.067741	Z0	<class 'numpy.float64'>	
29	0.118715	Z0 Z5	<class 'numpy.float64'>	
30	0.106870	Z0 Z4	<class 'numpy.float64'>	
31	0.118715	Z0 Z3	<class 'numpy.float64'>	
32	0.106870	Z0 Z2	<class 'numpy.float64'>	
33	0.123178	Z0 Z1	<class 'numpy.float64'>	

To construct our ansatz for the specified fermion space and fermion state, we have employed the Chemically Aware Unitary Coupled Cluster method with singles and doubles excitations (UCCSD). The circuit is synthesized using Jordan-Wigner encoding.

```
[ ]: from inquanto.ansatze import FermionSpaceAnsatzChemicallyAwareUCCSD
ansatz = FermionSpaceAnsatzChemicallyAwareUCCSD(space, state)
```

Next, a noiseless VQE simulation is executed on the QulacsBackend. The goal is to achieve a minimized expectation value and a set of circuit parameters corresponding to the ground state. These optimized parameters are subsequently

employed in the Hamiltonian averaging experiment later on.

```
[ ]: from inquanto.express import run_vqe
from pytket.extensions.qulacs import QulacsBackend

state_backend = QulacsBackend()
vqe = run_vqe(ansatz, hamiltonian, backend=state_backend, with_gradient=False)

parameters = vqe.final_parameters

# TIMER BLOCK-0 BEGINS AT 2024-02-19 10:02:38.479944
# TIMER BLOCK-0 ENDS - DURATION (s): 0.8142243 [0:00:00.814224]
```

To simulate the specific noise profiles of machines, we can load and apply them to our simulations using the `QuantinuumBackend`, which retrieves information from your Quantinuum account. The `QuantinuumBackend` offers a range of available emulators, such as H1-1E and H1-2E. These emulators are designed for specific devices and they run remotely on a server.

Parameters used:

`device_name` – Name of device, e.g. “H1-1”
`label` – Job labels used if Circuits have no name, defaults to “job”
`group` – string identifier of a collection of jobs, can be used for usage tracking.

The `pytket-quantinuum` extension allows the user to access the following quantum devices, emulators and syntax checkers.

```
[ ]: from pytket.extensions.quantinuum import QuantinuumBackend

backend = QuantinuumBackend(device_name="H1-1E", group ="Default - UK")
```

To reduce errors and inaccuracies caused by quantum noise and imperfections in the Quantinuum device, we can employ noise mitigation techniques. In this case, we will define the Qubit Operator symmetries within the system, enabling us to utilize PMSV (Partition Measurement Symmetry Verification). More information about `QubitOperator` can be found in the provided [link](#).

```
[ ]: from inquanto.operators import QubitOperator
from inquanto.protocols import PMSV

stabilizers = [QubitOperator("Z0 Z1 Z2 Z3 Z4 Z5", 1)]

mitms_pmsv = PMSV(stabilizers)
```

To expand the subspace for computing low-lying excited states, it is essential to define the basis. In this context, the function `generate_subspace_singlet_singles` is employed, which sequentially provides spin-adapted singlet-single excitation operators to preserve spin symmetry which are then mapped to qubits using the Jordan-Wigner method.

```
[ ]: from inquanto.mappings import QubitMappingJordanWigner

mapping = QubitMappingJordanWigner()

fermionic_excitations_singlets = list(space.generate_subspace_singlet_singles())
qubit_excitations_singlets = mapping.operator_map(fermionic_excitations_singlets)
```

Now that the Hamiltonian, Ansatz and QSE basis operators are defined, we can build the computable. The computable is the chemical quantity that we are interested in. In this example, we are interested in building the QSE computable using the `QSEMatricesComputable` function.

Parameters used:

state – Ansatz used to represent the ground state, it is used as a reference state for the excitations to generate the subspace.

hermitian_operator – A hermitian operator, typically a hamiltonian, to be expanded in the subspace.

expansion_operators – A List of excitation operators spanning the subspace.

```
[ ]: from inquanto.computables.composite import QSEMatricesComputable

qse = QSEMatricesComputable(
    state=ansatz,
    hermitian_operator=qubit_hamiltonian,
    expansion_operators=qubit_excitations_singlets,
)
```

To compute the expectation value of a Hermitian operator through operator averaging on the system register, we employ the PauliAveraging protocol. This protocol effectively implements the procedure outlined in [Operator Averaging](#).

```
[ ]: from inquanto.protocols import PauliAveraging
from pytket.partition import PauliPartitionStrat

protocol = PauliAveraging(
    backend=backend,
    shots_per_circuit=6000,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
)

protocol.build_from(parameters=parameters, computable=qse, noise_mitigation=mitms_
˓→pmsv)
<inquanto.protocols.averaging._pauli_averaging.PauliAveraging at 0x16ac06990>
```

Despite having many Qubit Operators to measure, this protocol ends up generating 30 measurement circuits due to the commutation relations measurement partitioning strategy pytket uses.

```
[ ]: protocol.n_circuit
30
```

To launch the circuits to the backend we have used the `launch` function. This method processes all the circuits associated with the expectation value calculations and returns a list of `ResultHandle` objects representing the handles for the results. We can pickle these `ResultHandle` objects so we can retrieve the results once they are ready. After our experiments have finished, we can obtain the results by utilizing the `retrieve` function, which retrieves distributions from the backend for the specified source.

```
[ ]: handles = protocol.launch()

[ ]: import pickle

with open("handles.pickle", "wb") as handle:
    pickle.dump(handles, handle, protocol=pickle.HIGHEST_PROTOCOL)

[ ]: completion_check=backend.circuit_status(handles[-1])[0].value #n-1
print(completion_check)
```

```
Circuit is queued.
```

```
[ ]: protocol.retrieve(handles)
<inquanto.protocols.averaging._pauli_averaging.PauliAveraging at 0x7fa080330990>
```

Once the results have been retrieved the Hamiltonian expectation value is evaluated as a classical post-processing step.

We now need to pass the output of the `QSEMatricesComputable` into `pd_safe_eigh`. This will solve the generalized eigenvalue equation and return the eigenvalues and eigenvectors.

```
[ ]: H, S = qse.evaluate(protocol.get_evaluator())
```

```
[ ]: from inquanto.core import pd_safe_eigh

e_vals_singlets, e_coeffs_singlets, _ = pd_safe_eigh(H, S)

print(e_vals_singlets)
print(e_coeffs_singlets)

[-39.729 -38.91 -38.871 -38.768 -38.273 -37.903]
[[-0.156+0.476j 0.004-0.008j 0.002-0.006j -0.018+0.023j 0.002-0.015j
 -0. -0.002j]
 [-0.311-0.135j -0.24 +0.368j -0.567+0.08j 3.806+1.333j 0.888+0.622j
 -0.223-0.734j]
 [-0.388-0.069j -0.606-0.105j -0.781+0.008j 4.621+0.967j -1.663-0.86j
 -0.85 +0.603j]
 [-0.003-0.003j 0.576+0.164j -0.384+0.022j 0.06 +0.043j -0.061+0.026j
 -0.004-0.055j]
 [-0.077-0.165j -1.04 +0.671j -0.19 -0.518j 0.644+1.722j -1.634+2.515j
 -5.204+1.559j]
 [ 0.16 +0.204j 0.08 +1.131j 0.391+0.141j -2.197-2.655j 0.879+3.241j
 -2.243+2.55j ]
 [-0.006+0.003j 0.111+0.338j 0.363+0.46j 0.113+0.117j -0.12 -0.056j
 -0.007+0.04j ]
 [-0.098-0.106j -0.638+0.085j -0.395+0.104j 1.81 +1.216j -1.329+1.704j
 1.878-1.188j]
 [ 0.17 +0.018j 0.057+0.8j 0.203+0.411j -1.287-0.708j 1.636+4.126j
 3.24 -1.954j]]
```

We perform additional post-processing to report the operator expansion for each excited state.

```
[ ]: from inquanto.operators import FermionOperator

e_vecs = []
for i in range(e_coeffs_singlets.shape[1]):
    operator = FermionOperator()
    for c, o in zip(e_coeffs_singlets[:, i], fermionic_excitations_singlets):
        operator += c * o
    operator.compress(abs_tol=1e-5)
    print(operator)
    e_vecs.append(operator)

(-0.15565783977198872+0.47621487975804033j, F0^ F0 ), (-0.15565783977198872+0.
 ↪47621487975804033j, F1^ F1 ), (-0.3109942131939058-0.13521605338785778j, F0^ F2 ), ↪
 (continues on next page)
```

(continued from previous page)

$\leftrightarrow (-0.3109942131939058 - 0.13521605338785778j, F1^ F3), (-0.38839307178519544 - 0.0693114600174391j, F0^ F4), (-0.38839307178519544 - 0.0693114600174391j, F1^ F5), (-0.003308852334800071 - 0.002955219215334559j, F2^ F0), (-0.003308852334800071 - 0.002955219215334559j, F3^ F1), (-0.07692730809458342 - 0.1649068223822534j, F2^ F2), \dots$
 $\leftrightarrow (-0.07692730809458342 - 0.1649068223822534j, F3^ F3), (0.16028553390102782 + 0.2040034257419802j, F3^ F5), (-0.006135282730581221 + 0.003137920667681536j, F4^ F0), (-0.006135282730581221 + 0.003137920667681536j, F5^ F1), (-0.09846811137359278 - 0.10620905658730515j, F4^ F2), \dots$
 $\leftrightarrow (-0.09846811137359278 - 0.10620905658730515j, F5^ F3), (0.16993533787082993 + 0.01769895101511043j, F5^ F5) (0.003529209585032674 - 0.00795687921061023j, F0^ F0), (0.003529209585032674 - 0.00795687921061023j, F1^ F1), (-0.24049475157327982 + 0.36783111046288186j, F0^ F2), \dots$
 $\leftrightarrow (-0.24049475157327982 + 0.36783111046288186j, F1^ F3), (-0.6060191423196757 - 0.10482632989727117j, F1^ F5), \dots$
 $\leftrightarrow (0.5763459636232429 + 0.16388842490954422j, F2^ F0), (0.5763459636232429 + 0.16388842490954422j, F3^ F1), (-1.0398193192980176 + 0.6711364842936922j, F2^ F2), \dots$
 $\leftrightarrow 1.0398193192980176 + 0.6711364842936922j, F3^ F3), (0.0799938237057527 + 1.1312076955513257j, F3^ F5), (0.11138656902183633 + 0.3380893725102996j, F4^ F0), (0.11138656902183633 + 0.3380893725102996j, F5^ F1), (-0.6378073443599778 + 0.08473450806257327j, F4^ F2), \dots$
 $\leftrightarrow 0.6378073443599778 + 0.08473450806257327j, F5^ F3), (0.057037317243711416 + 0.7995538108062838j, F5^ F5) (0.002165953897558483 - 0.005944635760931628j, F0^ F0), (0.002165953897558483 - 0.005944635760931628j, F1^ F1), (-0.5667818952627056 + 0.08033281708487372j, F0^ F2), \dots$
 $\leftrightarrow (-0.5667818952627056 + 0.08033281708487372j, F1^ F3), (-0.7806938566797863 + 0.08033281708487372j, F0^ F4), (-0.7806938566797863 + 0.08033281708487372j, F1^ F5), \dots$
 $\leftrightarrow 0.08375225559272712j, F0^ F4), (-0.7806938566797863 + 0.08033281708487372j, F1^ F5), \dots$
 $\leftrightarrow (-0.38358059039903575 + 0.02164733189057025j, F2^ F0), (-0.38358059039903575 + 0.02164733189057025j, F3^ F1), (-0.18951924122622935 - 0.5175643220754953j, F2^ F2), \dots$
 $\leftrightarrow (-0.18951924122622935 - 0.5175643220754953j, F3^ F3), (0.39067878123749633 + 0.14105366840747063j, F3^ F5), \dots$
 $\leftrightarrow 0.3633037576320181 + 0.45991596424530234j, F4^ F0), (0.3633037576320181 + 0.45991596424530234j, F5^ F1), (-0.3948958613818514 + 0.10360791672209882j, F4^ F2), \dots$
 $\leftrightarrow (-0.3948958613818514 + 0.10360791672209882j, F5^ F3), (0.2025074399411661 + 0.41092059234737977j, F5^ F5) (-0.017840047182897188 + 0.022744956740089296j, F0^ F0), (-0.017840047182897188 + 0.022744956740089296j, F1^ F1), (3.806126291783216 + 1.332658212611577j, F0^ F2), (3.806126291783216 + 1.332658212611577j, F1^ F3), (4.621131902658994 + 0.9671995436795001j, F1^ F5), (0.05993966785070938 + 0.04310122860320857j, F2^ F0), (0.05993966785070938 + 0.04310122860320857j, F2^ F1), (0.6441688080331356 + 1.7220627585285808j, F2^ F2), (0.6441688080331356 + 1.7220627585285808j, F3^ F3), (-2.1965755386805927 - 2.655483790470051j, F3^ F5), (0.655483790470051j, F2^ F4), (-2.1965755386805927 - 2.655483790470051j, F3^ F5), (0.1127899167896759 + 0.11659167840761259j, F4^ F0), (0.1127899167896759 + 0.11659167840761259j, F5^ F1), (1.809508151433246 + 1.2160491220244634j, F4^ F2), (1.809508151433246 + 1.2160491220244634j, F5^ F3), (-1.2874605040396814 - 0.7077883650219732j, F5^ F5) (0.0021294584990190677 - 0.01537836611697584j, F0^ F0), (0.0021294584990190677 - 0.01537836611697584j, F1^ F1), (0.8881021827827966 + 0.6217291428756166j, F0^ F2), (0.8881021827827966 + 0.6217291428756166j, F1^ F3), (-1.6634091829941275 - 0.85957877147374j, F1^ F5), (-1.6634091829941275 - 0.85957877147374j, F0^ F4), (-1.6634091829941275 - 0.85957877147374j, F1^ F5), (-0.06072464916897717 + 0.025946744324903372j, F2^ F0), (-0.06072464916897717 + 0.025946744324903372j, F3^ F1), (-1.6336871970371127 + 2.515156393240569j, F2^ F2), (-1.6336871970371127 + 2.515156393240569j, F3^ F2)$

(continues on next page)

(continued from previous page)

```

→ 1.6336871970371127+2.515156393240569j, F3^ F3 ), (0.8787624912303986+3.
→ 2406507977554675j, F2^ F4 ), (0.8787624912303986+3.2406507977554675j, F3^ F5 ), (-0.
→ 12020930798769339-0.055773317529686006j, F4^ F0 ), (-0.12020930798769339-0.
→ 055773317529686006j, F5^ F1 ), (-1.3286485705307103+1.7035098710284213j, F4^ F2 ),_
→ (-1.3286485705307103+1.7035098710284213j, F5^ F3 ), (1.6355067063551598+4.
→ 126046236307394j, F4^ F4 ), (1.6355067063551598+4.126046236307394j, F5^ F5 )
(-0.00020756753326981142-0.0016959811491862975j, F0^ F0 ), (-0.00020756753326981142-0.
→ 0016959811491862975j, F1^ F1 ), (-0.22283094986115085-0.7342082125387235j, F0^ F2 ),
→ (-0.22283094986115085-0.7342082125387235j, F1^ F3 ), (-0.8502303263381792+0.
→ 6025252787670932j, F0^ F4 ), (-0.8502303263381792+0.6025252787670932j, F1^ F5 ), (-
→ 0.004028844274401662-0.054943191515560284j, F2^ F0 ), (-0.004028844274401662-0.
→ 054943191515560284j, F3^ F1 ), (-5.203599573643404+1.5592992892443864j, F2^ F2 ), (-
→ 5.203599573643404+1.5592992892443864j, F3^ F3 ), (-2.2427076382898403+2.
→ 550269786999596j, F2^ F4 ), (-2.2427076382898403+2.550269786999596j, F3^ F5 ), (-0.
→ 006964622239363893+0.040135082617393666j, F4^ F0 ), (-0.006964622239363893+0.
→ 040135082617393666j, F5^ F1 ), (1.878027835548134-1.18788450746501j, F4^ F2 ), (1.
→ 878027835548134-1.18788450746501j, F5^ F3 ), (3.2403778890630868-1.953712459189611j,
→ F4^ F4 ), (3.2403778890630868-1.953712459189611j, F5^ F5 )

```

Alternatively, the Quantum Subspace Expansion algorithm implemented in InQuanto, known as `AlgorithmQSE`, can be employed. This is a black-box method enabling direct retrieval of eigenvalues and eigenvectors, stored as `final_value` and `final_states`, respectively. The algorithm addresses numerical instabilities during matrix diagonalization by eliminating near-linear dependencies in the basis. However, it's worth noting that the user has less control over the specific details of the process. Some examples that cannot be executed include launching, retrieving, and performing PMSV.

```

[ ]: from inquanto.algorithms import AlgorithmQSE

algorithm = AlgorithmQSE(
    computable_qse_matrices=qse,
    parameters=parameters,
)

protocol_expression = PauliAveraging(
    backend=backend,
    shots_per_circuit=1000,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets,
)

algorithm.build(protocol=protocol_expression)

#algorithm.run()

#print(algorithm.generate_report())
<inquanto.algorithms.qse._algorithm_qse.AlgorithmQSE at 0x7fa0534b7850>

```

18.5 Quantinuum H-Series

Although InQuanto is hardware agnostic and can be used with a variety of backends, the use of Quantinuum H-Series devices and emulators is recommended.

18.5.1 Quantinuum System Model H1

The Quantinuum System Model H1 generation quantum computer is packaged with some InQuanto licences and are available over the cloud. The H1 device features 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.

18.5.2 Getting Started

To get started on System Model H1, your organization's administrator needs to add you as a 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 & Conditions for hardware usage.

18.5.3 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

18.5.4 Use

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

18.5.5 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, but may obscure noise-dependent performance characteristics. It also prevents 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.

18.5.6 Access

Examples using running on Quantinuum Hardware are found on the [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>
```

18.5.7 InQuanto Administrators

Administrator Responsibilities

As an administrator, 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.

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.

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 directions for managing credits and tracking usage.

18.6 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.

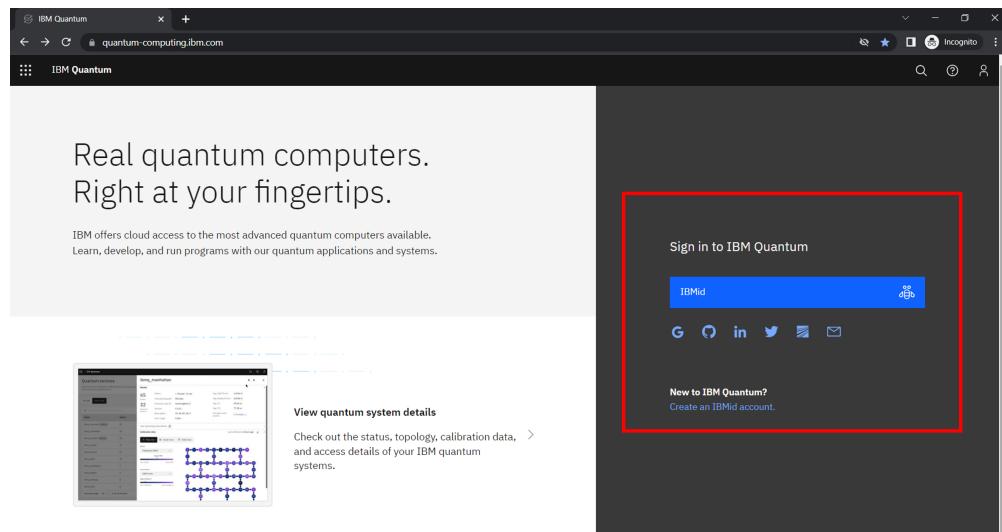
Anyone can make and use IBM resources, but free users will only have access to a few small devices.

To begin, make sure you have both InQuanto and `pytket.extensions.qiskit` in your Python environment (Use `pip install pytket-qiskit` if needed).

18.6.1 Create an IBMQ account

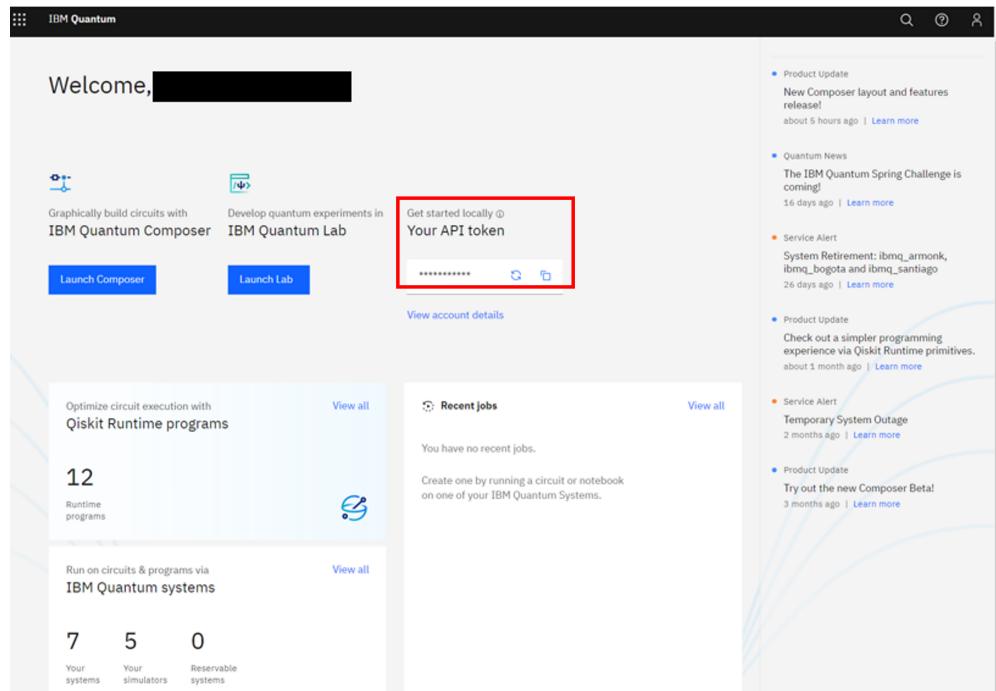
Go to: <https://quantum-computing.ibm.com/>

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:



18.6.2 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 invalidate the old token.

As well as the API token, the user will need to specify a machine to emulate. This is detailed below.

18.6.3 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.

Welcome, [REDACTED]

Graphically build circuits with IBM Quantum Composer

Develop quantum experiments in IBM Quantum Lab

Get started locally @ Your API token

[Launch Composer](#) [Launch Lab](#)

***** View account details

Optimize circuit execution with Qiskit Runtime programs [View all](#)

12 Runtime programs

Recent jobs

You have no recent jobs.

Create one by running a circuit or notebook on one of your IBM Quantum Systems.

Run on circuits & programs via IBM Quantum systems [View all](#)

7 Your systems 5 Your simulators 0 Reservable systems

When on the device page, the list can be filtered to show only devices available to the user.

The screenshot shows the 'Systems' tab of the IBM Quantum Services interface. It displays a grid of seven quantum systems: ibmq_manila, ibmq_bogota, ibmq_santiago, ibmq_quito, ibmq_belem, ibmq_lima, and ibmq_armonk. Each system card provides basic information like Qubits, QV, CLOPS, and Processor type. A red box highlights the top right corner of the grid area, which contains the text "Your systems (7)" with a dropdown arrow and a refresh icon.

System	Qubits	QV	CLOPS	Processor type
ibmq_manila	5	32	2.8K	Falcon r5.11L
ibmq_bogota	5	32	2.3K	Falcon r4L
ibmq_santiago	5	32		Falcon r4L
ibmq_quito	5	16	2.5K	Falcon r4T
ibmq_belem	5	16	2.5K	Falcon r4T
ibmq_lima	5	8	2.7K	Falcon r4T
ibmq_armonk	1	1		Canary r1.2

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.

The image consists of three vertically stacked screenshots from the InQuanto interface.

- Screenshot 1:** A summary card for the machine "ibmq_manila". It shows the following details:

System status	● Online
Processor type	Falcon r5.11L
Qubits	5
QV	32
CLOPS	2.8K

 There is also a small icon of a stylized bird or quill pen in the bottom right corner.
- Screenshot 2:** A detailed view of the machine "ibmq_manila". It provides more specific metrics and provider information:

Qubits	5	Status: ● Online	Avg. CNOT Error: 7.206e-3
QV	32	Total pending jobs: 143 jobs	Avg. Readout Error: 2.866e-2
CLOPS	2.8K	Processor type ⓘ: Falcon r5.11L	Avg. T1: 171.67 us
		Version: 1.0.29	Avg. T2: 56.69 us
		Basis gates: CX, ID, RZ, SX, X	Providers with access: 1 Providers
		Your usage: 0 jobs	Supports Qiskit Runtime: Yes
- Screenshot 3:** A table titled "Your access providers" showing the available provider "ibm-q/open/main":

Provider	Max shots	Max circuits	Max qubits per pulse gate	Max channels per pulse gate	Usage
ibm-q/open/main	20000	100	3	9	View jobs

To get the machine details needed for computing, scroll down or click the 'Providers with access' link.

The image shows two screenshots illustrating how to find provider details.

- Screenshot 1:** The "ibmq_manila" machine details page. A red box highlights the "Providers with access" link under the "Basis gates" section, which points to the "Your access providers" table.
- Screenshot 2:** The "Your access providers" table. A red box highlights the "Provider" column, and a black arrow points from the "Providers with access" link in the first screenshot to this column in the second screenshot.

Provider	Max shots	Max circuits	Max qubits per pulse gate	Max channels per pulse gate	Usage
ibm-q/open/main	20000	100	3	9	View jobs

In your python shell or notebook, the machine details can be set, for example using:

The screenshot shows the IBMQ provider interface. On the left, there is a card for the 'ibmq_manila' machine, which has a red box around its title. The card displays the following details:

5 Qubits	Status: • Online	Avg. CNOT Error: 7.206e-3
32 qv	Total pending jobs: 143 jobs	Avg. Readout Error: 2.866e-2
2.8K CLOPS	Processor type ⓘ: Falcon r5.11L	Avg. T1: 171.67 us
	Version: 1.0.29	Avg. T2: 56.69 us
	Basis gates: CX, ID, RZ, SX, X	Providers with access: 1 Providers ↓
	Your usage: 0 jobs	Supports Qiskit Runtime: Yes

To the right of the machine card is a table titled 'Your access providers' with a red box around its title. The table has three columns: Provider, Max shots, and Max circuits. It contains one row with a red box around the 'Provider' column:

Provider	Max shots	Max circuits
ibm-q/open/main	20000	100

```
from pytket.extensions.qiskit import IBMQEmulatorBackend
backend = IBMQEmulatorBackend(backend_name="ibmq_manila", instance="ibm-q/open/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
NINETEEN

CASE STUDY TUTORIALS - FE4N2

This is a set of tutorials based on modelling nitrogen activation and dissociation on iron clusters. Refer to the [research paper](#) for additional information. The tutorials are divided into three parts: creating optimized active space and Hamiltonian, constructing optimized circuits using ADAPT-VQE, and evaluating these circuits on noisy hardware.

This `InQ_tut_fe4n2_pickles.tar` file contains the *pickle* files for executing these tutorials.

<i>Classical Workflow</i>	Building the Fe4N2 system using AVAS and CASSCF	download
<i>ADAPT-VQE</i>	Building an efficient ansatz using ADAPT-VQE	download
<i>Fe4N2 H-Series</i>	Running Fe4N2 experiments on the H-Series	download

19.1 Fe4N2 - 1 - system construction with AVAS and CASSCF

In this tutorial, which serves as the introductory part of a three-part series ([second part](#) and [third part](#)), we will explore the classical workflow that must be followed before moving on to quantum computations. We will focus on a system that is more complex compared to a simple diatomic molecule like H₂. Additional details regarding the chemical systems, methodologies, and outcomes presented in this tutorial series are available in the associated [research paper](#).

The first step is to run Hartree-Fock (HF) calculations and in our case restricted open-shell Hartree-Fock (ROHF) with the Los Alamos National Laboratory (lanl2dz) effective core potential (ecp) and basis set. We use an InQuanto-PySCF driver to perform the molecular ROHF calculations, and store the results in an InQuanto QubitOperator.

The geometry of Fe₄N₂ has been optimized for several possible spin multiplicities (7, 9, 11 and 13) and we have found that the state with multiplicity (2S+1) 7 has the lowest energy.

Parameters used:

- `geometry` – Molecular geometry.
- `basis` – Atomic basis set valid for Mole class.
- `ecp` – Effective core potentials.
- `charge` – Total charge.
- `multiplicity` – Spin multiplicity, 2S+1.
- `point_group_symmetry` – Enable point group symmetry.
- `soscf` – Use Second-Order SCF solver (Newton's method).
- `verbose` – Control PySCF verbosity.

Even with a very good initial guess, making the Self-consistent field (SCF) procedure converge is sometimes challenging. To achieve quadratic convergence in the orbital optimization, PySCF implements a general second-order solver called the co-iterative augmented hessian (CIAH) method. By setting the `soscf` parameter to True, you can activate this method.

```
[ ]: %reset -f
# native pyscf
from pyscf import gto, scf

geometry = [
["Fe", [ 0.27291543, -1.18421072, -0.99978567]],
["Fe", [-1.35922190, 0.45608027, -0.52389493]],
["Fe", [-0.97901234, -1.08834609, 1.07468517]],
["Fe", [ 0.30675371, 0.72527495, 1.22446007]],
["N", [ 1.32881018, 0.05991762, -0.18585483]],
["N", [ 0.42975492, 1.03128398, -0.58960981]],
]

basis = "lanl2dz"
ecp = {'Fe':'lanl2dz'}
point_group_symmetry = False
multiplicity = 7
charge = 0
verbose = 1

import warnings
warnings.filterwarnings('ignore')
```

```
[ ]: from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularROHF, CASSCF

driver = ChemistryDriverPySCFMolecularROHF(geometry=geometry, basis=basis, ecp=ecp,
                                             charge=charge,
                                             multiplicity=multiplicity, point_group_
                                             symmetry=point_group_symmetry,
                                             soscf=True, verbose=verbose)
driver.run_hf()
-598.4570485072064
```

The NGLView visualizer for InQuanto can be used to display the system.

```
[ ]: from inquanto.geometries import GeometryMolecular
from inquanto.extensions.nglview import VisualizerNGL

fe4n2_geom = GeometryMolecular(geometry)
visualizer = VisualizerNGL(fe4n2_geom)

[ ]: #visualizer.visualize_molecule(atom_labels="index")
```



To limit the hardware resources we can employ a smaller active space using Atomic Valence Active Space ([AVAS](#)) approximation in the PySCF extension of InQuanto. AVAS is a simple and well-defined automated technique for constructing active orbital spaces for use in multi-configuration and multireference electronic structure calculations. Concretely, the technique constructs active molecular orbitals capable of describing all relevant electronic configurations emerging from a targeted set of atomic valence orbitals (e.g., the metal d orbitals in a coordination complex). This is achieved via a linear transformation of the occupied and unoccupied orbital spaces from an easily obtainable single-reference wavefunction (such as from a Hartree-Fock or Kohn-Sham calculations) based on projectors to targeted atomic valence orbitals. More information can be found in the original [paper](#).

Parameters used:

- `aolabels` – AO labels for AO active space.
- `aolabels_vir` – If given, separate AO labels for the virtual orbitals. If not given, `aolabels` is used.
- `n_occ` – None or number of localized occupied orbitals to create. If specified, the value of `threshold` is ignored.
- `n_vir` – None or number of localized virtual orbitals to create. If specified, the value of `threshold_vir` is ignored.
- `force_halves_active` – 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`.
- `freeze_half_filled` – If `True`, all half-filled orbitals (if present) are frozen, i.e. excluded from the AVAS transformation and from the active space.

The reference AO basis for the virtual orbitals is always the computational basis, in contrast to original AVAS but as in the [RE method](#).

For the Fe4N2 cluster if one constructs the active space from (partially) filled Fe d orbitals and the N-N occupied and virtual orbitals, no excitations from Fe 3d to N-N σ or π orbitals are found in the CI wavefunction. We have also separately verified that excitations from the occupied N-N orbitals to the empty Fe-3d orbitals do not contribute. This means that the excitations in the Fe d manifold and in the N-N bond are not coupled and that a reduced active space can be created consisting of orbitals of the dinitrogen system and selected Fe orbitals interacting with it, i.e. excluding all half-filled 3d-like orbitals on Fe.

Here, we directly specify the number of localized occupied orbitals (`n_occ`) and the number of localized virtual orbitals (`n_vir`). Our selection of the occupied and virtual orbitals is influenced by the available computational resources. In this particular problem, we opt for a compact (4,3) active space, which is well-suited for the upcoming quantum calculations involving 6 qubits.

Alternatively, it is also possible to define the values of `threshold` and `threshold_vir`. These represent the truncation thresholds for the AO-projector, above which atomic orbitals (AOs) are retained within the active space for occupied and virtual orbitals, respectively.

```
[ ]: from inquanto.extensions.pyscf import AVAS

ao_pattern = ['N 2p']
ao_pattern_vir = ['N 2p', 'Fe 3d']
avas = AVAS(aolabels=ao_pattern, aolabels_vir=ao_pattern_vir,
            n_occ=2, n_vir=1, force_halves_active=False,
            freeze_half_filled=True, verbose=verbose)
```

We can now create a fresh InQuanto Hamiltonian object using the active space we have defined with AVAS.

Parameters used:

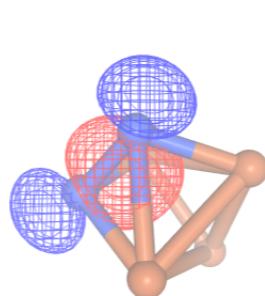
- frozen – Frozen orbital information.
- transf – Orbital transformer.

```
[ ]: driver_AVAS = ChemistryDriverPySCFMolecularROHF(geometry=geometry, basis=basis,
                                                       ↪ecp=ecp, charge=charge,
                                                       ↪group_symmetry=point_group_symmetry,
                                                       ↪multiplicity=multiplicity, point_
                                                       ↪soscf=True, transf=avas, frozen=avas,
                                                       ↪frozenf, verbose=verbose)
```

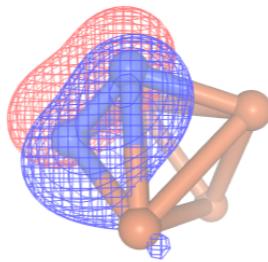
The next step involves performing CASSCF calculations. However, as these calculations tend to be more computationally intensive, it is prudent to visualize the selected active orbitals beforehand. Once more, we will utilize the NGLView visualizer for this purpose.

```
[ ]: orbital_cubes = driver_AVAS.get_cube_orbitals()
orbitals = [visualizer.visualize_orbitals(orb, red_isodevel=-2.0, blue_isodevel=2.0) ↪
           ↪for orb in orbital_cubes]
```

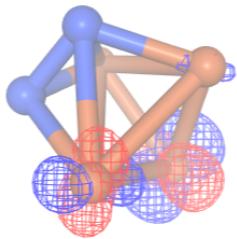
```
[ ]: #orbitals[0]
```



```
[ ]: #orbitals[1]
```



```
[ ]: #orbitals[2]
```



For the CASSCF calculation, we will employ the most minimal strategy, which involves selecting orbitals (and electrons) around the Fermi level to match the user-specified number of orbitals and electrons. More information on running CASSCF with InQuanto-PySCF extension can be found [here](#).

In order to construct the Hamiltonian based on the CASSCF orbitals, it is necessary to create a new, final InQuanto Hamiltonian object. In this instance, we have employed 9 active spatial orbitals and 10 active electrons for the CASSCF calculation. This choice strikes a balance between obtaining an accurate representation of our system and conducting a calculation that does not demand excessive computational resources.

Additionally, in this step we can analyze the configuration interaction (CI) coefficients by setting the `print_ci_coeff` parameter to True. This is very important in deciding the suitability of our chosen active space. Notably, only the first[0], second[1], and ninth[8] active orbitals are significantly active. We can visualize these orbitals using NGLView visualizer. It is important to avoid a scenario where only the d orbitals of the Fe atoms are active, as this would result in a state comprising an equal mixture of 6 configurations distinguished by the placement of electrons within the d orbitals. Expanding the active space to include the orbitals of the nitrogen atoms would lead to an overly large active space, involving a minimum of 10 spatial orbitals.

There is not any standardized approach for selecting the atomic orbital (AO) labels and the number of localized occupied and virtual orbitals. However, the choice depends on the system's characteristics and the allowable size of the active space. We can manipulate various attributes of the AVAS method to achieve the desired outcome, adapting it to the specific needs of the system.

Parameters used:

- `init_orbitals` – Initial orbital coefficients, can be from a different geometry.
- `print_ci_coeff` – If True, the determinants which have coefficients > `ci_print_cutoff` will be printed out.

- ci_print_cutoff – Tolerance for printing CI coefficients.
- max_cycle – Maximum number of CASSCF macroiterations

```
[ ]: import numpy as np

final_driver = ChemistryDriverPySCFMolecularROHF(geometry=geometry, basis=basis,
    ↪ecp=ecp, charge=charge,
                                multiplicity=multiplicity, point_
    ↪group_symmetry=point_group_symmetry,
                                soscf=True, verbose=verbose,
    ↪frozen=avas.frozenf,
                                transf=CASSCF(9, 10, init_
    ↪orbitals=driver_AVAS.get_orbital_coefficients(),
                                print_ci_coeff=True, ci_print_
    ↪cutoff=0.03, max_cycle=100))

hamiltonian, space, state = final_driver.get_system()

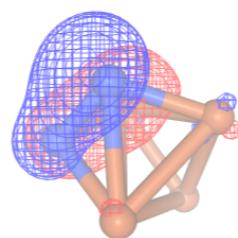
qubit_hamiltonian = hamiltonian.qubit_encode()

** Largest CI components **
[alpha occ-orbitals] [beta occ-orbitals] CI coefficient
[0 1 2 3 4 5 6 7] [0 1] -0.914034639529
[0 1 2 3 4 5 6 7] [0 8] -0.087749343782
[0 1 2 4 5 6 7 8] [0 3] -0.109127674188
[0 1 3 4 5 6 7 8] [0 1] -0.033936544018
[0 1 3 4 5 6 7 8] [0 8] 0.039567732683
[0 2 3 4 5 6 7 8] [0 1] 0.089215434900
[0 2 3 4 5 6 7 8] [0 2] 0.083166002823
[0 2 3 4 5 6 7 8] [0 8] 0.342939298962
[1 2 3 4 5 6 7 8] [1 8] 0.054090558795
```

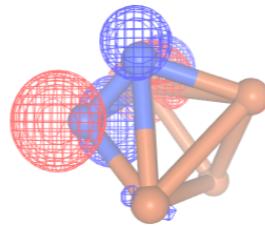
Once more, we employ the NGLView visualizer to display the orbitals using our final InQuanto driver.

```
[ ]: orbital_cubes_final = final_driver.get_cube_orbitals()
orbitals_final = [visualizer.visualize_orbitals(orb, red_isolevel=-2.0, blue_
    ↪isolevel=2.0) for orb in orbital_cubes_final]
```

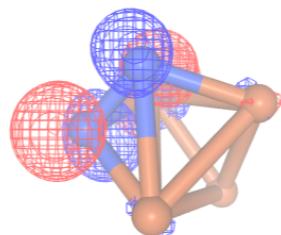
```
[ ]: #orbitals_final[0]
```



```
[ ]: #orbitals_final[1]
```



```
[ ]: #orbitals_final[2]
```



Once the space and state are defined, we can utilize the `print_state` method to showcase the fermionic state and verify the number of qubits required for the quantum calculations.

```
[ ]: space.print_state(state)
```

```
0 0a      : 1
1 0b      : 1
2 1a      : 1
3 1b      : 1
4 2a      : 0
5 2b      : 0
```

As a final step, we can pickle the qubit Hamiltonian, space and state making it convenient for us to import and utilize them in the *second part* of this tutorial.

```
[ ]: import pickle

with open('InQ_tut_fe4n2_space.pickle', 'wb') as handle:
    pickle.dump(space, handle, protocol=pickle.HIGHEST_PROTOCOL)

with open('InQ_tut_fe4n2_state.pickle', 'wb') as handle:
    pickle.dump(state, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

(continues on next page)

(continued from previous page)

```
with open('InQ_tut_fe4n2_qubit_hamiltonian.pickle', 'wb') as handle:
    pickle.dump(qubit_hamiltonian, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

19.2 Fe4N2 - 2 - circuit construction with ADAPT-VQE

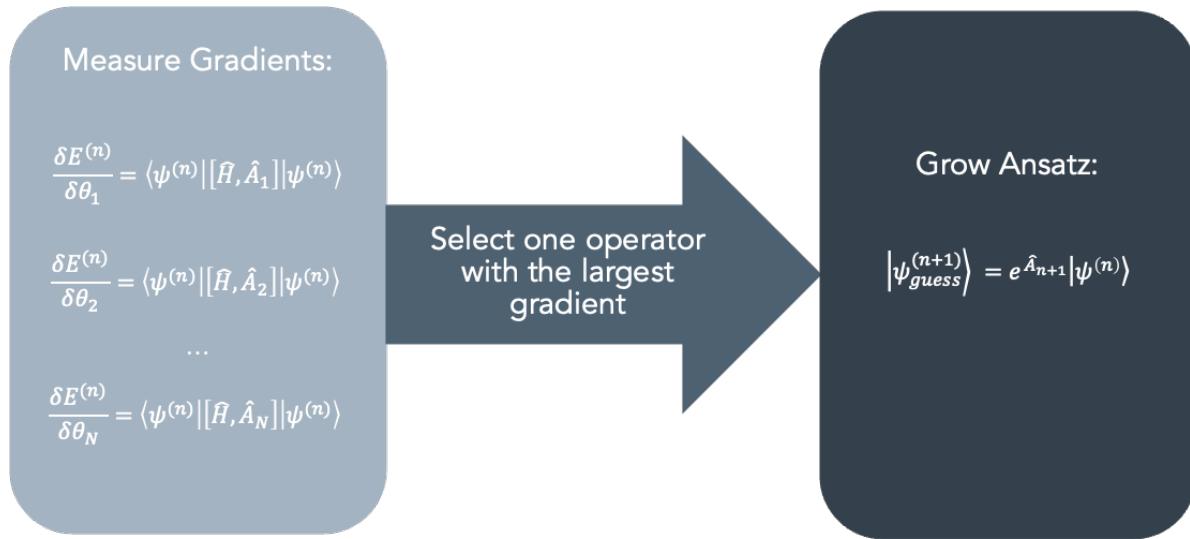
This tutorial aims to introduce the fundamental methodology for conducting quantum chemistry calculations using the Adaptive Variational Quantum Eigensolver (ADAPT-VQE) algorithm. While we provide essential insights into the theory behind this adaptive algorithm, you can access more comprehensive information in the [original paper](#). In this example, our focus will be on calculating the electronic ground state energy of the Fe₄N₂ molecule in a specific geometric configuration. This tutorial builds upon the foundation laid in the first part of a three-part series tutorial, where we covered the classical workflow for defining the chemical system. It is important to note that you should run the [first part](#) before proceeding with this tutorial. The [third part](#) can be executed for thoroughness, but it is not mandatory. For a more thorough understanding of how to perform quantum chemistry calculations on quantum computers using InQuanto, we recommend first reviewing the [VQE tutorial](#).

ADAPT-VQE improves upon the standard VQE algorithm by adaptively selecting gates for optimization, leading to faster convergence, enhanced efficiency, increased accuracy, scalability to larger systems, and improved robustness against noise and errors. These advantages make ADAPT-VQE a valuable tool for quantum chemistry simulations and other quantum computing applications.

Here's a step-by-step explanation of how the ADAPT-VQE algorithm works:

- **Quantum Circuit Ansatz:** ADAPT-VQE starts with the construction of a quantum circuit ansatz. This ansatz represents a parameterized quantum circuit that prepares an approximate wavefunction for the quantum system. The circuit typically consists of a sequence of quantum gates, and the parameters of these gates are adjusted during the optimization process to minimize the energy of the system.
- **Variational Optimization:** The algorithm employs a classical optimizer, such as the gradient descent or the Conjugate Gradient method, to adjust the parameters of the quantum circuit ansatz. The goal is to find the set of parameters that minimizes the expectation value of the system's Hamiltonian with respect to the prepared quantum state.
- **Energy Calculation:** At each optimization step, the quantum circuit is executed on a quantum computer to measure the expectation value of the Hamiltonian. This measurement provides an estimate of the system's energy based on the current quantum state prepared by the circuit.
- **Adaptive Step:** ADAPT-VQE is “adaptive” because it dynamically selects which gates in the quantum circuit to optimize at each step. It identifies which gates have the most significant impact on the energy and focuses optimization efforts on those gates, effectively “adapting” the circuit to improve accuracy efficiently. In this context, the ansatz grows as follows: The operator with the most significant gradient is identified, and it is added to the beginning of the ansatz, accompanied by the introduction of a new variational parameter.
- **Convergence:** The optimization process continues iteratively, adjusting the parameters and adaptively modifying the circuit until a convergence criterion is met. This criterion is typically based on a predefined energy threshold or a maximum number of optimization steps.
- **Ground-State Energy:** Once the optimization converges, ADAPT-VQE provides an estimate of the ground-state energy of the molecular system.

The figure below illustrates a summary of the ansatz growing procedure.



Additional details regarding the chemical systems, methodologies, and outcomes presented in this tutorial series are available in the associated [research paper](#).

To initiate our investigation, we start by importing the qubit Hamiltonian, the state, and the space from the [first part](#).

```
[ ]: import pickle

with open('InQ_tut_fe4n2_state.pickle', 'rb') as handle:
    state = pickle.load(handle)

with open('InQ_tut_fe4n2_qubit_hamiltonian.pickle', 'rb') as handle:
    qubit_hamiltonian = pickle.load(handle)

with open('InQ_tut_fe4n2_space.pickle', 'rb') as handle:
    space = pickle.load(handle)
```

We are going to use our InQuanto **ADAPT** (AlgorithmFermionicAdaptVQE) algorithm for our calculations. The Fermionic ADAPT method being utilized is essentially an adaptation of the standard ADAPT algorithm. It has been customized to effectively approximate the ground-state energy of fermionic systems.

Parameters used:

- pool – Holds the pool of Pauli terms which go the TrotterAnsatz object.
- state – Initial (para-)fermionic reference state for the chemical system in question (usually the HF determinant).
- hamiltonian – The Hermitian operator to measure for the lowest eigenvalue.
- minimizer – Variational Minimizer to use for the ADAPT experiment.
- tolerance – Expectation value of commutation between pool and Hamiltonian at which loop is stopped.
- disp – If the algorithm should display variational data every iteration.

Tolerance values do not have universally defined standards and often rely on practical experience. As indicated in the original ADAPT paper, even when employing a stricter threshold, such as 0.001, the ADAPTVQE algorithm does not terminate prematurely. It consistently produces highly accurate results, even when dealing with strongly correlated systems.

Since we are performing an ADAPT-VQE experiment, we must construct a pool of exponents from which we will select only those with the largest gradients during the optimization step(s). Here, we have constructed a pool of excitation

operators which correspond to a UCCSD ansatz. The [InQuanto webpage](#) provides additional details on the process of generating single and double excitations based on a reference fermion state.

An alternative approach involves employing a unitary coupled-cluster (UCC) ansatz referred to as k-UpCCGSD. This ansatz relies on a family of sparse generalized doubles operators, offering an efficient and progressively refinable unitary coupled-cluster wavefunction suitable for implementation on near-term quantum computers. k-UpCCGSD incorporates k products of the exponential of pair-coupled-cluster double excitation operators, in conjunction with generalized single excitation operators. In some cases, utilizing k-UpCCGSD with k=1 (1 layer) results in shallower circuits compared to UCCSD. You can easily experiment with this approach in the tutorial by commenting out the UCCSD ansatz and uncommenting the k-UpCCGSD ansatz. Further details can be found on the [InQuanto webpage](#).

```
[ ]: #UCCSD
exponent_pool = space.construct_single_ucc_operators(state)
exponent_pool += space.construct_double_ucc_operators(state)
```

Alternatively we could construct a larger exponent pool using the generalized forms:

```
exponent_pool = space.construct_generalised_single_ucc_operators()
exponent_pool = space.construct_generalised_pair_double_ucc_operators()
```

Since we are conducting a variational experiment, it is essential to opt for a minimization technique. In this context, we have decided to utilize the L-BFGS-B minimization algorithm provided by SciPy. `MinimizerScipy` is an InQuanto wrapper for Scipy minimization routines.

```
[ ]: from inquanto.minimizers import MinimizerScipy

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

[ ]: from inquanto.algorithms import AlgorithmFermionicAdaptVQE

adapt = AlgorithmFermionicAdaptVQE(pool=exponent_pool, state=state, hamiltonian=qubit_
    ↪hamiltonian,
                                         minimizer=scipy_minimizer, tolerance=1e-3 , ↪
    ↪disp=True)
```

The ADAPT-VQE algorithm will be run on the [Qulacs statevector simulator](#) to build the ansatz and determine its parameters.

Moving forward, we must create the algorithm object, which involves specifying a protocol object. This protocol dictates how the objective function is assessed. In this case, we select a state-vector protocol (`SparseStatevectorProtocol`) since we have selected a state-vector backend.

The `build` method constructs the ADAPT algorithm by following these protocols:

- `protocol_expectation` – The protocol used for expectation value calculation.
- `protocol_pool_metric` – The protocol used to determine the excitation selection metric.
- `protocol_gradient` – The protocol used for gradient calculation.

Subsequently, we utilize the `run` method to carry out the ADAPT experiment. Upon completion, you will have access to a `TrotterAnsatz` instance, the optimized energy, and the final parameter values.

IPython's `%%capture` cell magic captures `stdout/stderr` and can be configured to either discard or store them. It defaults to discarding, offering a simple way to suppress unwanted output. Comment the command to display all outputs.

```
[ ]: %%capture
```

(continues on next page)

(continued from previous page)

```

from pytket.extensions.qulacs import QulacsBackend
from inquanto.protocols.statevector._sparse_sv import SparseStatevectorProtocol
from inquanto.operators import FermionOperatorList

backend = QulacsBackend()

adapt_protocol = SparseStatevectorProtocol(backend)
adapt.build(
    protocol_expectation=adapt_protocol,
    protocol_pool_metric=adapt_protocol,
    protocol_gradient=adapt_protocol
)

adapt.run()

```

In InQuanto, each algorithm object is equipped with a useful method called `generate_report`. This method serves the purpose of retrieving specific quantities of interest. However, if you're interested in accessing specific results, you can simply use the relevant key, such as `'final_value'` to obtain the ground state energy.

```
[ ]: results = adapt.generate_report()
print("Minimum Energy: {}".format(results["final_value"]))

Minimum Energy: -598.5059938316066
```

The `final_parameters` property provides the optimized ansatz parameters, while the `get_exponents_with_symbols` method returns a sublist of the fermionic pool operator list containing symbols.

```
[ ]: gs_parameters = adapt.final_parameters
exponents_with_symbols = adapt.get_exponents_with_symbols()
print(gs_parameters.df())

   ordering symbol      value
0          0    d0 -0.059984
1          1    d1  0.006644
2          2    d2  0.006712
3          3    d3 -0.347714
4          4    s0 -0.002197
5          5    s1 -0.002342
6          6    s3 -0.005704
```

We can pickle both the ground state parameters and the list of fermionic pool operators. This allows for easy import and utilization in the *third part* of this tutorial.

```
[ ]: with open('InQ_tut_fe4n2_gs_parameters.pickle', 'wb') as handle:
    pickle.dump(gs_parameters, handle, protocol=pickle.HIGHEST_PROTOCOL)

with open('InQ_tut_fe4n2_exponents_with_symbols.pickle', 'wb') as handle:
    pickle.dump(exponents_with_symbols, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

The Fermionic ADAPT algorithm features a `get_ansatz` function that generates an ansatz circuit built from symbols-containing operators of the fermionic pool. By default, it utilizes an efficient ansatz circuit compilation approach employing the `FermionSpaceStateExpChemicallyAware` class. This approach helps minimize the computational resources needed.

```
[ ]: from inquanto.ansatzes import FermionSpaceStateExpChemicallyAware

ansatz= adapt.get_ansatz(state=state, fermion_ansatz_
    ↪type=FermionSpaceStateExpChemicallyAware)

print(ansatz.state_circuit)

<tket::Circuit, qubits=6, gates=311>
```

The `state_circuit` property in InQuanto represents the symbolic state circuit, which comes with a default compilation. If you are working in a Jupyter environment, you can visualize this symbolic state circuit by utilizing the `render_circuit_jupyter` function from the `pytket` library.

```
[ ]: from pytket.circuit.display import render_circuit_jupyter

render_circuit_jupyter(ansatz.state_circuit)

<IPython.core.display.HTML object>

<tket::Circuit, qubits=6, gates=311>
```

To conclude the analysis, you can display additional information about the circuit by showing the total number of qubits, parameters, and CNOT gates it contains. In this circuit, there are a total of 6 qubits, 7 parameters, and 59 CNOT gates. Notably, the multi-qubit gates, specifically the CNOT gates, are the primary contributors to noise within this gate decomposition.

```
[ ]: from pytket.circuit import OpType

n(cx)_sum = lambda circuit: sum([1 for com in circuit if com.op.type == OpType.CX])

print(f"# of qubits: {ansatz.n_qubits}")
print(f"# of symbols: {ansatz.n_symbols}")
print(f"# of CX gates: {n(cx)_sum(ansatz.state_circuit)}")

# of qubits: 6
# of symbols: 7
# of CX gates: 59
```

19.3 Fe4N2 - 3 - calculations on the H-series

This part marks the conclusion of the three-part series tutorial, and the user will learn how to conduct both emulator and hardware calculations using Quantinuum H-Series hardware on a system more complex than *H2*. It is important to note that one should run the [first part](#) and [second part](#) before proceeding to this tutorial. Additional details regarding the chemical systems, methodologies, and outcomes presented in this tutorial series are available in the associated [research paper](#).

Let us move beyond a simple single-point energy calculation that is neither optimized nor variationally solved. Instead, we build upon the results obtained in the second part of this tutorial series, where the Adaptive Variational Quantum Eigensolver (ADAPT-VQE) algorithm was used to compute the total energy of the Fe_3N_2 molecule. This tutorial is also an extension of the initial part of the series, which provides detailed insights into the classical workflow necessary for defining the system.

To carry out these calculations, access to the Quantinuum systems is essential, and this access must be granted by the respective InQuanto administrator. Additionally, the user will require the appropriate credentials to run computations on a 6-qubit machine. Instructions on how to obtain access and in general more information about Quantinuum H-Series can be found on the linked [page](#) (for administrators).

Here are the steps outlined:

- Import the parameters characterizing the ground-state wavefunction.
- Conduct calculations on the Quantinuum emulator utilizing the PMSV error mitigation method.
- Execute calculations on the Quantinuum hardware using the PMSV error mitigation method.

To begin with, let us start by importing the qubit Hamiltonian, the fermionic state and space from the first part, along with the ground state parameters and the list of fermionic pool operators from the *second part* of this tutorial.

```
[ ]: import pickle

with open('InQ_tut_fe4n2_qubit_hamiltonian.pickle', 'rb') as handle:
    qubit_hamiltonian = pickle.load(handle)

with open('InQ_tut_fe4n2_state.pickle', 'rb') as handle:
    state = pickle.load(handle)

with open('InQ_tut_fe4n2_space.pickle', 'rb') as handle:
    space = pickle.load(handle)

with open('InQ_tut_fe4n2_gs_parameters.pickle', 'rb') as handle:
    gs_parameters = pickle.load(handle)

with open('InQ_tut_fe4n2_exponents_with_symbols.pickle', 'rb') as handle:
    exponents_with_symbols = pickle.load(handle)

import warnings
warnings.filterwarnings('ignore')
```

InQuanto employs an efficient ansatz circuit compilation approach, provided by the `FermionSpaceStateExpChemicallyAware` class. This method is designed to minimize the computational resources required.

Parameters used:

- `fermion_operator_exponents` – Contains exponents and symbols. Assuming input exponents are ordered as single exponents first, followed by double exponents.
- `fermion_state` – Initial fermionic reference state.

```
[ ]: from inquanto.ansatzes import FermionSpaceStateExpChemicallyAware

ansatz=FermionSpaceStateExpChemicallyAware(fermion_operator_exponents=exponents_with_
    ↴symbols, fermion_state=state)
```

The `state_circuit` attribute within InQuanto represents the symbolic state circuit, complete with a default compilation. In a Jupyter environment, one can visualize this symbolic state circuit by making use of the `render_circuit_jupyter` function provided by the `pytket` library.

```
[ ]: from pytket.circuit.display import render_circuit_jupyter

render_circuit_jupyter(ansatz.state_circuit)
<IPython.core.display.HTML object>
```

To obtain a concise overview of the quantum resource costs associated with creating the Ansatz, one can utilize the `generate_report` method of the Ansatz object. This report typically includes information such as the circuit depth, total gate count, the count of Ansatz parameters, and the number of required qubits. In this circuit, the circuit depth is

133 and there are a total of 311 gates, 7 parameters and 6 qubits. These metrics provide an overview of the complexity and requirements of the quantum circuit. The circuit depth is the length of the longest path from the input (or from a preparation) to the output (or a measurement gate), moving forward in time along qubit wires.

For more in-depth diagnostics and analysis of the circuit, the user can examine the tket circuit object directly, which is usually accessed through `ansatz.state_circuit`. As an example, one can count the number of CNOT gates in the circuit. For further details on how to analyze tket circuits, the interested reader can refer to the [tket documentation](#), which provides comprehensive information and guidance on this topic.

```
[ ]: from pytket.circuit import OpType

print('ANSATZ REPORT:')
print(ansatz.generate_report())
print(ansatz.get_circuit())
print('\nCNOT GATES: {}'.format(ansatz.state_circuit.n_gates_of_type(OpType.CX)))

ANSATZ REPORT:
{'ansatz_circuit_depth': 133, 'ansatz_circuit_gates': 311, 'n_parameters': 7, 'n_qubits': 6}
<tket::Circuit, qubits=6, gates=311>

CNOT GATES: 59
```

Conducting emulator experiments prior to hardware experiments is a pivotal phase in the development and optimization of quantum algorithms and applications. Emulators offer a controlled setting with the possibility to refine algorithms, explore quantum error correction techniques, and gain valuable insights into the performance of quantum circuits, all without being restricted by the limitations of physical hardware.

To simulate the specific noise profiles of machines, one can load and apply them to the simulations using the `QuantinuumBackend`, which retrieves information from the user's Quantinuum account. The `QuantinuumBackend` offers a range of available emulators, such as H1-1E and H1-2E. These emulators are designed for specific devices and they run remotely on a server.

Parameters used:

- `device_name` – Name of device, e.g. “H1-1”
- `label` – Job labels used if Circuits have no name, defaults to “job”
- `group` – String identifier of a collection of jobs, can be used for usage tracking.

```
[ ]: from pytket.extensions.quantinuum import QuantinuumBackend

backend = QuantinuumBackend(device_name="", label = "", group ="") #, label=""
```

To reduce errors and inaccuracies caused by quantum noise and imperfections in the Quantinuum device, one can employ noise mitigation techniques. In this case, let us define the Qubit Operator symmetries within the system, which enables to utilize `PMSV` (Partition Measurement Symmetry Verification). PMSV is an efficient technique for symmetry-verified quantum calculations. It represents molecular symmetries using Pauli strings, including mirror planes (`Z2`) and electron-number conservation (`U1`). For systems with Abelian point group symmetry, qubit tapering methods can be applied. PMSV uses commutation between Pauli symmetries and Hamiltonian terms for symmetry verification. It groups them into sets of commuting Pauli strings. If each string in a set commutes with the symmetry operator, measurement circuits for that set can be verified for symmetry without additional quantum resources, discarding measurements violating system point group symmetries.

Parameters used:

`stabilizers` – List of state stabilizers as `QubitOperators` with only a single pauli strings in them.

The InQuanto `symmetry_operators_z2_in_sector` function is employed to retrieve a list of symmetry operators applicable to the system under consideration. These symmetry operators are associated with the point group, spin parity, and particle number parity Z2 symmetries that uphold a specific symmetry sector. The users can find additional details in the linked [page](#).

```
[ ]: from inquanto.protocols.averaging._mitigation import PMSV
from inquanto.mappings import QubitMappingJordanWigner

stabilizers = QubitMappingJordanWigner().operator_map(
    space.symmetry_operators_z2_in_sector(
        state
    )
)

mitms_pmsv = PMSV(stabilizers)
```

Here, a technique known as “batching” is employed, wherein each experiment is iterated a certain number of times with a specific shot count as the target. For instance, if one aims for 10,000 shots, one conducts the experiment ten times, resulting in a total of 100,000 shots for that experiment. This limitation exists because each experiment can only accommodate a maximum of 10,000 shots and this implementation helps to avoid a single task from monopolizing the system’s resources or a user accidentally using all their credits in a single instance.

To compute the expectation value of a Hermitian operator through operator averaging on the system register, the Pauli-Averaging protocol is employed. This protocol effectively implements the procedure outlined in [‘Hamiltonian Averaging’](#). To launch the circuits to the backend the `launch` function is used. This method processes all the circuits associated with the expectation value calculations and returns a list of `ResultHandle` objects representing the handles for the results. One can pickle these `ResultHandle` objects so that the results can be easily retrieved. The user can monitor the progress of the experiments on the [Quantinuum page](#) by using the same credentials used to execute the experiments.

```
[ ]: from inquanto.protocols import PauliAveraging
from pytket.partition import PauliPartitionStrat

set_shots_10k = [10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, ↴
                 100000]
repeats = 10

#build and compile the circuits once
protocol_template = PauliAveraging(
    backend,
    shots_per_circuit=10000,
    pauli_partition_strategy=PauliPartitionStrat.CommutingSets
)
protocol_template.build(gs_parameters, ansatz, qubit_hamiltonian, noise_
    ↴mitigation=mitms_pmsv)
protocol_pickle=protocol_template.dumps()
protocol_template.n_circuit

# launch 10 repeats of these circuits
for i in range(repeats):
    protocol= PauliAveraging.loads(protocol_pickle, backend)
    handles = protocol.launch()
    with open( "handles_" + str(i) + ".pickle", 'wb') as handle:
        pickle.dump(handles, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

After the experiments have finished, one can obtain the results by utilizing the `retrieve` function, which retrieves distributions from the backend for the specified source. The expectation value of a kernel for a specified quantum state is calculated by using the `evaluate_expectation_value` function. In addition, the `evaluate_expectation_uvalue` function can be used to calculate the expectation value of the Hermitian kernel while considering linear error propagation theory. The `std_dev` property returns the standard deviation which is used as the error associated with the calculation.

```
[ ]: repeats = 10
emulator_energies_10k = []
emulator_10k = []

for i in range(repeats):
    with open("handles_" + str(i) + ".pickle", 'rb') as handle:
        handles = pickle.load(handle)
    #only need 1 copy of protocol to eval sets of results
    protocol.retrieve(handles)
    energy_value = protocol.evaluate_expectation_value(ansatz, qubit_hamiltonian)
    emulator_energies_10k.append(energy_value)
    error= protocol.evaluate_expectation_uvalue(state, qubit_hamiltonian)
    emulator_10k.append(error)
```

Additionally, one can showcase circuit statistics after employing the `get_compiled_circuit` function to compile the sequence of circuits. In this case, both the circuit depth for each circuit measurement and the circuit depth associated with 2-qubit gates are visualized.

```
[ ]: for meas_circ in protocol.get_circuits():
    print("Circuit depth =", backend.get_compiled_circuit(meas_circ).depth())
    print("2qb gate depth =", backend.get_compiled_circuit(meas_circ).depth_by_type(
        →{OpType.CX, OpType.CZ}))
```

After completing the 10 repetitions, the mean value is computed. The mean value is used to determine the energy values for shot counts such as 20k, 30k, and so forth. Meanwhile, the standard deviation is used to calculate the error bars.

```
[ ]: import numpy as np

emulator_10k_mean = []
for i in range (1,11):
    emulator_10k_mean.append(np.mean(emulator_energies_10k[:i]))

emulator_10k_std= []
for i in range (1,11):
    emulator_10k_std.append(numpy.std(emulator_10k[:i]))
```

This procedure can be reiterated for various sets of shots to analyze how the number of shots impacts the energy and error estimations.

```
[ ]: #target:5k
set_shots_5k = [5000, 10000, 15000, 20000, 25000, 30000, 35000, 40000, 45000, 50000]

emulator_5k_mean = [-598.5233976942287, -598.528748007567, -598.5254365672856, -598.
    ↪5266267836435, -598.5252272612659,
    ↪-598.5241590945266, -598.5255975795899, -598.525113940237, -598.
    ↪5249076903503, -598.523958110884]

emulator_5k_std =[ 0, 0.0016, 0.0015, 0.0014, 0.0013, 0.0012, 0.0011, 0.0010, 0.00095, 0.
```

(continues on next page)

(continued from previous page)

```

↪0009]

#target:4k
set_shots_4k = [4000, 8000, 12000, 16000, 20000, 24000, 28000, 32000, 36000, 40000]

emulator_4k_mean = [-598.5261956279538, -598.5239748592571, -598.5250271719032, -598.
↪5236355685819, -598.5232948710519,
                   -598.5243534836842, -598.5244650328887, -598.5251092284366, -
↪598.524141170795, -598.5235882519817]

emulator_4k_std =[0,0.00163,0.00154,0.0015,0.0014,0.0013,0.0012,0.00115,0.0011, 0.
↪0010]

#target:2.5k
set_shots_2_5k = [2500, 5000, 7500, 10000, 12500, 15000, 17500, 20000, 22500, 25000]

emulator_2_5k_mean = [-598.5386462829877, -598.528495041739, -598.5223780085176, -598.
↪5221927569415, -598.5220793440851,
                   -598.522950988168, -598.5245581694165, -598.5236303692766, -598.
↪5243398619134, -598.5237327682132]

emulator_2_5k_std =[0,0.0017,0.00165,0.0016,0.00155,0.0015,0.00145,0.0014,0.00135,0.
↪0013]

```

Once collected all the results, one can proceed to visualize and analyze the data. By running the same set of instructions for different shot configurations, one can observe that the error bars notably decrease when a sufficiently large number of samples is used.

```

[ ]: import matplotlib.pyplot as plt
      import seaborn as sns

      sns.set_style("whitegrid")

      #plt.plot(set_shots_10k, emulator_10k_mean, label='emulator_10k',color='red')
      plt.plot(set_shots_5k, emulator_5k_mean, label='emulator_5k',color='black')
      plt.plot(set_shots_4k, emulator_4k_mean, label='emulator_4k',color='teal')
      plt.plot(set_shots_2_5k, emulator_2_5k_mean, label='emulator_2.5k',color='orange')

      plt.rcParams["figure.figsize"] = (10,6)
      plt.xticks(fontsize=15 )
      plt.yticks(fontsize=15 )

      #y_error_10k = emulator_10k_std
      y_error_5k = emulator_5k_std
      y_error_4k = emulator_4k_std
      y_error_2_5k = emulator_2_5k_std

```

(continues on next page)

(continued from previous page)

```
#plt.errorbar(set_shots_10k, emulator_10k_mean,
#              yerr = y_error_10k, fmt ='s', color='red',
#              elinewidth=1,capsize=2)

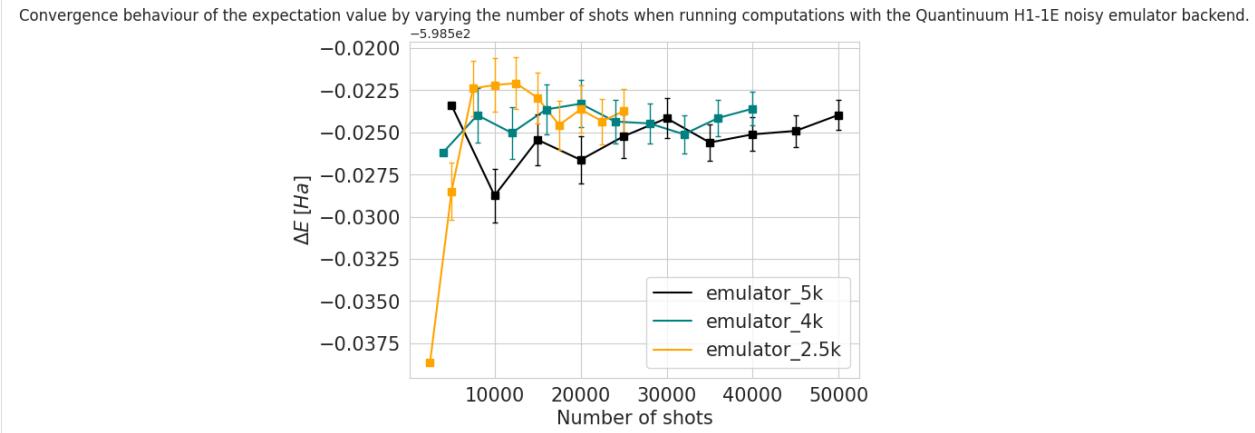
plt.errorbar(set_shots_5k, emulator_5k_mean,
             yerr = y_error_5k, fmt ='s', color='black',
             elinewidth=1,capsize=2)

plt.errorbar(set_shots_4k, emulator_4k_mean,
             yerr = y_error_4k, fmt ='s', color='teal',
             elinewidth=1,capsize=2)

plt.errorbar(set_shots_2_5k, emulator_2_5k_mean,
             yerr = y_error_2_5k, fmt ='s', color='orange',
             elinewidth=1,capsize=2)

plt.xlabel('Number of shots',fontsize=15)
plt.ylabel('$\Delta E \backslash; [Ha]$',fontsize=15)
plt.title("Convergence behavior of the expectation value by varying the number of shots when running computations with the Quantinuum H1-1E noisy emulator backend.")
plt.legend(fontsize=15)

<matplotlib.legend.Legend at 0x7fe543b92b00>
```



With InQuanto, it is extremely easy to go from hardware emulation to hardware experiment. This is because `pytket.extensions.quantinuum` requires the same details for emulating the hardware as for sending the experiment to the physical quantum circuit, the only difference being the substitution of the name of the quantum device. For example, from 'H1-1E' to 'H1-1'.

Here, only one series of experiments was performed on the hardware device. To limit the effects of noise in estimating expectation values, the PMSV method was used.

In this tutorial, three hardware experiments for three different structures of Fe₄N₂ cluster were performed in order to calculate the activation and dissociation energies of nitrogen on iron cluster. More specifically, in this tutorial, the emulator calculations with 4000 shots were used. In addition, a hardware experiment was conducted simply by changing 'H1-1E' to 'H1-1', and these results served as the starting points in the figure below. Subsequently, one can re-run the calculations described in this tutorial, altering only the initial geometry in the first part. Namely, one can use the geometry where the N-N bond is stretched, representing the transition state and denoted as 'ts' in the figure. Finally, one can repeat the three tutorials once again, using a geometry where the N-N bond breaks, denoted as the 'end' in the figure. Notice that in these six experiments (two for each geometry), the single experiments with both the emulator and hardware were used

instead of the batching method described above. The primary goal of this tutorial is to offer a deeper understanding of how quantum computers can contribute to the comprehension of dissociation processes.

The electronic activation energy with respect to the kinetic constant of the process is calculated as $E_{Fe/N_2}^{tot} - E_{Fe/N_2^*}^{tot}$, while the activation energy is $E_{Fe/N_2}^{tot} - E_{Fe/2N}^{tot}$.

The data points computed on the hardware are in excellent agreement with the emulation results.

```
[ ]: # these are example solved energies from hardware and emulator

sns.set_style("whitegrid")

steps=[1,2,3]
emulator=[0,0.04681999999991149,-0.02102999999996107 ]
hardware=[0,0.0623299999997446,-0.01869999999967076]

plt.hlines(y=0, xmin=0, xmax=12, ls='--', colors='black')

plt.scatter(x=steps, y=hardware, s=200, c="black", label="Hardware")
plt.plot(steps, hardware, c="black")

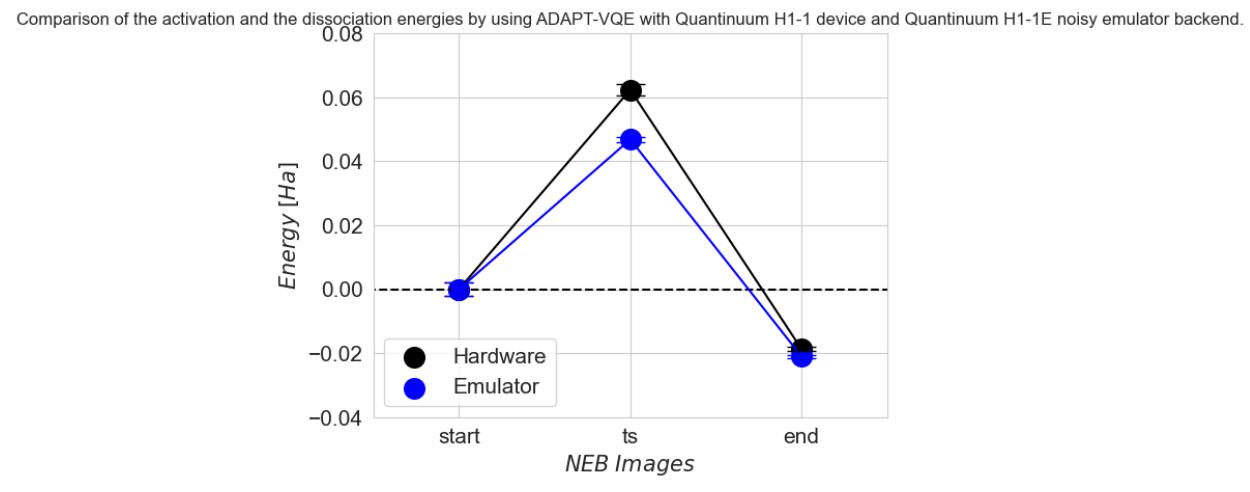
y_error_hard = [0.00196006200237305,0.00171393177265877,0.000768517902650537]
plt.errorbar(steps, hardware,
             yerr = y_error_hard, fmt ='o', color='black',
             elinewidth=10,capsize=10)

plt.scatter(x=steps, y=emulator, s=200, c="blue", label="Emulator")
plt.plot(steps, emulator, c="blue")

y_error_em = [0.00222116960181585,0.000835916404261285,0.000489635318758811]

plt.errorbar(steps, emulator,
             yerr = y_error_em, fmt ='o', color='blue',
             elinewidth=10,capsize=10)

plt.ylabel('$Energy\; [Ha]$', fontsize=15)
plt.xlabel('$NEB \; Images$', fontsize=15)
plt.xticks(fontsize=15 )
plt.yticks(fontsize=15 )
plt.xlim([0.5, 3.5])
plt.ylim([-0.04, 0.08])
x=[1,2,3]
my_xticks = ['start','ts','end']
plt.xticks(x, my_xticks)
plt.legend(fontsize=15, loc='lower left')
plt.title("Comparison of the activation and the dissociation energies by using ADAPT-  
→VQE with Quantinuum H1-1 device and Quantinuum H1-1E noisy emulator backend.")
plt.show()
```



FRAGMENTATION TUTORIALS

These tutorials demonstrate how to construct small subsystems which can be evaluated with wave function / quantum computational methods whilst interacting with a larger environment.

DMET	Tackling larger systems with fragmentation	download
WF-in-DFT	Projection-based embedding	download
NEVPT2 + AC0 corrections	NEVPT2 and AC0 energy corrections	download
WFT-in-DFT + NEVPT2 or AC0	Projection-based embedding with energy corrections	download

20.1 Tackling larger systems with fragmentation

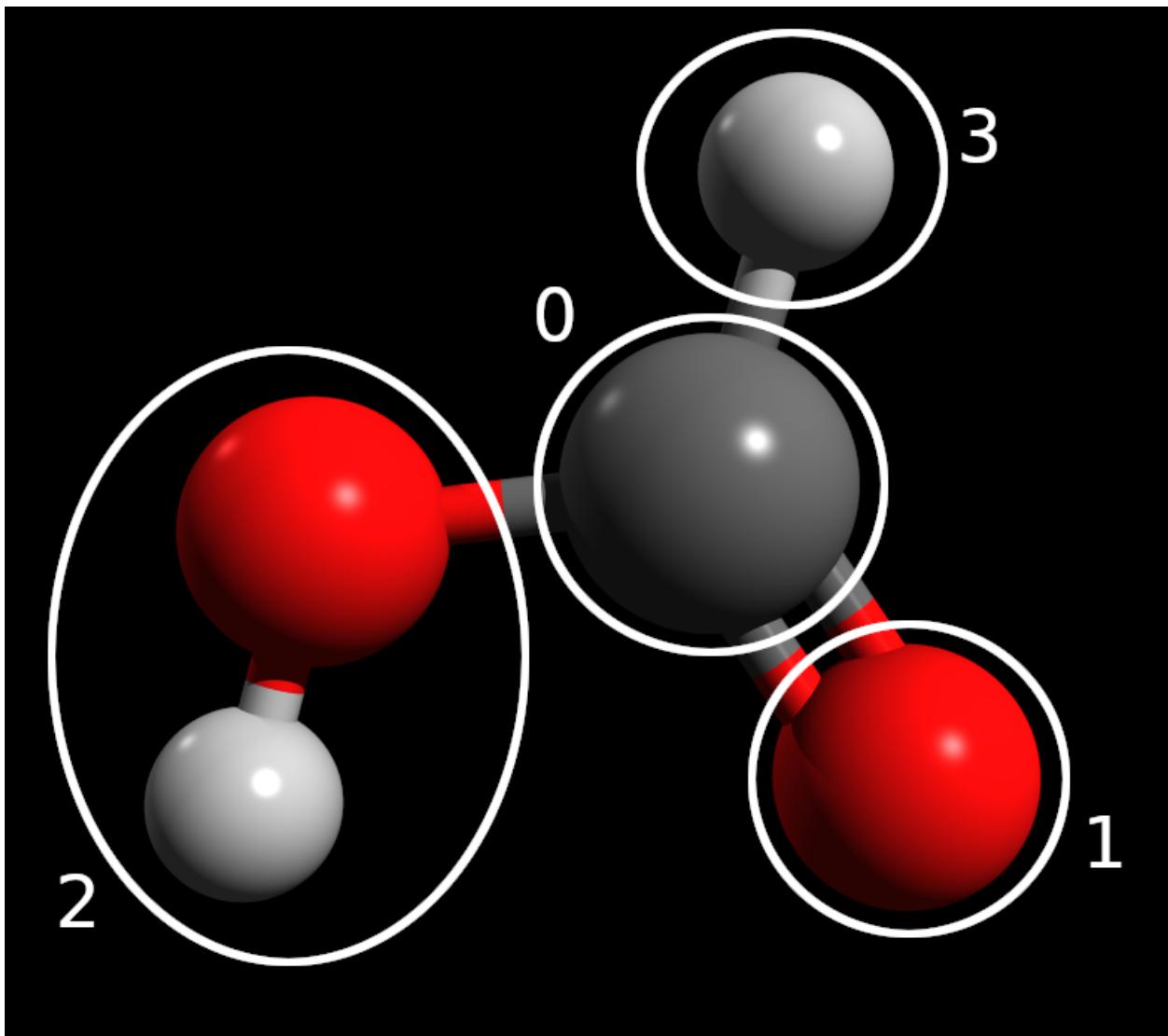
In the *basic VQE* and *extended VQE* tutorials, we covered how to run a simple VQE calculation using InQuanto and some optimizations that can be performed. Here, we look at using Density Matrix Embedding Theory (DMET) to examine a larger system by fragmenting it. As an example, we consider HCOOH (formic 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 than 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 `inquito-pyscf` extension.

```
[ ]: from inquito.geometries import GeometryMolecular
from inquito.embeddings import DMETRHF
from inquito.extensions.pyscf import ChemistryDriverPySCFMolecularRHF
from inquito.extensions.pyscf import DMETRHFFragmentPySCFCCSD, get_fragment_orbital_
    ↪masks, get_fragment_orbitals
from pytket.extensions.qiskit import AerStateBackend

~/lib/python3.11/site-packages/pyscf/dft/libxc.py:771: UserWarning: Since PySCF-2.3, ↪
    ↪B3LYP (and B3P86) are changed to the VWN-RPA variant, corresponding to the original ↪
    ↪definition by Stephens et al. (issue 1480) and the same as the B3LYP functional in ↪
    ↪Gaussian. To restore the VWN5 definition, you can put the setting "B3LYP_WITH_VWN5= True" in pyscf_conf.py
    warnings.warn('Since PySCF-2.3, B3LYP (and B3P86) are changed to the VWN-RPA variant, '
```

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]],
        ['O', [1.171, 0.120, 0.000]],
        ['H', [-0.389, 1.475, 0.000]],
        ['H', [-0.609, -1.355, 0.000]]]
```

(continues on next page)

(continued from previous page)

```

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 initialization 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 orthogonalization 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>=-186.31338432615368 EFR=-58.83801802368312 Q=0.  
↪ 020105631085688636  
# FRAGMENT 1 - O: <H>=-186.31338532077353 EFR=-94.07923540836005 Q=-0.  
↪ 08915722472857546  
# FRAGMENT 2 - OH: <H>=-186.29269508290201 EFR=-98.21603860323467 Q=-0.  
↪ 07076155306003251  
# FRAGMENT 3 - H: <H>=-186.23216575139472 EFR=-3.8054069201359595 Q=0.  
↪ 002986786186124024  
# NEWTON ITERATION - CHEMICAL POTENTIAL 0.0001  
# FRAGMENT 0 - C: <H>=-186.31396619851338 EFR=-58.83898988857606 Q=0.  
↪ 020314471024403424  
# FRAGMENT 1 - O: <H>=-186.31419656704082 EFR=-94.0797095185983 Q=-0.  
↪ 08903855658752668  
# FRAGMENT 2 - OH: <H>=-186.29359289137557 EFR=-98.21647309737236 Q=-0.  
↪ 07064482022362562  
# FRAGMENT 3 - H: <H>=-186.23226272837812 EFR=-3.805683950277535 Q=0.  
↪ 003066335733261094  
# NEWTON ITERATION - CHEMICAL POTENTIAL 0.02612234664464692  
# FRAGMENT 0 - C: <H>=-186.4660997003911 EFR=-59.09129439130406 Q=0.  
↪ 07462903595820602  
# FRAGMENT 1 - O: <H>=-186.52571212206996 EFR=-94.20327799731078 Q=-0.  
↪ 05804063711884311  
# FRAGMENT 2 - OH: <H>=-186.52760819713768 EFR=-98.32803927524827 Q=-0.  
↪ 04067365994348471  
# FRAGMENT 3 - H: <H>=-186.25776880475564 EFR=-3.8776501906423517 Q=0.  
↪ 023767280256090095  
# NEWTON ITERATION - CHEMICAL POTENTIAL 0.02618319569991223  
# FRAGMENT 0 - C: <H>=-186.46645711276085 EFR=-59.091882980913326 Q=0.  
↪ 07475597327765104  
# FRAGMENT 1 - O: <H>=-186.5262076771364 EFR=-94.203567426381 Q=-0.05796786917953156  
# FRAGMENT 2 - OH: <H>=-186.52815629411148 EFR=-98.3282967036768 Q=-0.  
↪ 04060450873466159  
# FRAGMENT 3 - H: <H>=-186.25782907787817 EFR=-3.877818167978842 Q=0.  
↪ 023815682210136968  
# CHEMICAL POTENTIAL 0.02618333425876916  
# FINAL PARAMETERS: []  
# FINAL CHEMICAL POTENTIAL: 0.02618333425876916  
# FINAL ENERGY: -186.53116138512055
```

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: -186.37687192609545
REFERENCE CCSD ENERGY: -186.40300888507934
```

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.

20.2 Projection-based embedding

This tutorial aims to provide an introduction to projection-based embedding ([Manby et al \(2012\)](#)) for quantum chemistry calculations. Projection-based embedding, particularly within Density Functional Theory (DFT), is employed to address the computational challenges associated with large systems. This technique divides the system into a subsystem (e.g., a molecule) and an environment (e.g., crystal or solvent), treating them with different methods to optimize computational efficiency while maintaining accuracy for the region of interest. The fundamental concept involves projecting the wavefunction or electron density of the entire system onto the subsystem using projection operators. These operators define the interaction between the electronic structure of the subsystem and the surrounding environment.

The implementation involves partitioning the system, projecting the electronic information onto the subsystem using projection operators, and performing separate calculations for the subsystem (using DFT) and the environment (using a simpler method). The results are then combined to obtain the overall electronic structure of the system. The form of the projection operators depends on the specific details of the chosen embedding method. These operators encompass terms related to electron density, potential, or other relevant properties that describe the interaction between the subsystem and the environment.

Applications of projection-based embedding are prominent in materials science, especially in studying electronic properties in complex environments like surfaces or interfaces. The accuracy of this technique depends on the chosen methods for the subsystem and the approximations made in describing the environment.

Wavefunction-in-DFT Embedding

The projection-based approach readily allows for wavefunction-in-DFT (WF-in-DFT) embedding, in which subsystem A is treated using a WF-level description and subsystem B is described at the DFT level. Here, the WF-in-DFT energy is simply obtained by substituting the DFT energy of subsystem A with the corresponding WF energy:

$$\begin{aligned} E_{\text{WF-in-DFT}} [\tilde{\Psi}^A; \gamma^A, \gamma^B] = & E_{\text{WF}} [\tilde{\Psi}^A] + \text{tr} [(\tilde{\gamma}^A - \gamma^A) \mathbf{v}_{\text{emb}} [\gamma^A, \gamma^B]] + \\ & + E_{\text{DFT}} [\gamma^A + \gamma^B] - E_{\text{DFT}} [\gamma^A] + \mu \text{tr} [\tilde{\gamma}^A \mathbf{P}^B] \end{aligned} \quad (20.1)$$

where $\tilde{\Psi}^A$ is the WF for subsystem A, γ^A is the one-particle reduced density matrix corresponding to $\tilde{\Psi}^A$, and $E_{WF} [\tilde{\Psi}^A]$ is the WF energy of subsystem A.

A projection-based WF-in-DFT embedding calculation proceeds as follows. A KS-DFT calculation is first performed over the full system. The resulting occupied MOs are localized and partitioned into two sets, corresponding to subsystems A and B. These sets are used to construct \mathbf{h}^{A-in-B} ,

$$\mathbf{h}^{A-in-B} [\gamma^A, \gamma^B] = \mathbf{h} + \mathbf{v}_{emb} [\gamma^A, \gamma^B] + \mu \mathbf{P}^B \quad (20.2)$$

which is an effective one-electron Hamiltonian containing the standard one-electron Hamiltonian, the embedding potential, and the projection operator. Finally, a correlated WF calculation is performed on subsystem A wherein \mathbf{h}^{A-in-B} replaces the standard one-electron Hamiltonian. The final WF-in-DFT energy is given by equation 1.

The WF calculation for subsystem A consists of two steps: first, a set of reference orbitals is generated, and second, a correlated WF calculation is performed using those orbitals. The reference orbitals can be obtained either via Hartree–Fock (HF) or a multiconfigurational method. For the former case, the subsystem A post-HF calculation begins with HF-in-DFT embedding. The HF-in-DFT Fock matrix, \mathbf{F}^A , is derived by inserting a Slater determinant for the subsystem A WF into equation 1 and differentiating with respect to $\tilde{\gamma}_{HF}^A$, giving

$$\mathbf{F}^A = \frac{\partial}{\partial \tilde{\gamma}_{HF}^A} E_{HF-in-DFT} [\tilde{\gamma}_{HF}^A; \gamma^A, \gamma^B] = \mathbf{h}^{A-in-B} [\gamma^A, \gamma^B] + \mathbf{g} [\tilde{\gamma}_{HF}^A] \quad (20.3)$$

where \mathbf{g} includes all of the usual HF two-electron terms and \mathbf{h}^{A-in-B} represents the effective one-electron Hamiltonian given by equation 2. Once the subsystem A HF MOs are optimized in the presence of the DFT embedding potential, they are used for the correlated subsystem A post-HF calculation. An analogous procedure holds for the case of multireference methods, wherein a multiconfigurational WF is substituted in place of the single Slater determinant in equation 1. In this way, projection-based WF-in-DFT embedding can be readily performed with any existing WF method (or quantum impurity solver) simply by modifying the one-electron Hamiltonian in the WF method to include the projection-based embedding terms. While projection-based embedding is exact for (same-functional) DFT-in-DFT embedding, projection-based WF-in-DFT embedding is necessarily approximate. For additional details, you can refer to [Manby et al \(2012\)](#).

WF-in-DFT in InQuanto

In InQuanto, WF-in-DFT is partially based on [PsiEmbed](#), a computational chemistry software package tailored for quantum mechanical calculations, particularly focusing on projection-based embedding techniques. To demonstrate this embedding method, we employ the stretched ethane (C2H6) molecule as an example, as depicted below using the NGLView visualizer for InQuanto.

Here are the steps outlined:

- Define the system
- Define the embedded driver
- Run VQE to get the ground state energy using the embedded driver

```
[ ]: import warnings
warnings.filterwarnings('ignore')

geometry = [
    ["C", [ 0.000000000, -0.664929641,  0.141563265]],
    ["C", [ 0.000000000,  0.664929641,  0.141563265]],
    ["H", [ 0.923341000, -1.237750972,  0.141563265]],
    ["H", [-0.923341000, -1.237750972,  0.141563265]],
    ["H", [ 0.923341000,  1.237750972,  0.141563265]],
    ["H", [-0.923341000,  1.237750972,  0.141563265]],
    ["H", [ 0.000000000,  0.000000000, -3.937392259]],
    ["H", [ 0.000000000,  0.000000000,  4.200000000]],
```

(continues on next page)

(continued from previous page)

]

The NGLView visualizer for InQuanto can be used to display the system.

```
[ ]: from inquanto.geometries import GeometryMolecular
from inquanto.extensions.nglview import VisualizerNGL

C2H6_geom = GeometryMolecular(geometry)
visualizer = VisualizerNGL(C2H6_geom)

[ ]: #visualizer.visualize_molecule()
```



The initial step involves creating the InQuanto-PySCF projection-based embedding driver (`inquanto.extensions.pyscf.ChemistryDriverPySCFEmbeddingROHF`). This driver is essential for conducting molecular ROHF calculations and storing the outcomes in an InQuanto QubitOperator.

Parameters used:

`geometry` – Molecular geometry.

`basis` – Atomic basis set valid for Mole class.

`multiplicity` – Spin multiplicity of the total system.

`frozen` – Frozen orbital information.

`functional` – KS functional to use for the system calculation, or None if RHF is desired.

The `FromActiveSpace` function aids in determining the frozen orbital list based on the information provided about the active space. In this context, `ncas` represents the number of active orbitals, and `nelecas` represents the number of active electrons.

```
[ ]: from inquanto.extensions.pyscf import ChemistryDriverPySCFEmbeddingROHF,
    ↪FromActiveSpace

ncas, nelecas = 2, 2

driver = ChemistryDriverPySCFEmbeddingROHF (
    geometry=geometry,
    basis="3-21G*",
    multiplicity=3,
    frozen=FromActiveSpace(ncas, nelecas),
    functional="b3lyp5",
)
```

Additionally, it's possible to compute the HF energy using the InQuanto driver.

```
[ ]: driver.run_hf()
-79.0951494122172
```

The `get_system` function is responsible for computing the fermionic Hamiltonian operator, Fock space, and Hartree Fock state. The `qubit_encode` function carries out qubit encoding, utilizing the mapping class associated with the current integral operator. The default mapping approach is the Jordan-Wigner method.

```
[ ]: chem_hamiltonian, space, state = driver.get_system()

qubit_hamiltonian = chem_hamiltonian.qubit_encode()
```

To construct our ansatz for the specified fermion space and fermion state, we have employed the Chemically Aware Unitary Coupled Cluster with singles and doubles excitations (UCCSD). The circuit is synthesized using Jordan-Wigner encoding.

```
[ ]: from inquanto.ansatzes import FermionSpaceAnsatzChemicallyAwareUCCSD

ansatz = FermionSpaceAnsatzChemicallyAwareUCCSD(space, state)
```

Here, we carry out a straightforward VQE experiment to obtain the ground state energy of our system. For a more extensive guide on executing VQE calculations with InQuanto on quantum computers, we suggest referring to the [VQE tutorial](#). To define where the computation is performed we set the `backend` to `AerStateBackend()`.

```
[ ]: from inquanto.express import run_vqe
from pytket.extensions.qiskit import AerStateBackend
from inquanto.minimizers import MinimizerRotosolve

backend = AerStateBackend()

vqe = run_vqe(
    ansatz,
    qubit_hamiltonian,
    backend=backend,
    with_gradient=False,
    minimizer=MinimizerRotosolve(),
)
# TIMER BLOCK-1 BEGINS AT 2024-02-15 21:02:39.509513
# TIMER BLOCK-1 ENDS - DURATION (s): 0.0184985 [0:00:00.018499]
```

Finally, we can print out the calculated energy value using VQE and compare the results with those obtained from CASCI and HF methods.

```
[ ]: vqe.final_value
-79.15210375702887
```

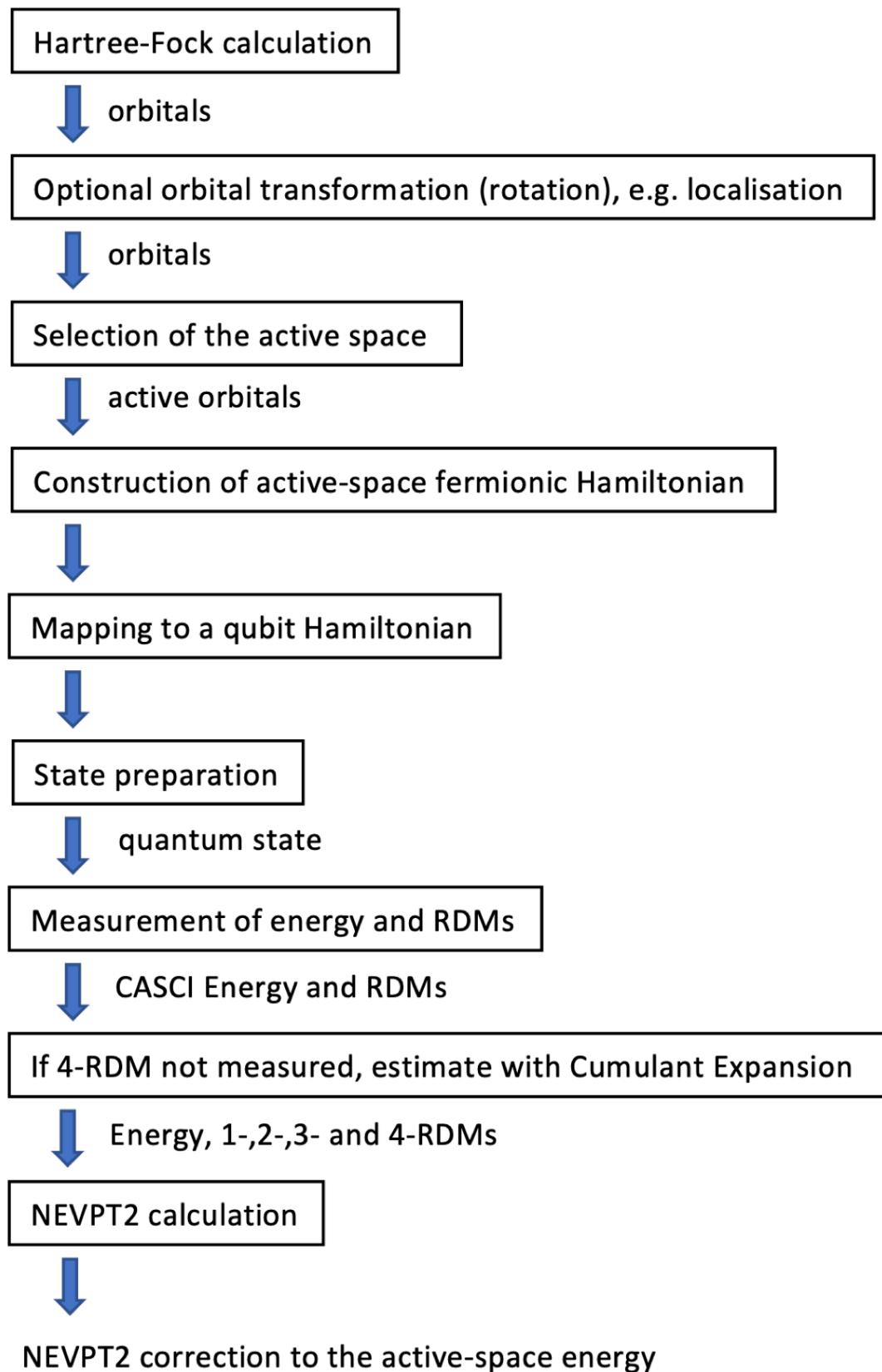
20.3 NEVPT2 and AC0 energy corrections

In this tutorial, we will explore the use of NEVPT2 (N-electron valence state perturbation theory with second-order approximations) as a perturbation theory approach that can be used to improve the accuracy of our electronic structure calculations beyond the mean-field methods like Hartree-Fock or density functional theory (DFT) when running quantum

computations. Our focus will be on the Li₂ system, which presents a higher level of complexity compared to the H₂ molecule while still maintaining a reasonable level of simplicity.

A novel approach that combines quantum and classical methods for strongly contracted N-electron Valence State 2nd-order Perturbation Theory (SC-NEVPT2) is implemented in InQuanto. In this method, the static correlation effects typically handled by the Complete Active Space Configuration Interaction (CASCI) step are replaced by quantum computer simulations. Specifically, we use a quantum computer to measure n-particle Reduced Density Matrices (n-RDMs), and these measurements are then employed in a classical SC-NEVPT2 calculation to approximate the remaining dynamic electron correlation effects. Additionally, we explore the application of the cumulant expansion to either approximate the entire 4-RDM matrix or only its zero elements. This work not only showcases noiseless state-vector quantum simulations but also marks the first instance of a hybrid quantum-classical calculation for multi-reference perturbation theory, with the quantum component executed on a quantum computer.

The flowchart depicted below provides a concise overview of this approach, with the “State preparation” step in the current implementation being handled by the Variational Quantum Eigensolver (VQE). For further details and comprehensive information about this approach, see [Krompiec & Muñoz Ramo \(2022\)](#).



As an alternative approach, we can calculate the energy's AC0 correction using the provided density matrices. Here, one- and two-particle reduced density matrices are employed to account for electron correlation effects. The AC-CAS (Adiabatic Connection Construction and Extended Random Phase Approximation for Complete Active Space Wave Functions) is a computational method within the field of quantum chemistry employed to investigate the electronic structure and properties of molecular systems. This method combines two crucial techniques: the adiabatic connection (AC) method and the complete active space (CAS) approach, along with the extended random phase approximation.

In this context, we have utilized the AC0-CAS method, known as the First-Order Expansion of the AC Integrand at $\alpha = 0$. Starting with $\alpha = 0$ essentially means beginning with the simplest form of the system, which corresponds to the reference system with non-interacting electrons. Subsequently, electron-electron interactions are introduced perturbatively, taking into account only the first-order effects. This approach is commonly employed in quantum chemistry to make calculations more manageable while still capturing some degree of electron correlation. AC0-CAS proves particularly valuable in striking a balance between computational efficiency and accuracy in electronic structure calculations. While it may not encompass the full spectrum of electron correlation effects, it can still yield reasonably accurate results for a broad range of molecular systems. Further information on the AC0 correction can be found in [Pastorczak & Pernal \(2018\)](#).

The following steps outline the process:

- Define the system
- Compute the NEVPT2 correction to the energy using RDMs from VQE
- Compute the NEVPT2 correction to the energy using RDMs from CASCI
- Compute the AC0 correction to the energy

The first step is to run Hartree-Fock (HF) calculations and in our case restricted Hartree-Fock (RHF) with the 6-31G basis set. We use an InQuanto-PySCF driver to perform the molecular RHF calculations, and store the resulting Hamiltonian after it is qubit encoded.

Parameters used:

`geometry` – Molecular geometry.

`basis` – Atomic basis set valid for Mole class.

`charge` – Total charge.

`transf` – Orbital transformer.

`frozen` – Frozen orbital information.

`point_group_symmetry` – Enable point group symmetry.

`verbose` – Control PySCF verbosity.

The `FromActiveSpace` function aids in determining the frozen orbital list based on the information provided about the active space. In this context, `ncas` represents the number of active orbitals, and `nelecas` represents the number of active electrons. In this case, we have opted to perform postprocessing of orbitals using the Complete Active Space Self-Consistent Field (CASSCF) method to construct molecular integrals. We pass two key parameters, `ncas` (the number of active spatial orbitals) and `nelecas` (the number of active electrons), to `pyscf.mcscf.CASSCF`. This approach allows us to effectively utilize the CASSCF method for the subsequent steps involving molecular integrals.

```
[ ]: import warnings
warnings.filterwarnings('ignore')

from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF, CASSCF,
    FromActiveSpace

geometry = [{"Li": [0.0, 0.0, 0.0]}, {"Li": [2.63, 0.0, 0.0]}]
```

(continues on next page)

(continued from previous page)

```

ncas, nelecas = 4, 2
driver = ChemistryDriverPySCFMolecularRHF(
    basis="6-31G",
    charge=0,
    geometry=geometry,
    transf=CASSCF(ncas, nelecas),
    frozen=FromActiveSpace(ncas, nelecas),
    point_group_symmetry=True,
    verbose=0,
)
driver.run_hf()
-14.864996327400355

```

The `get_system` function is responsible for computing the fermionic Hamiltonian operator, Fock space, and Hartree Fock state. The `qubit_encode` function carries out qubit encoding, utilizing the mapping class associated with the current integral operator. The default mapping is Jordan-Wigner (used throughout this tutorial).

```
[ ]: chemistry_hamiltonian, space, state = driver.get_system()
qubit_hamiltonian = chemistry_hamiltonian.qubit_encode()
```

To construct our ansatz for the specified fermion space and fermion state, we have employed the Chemically Aware Unitary Coupled Cluster with singles and doubles excitations (UCCSD).

```
[ ]: from inquanto.ansatzes import FermionSpaceAnsatzChemicallyAwareUCCSD
ansatz = FermionSpaceAnsatzChemicallyAwareUCCSD(space, state)
```

Here, we conduct a simple VQE experiment to determine the ground state energy of our system. For a more comprehensive guide on performing VQE calculations using InQuanto on quantum computers, we recommend referring to the [VQE tutorial](#). To define where the computation is performed we set the `backend` to `AerStateBackend()`.

```
[ ]: from pytket.extensions.qiskit import AerStateBackend
from inquanto.express import run_vqe
from inquanto.minimizers import MinimizerRotosolve

backend = AerStateBackend()

vqe = run_vqe(
    ansatz,
    qubit_hamiltonian,
    backend=backend,
    with_gradient=False,
    minimizer=MinimizerRotosolve(),
)
# TIMER BLOCK-1 BEGINS AT 2024-02-15 19:05:04.004600
# TIMER BLOCK-1 ENDS - DURATION (s): 0.6568890 [0:00:00.656889]
```

The InQuanto `inquanto.spaces.QubitSpace.symmetry_operators_z2` function is employed to retrieve a list of symmetry operators applicable to our system. These symmetry operators are associated with the point group, spin parity, and particle number parity \mathbb{Z}_2 symmetries that uphold a specific symmetry sector.

```
[ ]: from inquanto.spaces import QubitSpace

symmetry_operators = QubitSpace(space.n_spin_orb).symmetry_operators_z2(
    qubit_hamiltonian
)
```

The NEVPT2 implementation in PySCF, which is used in InQuanto, utilizes Pre-Density Matrices (PDMs). These PDMs are defined using the same creation and annihilation operators as the corresponding Reduced Density Matrices (RDMs), but they are applied in a different order. The class `inquanto.computables.composite.PDM1234RealComputable` computes the 1st, 2nd, 3rd, and 4th PDMs for a specific state, as determined by the provided `ansatz` and parameters.

Parameters used:

`space` – Fermion occupation space spanned by this RDM.
`ansatz` – Ansatz state with respect to which expectation values are computed.
`encoding` – Fermion to qubit mapping.
`symmetry_operators` – \mathbb{Z}_2 symmetries of the Hamiltonian.
`cas_elec` – Number of active electrons.
`cas_orbs` – Number of active orbitals.

The computation of the 4-RDM can be computationally expensive. Therefore, for chemical systems with more than 3 active electrons (where the 4-RDM is non-negligible), we estimate the 4-PDM using the cumulant expansion approximation. This approximation is set by the class parameter `cu4`, which is `True` by default. If this parameter is set to `False`, the 4-PDM is directly computed. For further insights into the impact of this approximation on both computational cost and accuracy, please refer to [Krompiec & Muñoz Ramo \(2022\)](#).

```
[ ]: from inquanto.computables.composite import PDM1234RealComputable
from inquanto.mappings import QubitMappingJordanWigner

pdm_computables = PDM1234RealComputable(
    space=space,
    ansatz=ansatz,
    encoding=QubitMappingJordanWigner(),
    symmetry_operators=symmetry_operators,
    cas_elec=nelecas,
    cas_orbs=ncas,
)
```

Subsequently, the evaluation PDMs is carried out using the `inquanto.protocols.SparseStatevectorProtocol` class, which is designed for sparse statevector calculations while utilizing caching. The `get_evaluator` method is employed to generate and provide a function (`evaluator`) that receives a specific quantum computable and computes it based on its type.

```
[ ]: from inquanto.protocols import SparseStatevectorProtocol

pdms = pdm_computables.evaluate(
    SparseStatevectorProtocol(backend).get_evaluator(vqe.final_parameters)
)
```

Finally, the `get_nevpt2_correction` function is employed to calculate the strongly contracted NEVPT2 correction to the energy using the provided density matrices.

```
[ ]: nevpt2_energy = driver.get_nevpt2_correction(pdms)
```

In a similar way, using the same driver, we can calculate the AC0 energy correction, but this time we will make use of the `inquanto.computables.composite.SpinlessNBodyRDMArrayRealComputable` class which calculates a general n-body RDM $\Gamma_{n...m}^{i...j} = \langle \Psi_0 | \hat{E}_{n...m}^{i...j} | \Psi_0 \rangle$ where $\hat{E}_{n...m}^{i...j}$ is a spin-traced excitation operator. For example, in the one-body case this is:

$$\hat{E}_q^p = \hat{a}_{p\uparrow}^\dagger \hat{a}_{q\uparrow} + \hat{a}_{p\downarrow}^\dagger \hat{a}_{q\downarrow} \quad (20.4)$$

And in the two-body case:

$$\hat{E}_{qs}^{pr} = \hat{a}_{p\uparrow}^\dagger \hat{a}_{r\uparrow}^\dagger \hat{a}_{s\uparrow} \hat{a}_{q\uparrow} + \hat{a}_{p\uparrow}^\dagger \hat{a}_{r\downarrow}^\dagger \hat{a}_{s\downarrow} \hat{a}_{q\uparrow} + \hat{a}_{p\downarrow}^\dagger \hat{a}_{r\uparrow}^\dagger \hat{a}_{s\uparrow} \hat{a}_{q\downarrow} + \hat{a}_{p\downarrow}^\dagger \hat{a}_{r\downarrow}^\dagger \hat{a}_{s\downarrow} \hat{a}_{q\downarrow} \quad (20.5)$$

Spin-traced 1-RDM and 2-RDM refer to reduced density matrices where the electron spin degrees of freedom have been summed over or “traced out.” This is done to obtain the spatial (orbital) electronic density matrices, which describe the distribution of electrons in terms of their spatial coordinates or orbitals, while ignoring their spin states.

Parameters used:

`n` – Rank of the spin-traced RDM (n-RDM).

`fermion_space` – Fermion space where the operators are defined.

`ansatz` – Ansatz with respect to which the expectation values are computed.

`encoding` – Qubit encoding from fermion space to qubit space.

`symmetry_operators` – List of \mathbb{Z}_2 symmetries of the Hamiltonian.

`taperer` – Optional taperer object.

The `inquanto.protocols.SparseStatevectorProtocol` class and its `get_evaluator` method are utilized in the same way as previously.

```
[ ]: from inquanto.computables.composite import SpinlessNBodyRDMArrayRealComputable
from inquanto.computables import ComputableList

rdm_computable = ComputableList(
    [
        SpinlessNBodyRDMArrayRealComputable(
            n=n,
            fermion_space=space,
            ansatz=ansatz,
            encoding=QubitMappingJordanWigner(),
            symmetry_operators=symmetry_operators,
            taperer=None,
        )
        for n in (1, 2)
    ]
)
rdm_computed = rdm_computable.evaluate(
    SparseStatevectorProtocol(backend).get_evaluator(vqe.final_parameters)
)

dm1 = rdm_computed[0]
dm2 = rdm_computed[1]
```

The `get_ac0_correction` function is employed to compute the AC0 correction to the energy from the provided density matrices.

```
[ ]: ac0_energy = driver.get_ac0_correction((dm1, dm2))
```

We can now compare the various energy results.

An alternative approach would be to perform CASSCF calculations to obtain the PDMs and RDMs. In this process, we initially compute the PDMs by executing the CASSCF calculation using the PySCF package. For additional details, you can refer to the relevant PySCF documentation. Following this, our RHF InQuanto driver has been created and the HF orbital coefficients have been replaced with those obtained from the CASSCF calculation in order to improve the description of the orbitals and enhance the representation of the system.

```
[ ]: from pyscf import gto, scf, fci, mcscf

mf = gto.M(atom=geometry, basis="6-31g").apply(scf.RHF).run()
mc = mcscf.CASSCF(mf, ncas, nelecas)
mc.run()
mf.mo_coeff=mc.mo_coeff
(pdm1, pdm2, pdm3, pdm4) = fci.rdm.make_dm1234(
    "FCI4pdm_kern_sf", mc.ci, mc.ci, ncas, nelecas
)

driver_casscf = ChemistryDriverPySCFMolecularRHF.from_mf(
    mf, frozen=FromActiveSpace(ncas, nelecas)(mf)
)

converged SCF energy = -14.8652654987828
CASSCF energy = -14.8887816132799
CASCI E = -14.8887816132799 E(CI) = -0.619242398867161 S^2 = 0.0000000
```

We can once again employ the `get_nevpt2_correction` function to compute the strongly contracted NEVPT2 correction to the energy.

```
[ ]: nevpt2_energy_casscf = driver_casscf.get_nevpt2_correction(rdm1=(pdm1, pdm2, pdm3, pdm4))

Sr (-1)', E = 0.00000000000063
Si (+1)', E = -0.00001592230223
Sijrs (0) , E = -0.00022778404137
Sijr (+1) , E = -0.00007024183793
Srsi (-1) , E = -0.00023752538838
Srs (-2) , E = -0.00127421582345
Sij (+2) , E = -0.00002668012481
Sir (0)', E = -0.00013325632488
Nevpt2 Energy = -0.001985625842400
```

The `make_dm1234` function in PySCF calculates the PDMs, which do not directly correspond to the 1-, 2-, 3-, and 4-particle density matrices. The `reorder_dm12` function is then used to convert these PDMs into the actual 1- and 2-particle density matrices.

```
[ ]: rdm1, rdm2 = fci.rdm.reorder_dm12(pdm1, pdm2)

ac0_energy_casscf = driver_casscf.get_ac0_correction((rdm1, rdm2))
```

As the final step, we can compare the ground-state energy results obtained using different methods.

```
[ ]: print("VQE Energy: {:.6f} Ha".format(vqe.generate_report()["final_value"]))
print("VQE correction to HF: {:.6f} Ha".format(vqe.generate_report()["final_value"]-
    ↪driver.run_hf()))

print("NEVPT2 correction using VQE PDMs: {:.6f} Ha".format(nevpt2_energy))
print("Total VQE+NEVPT2 energy: {:.6f} Ha".format(vqe.generate_report()["final_value"
    ↪"]+nevpt2_energy))

print("NEVPT2 correction using CASSCF PDMs: {:.6f} Ha".format(nevpt2_energy_casscf))
print("Total VQE+NEVPT2 energy: {:.6f} Ha".format(vqe.generate_report()["final_value"
    ↪"]+nevpt2_energy_casscf))

print("AC0 correction using VQE PDMs: {:.6f} Ha".format(ac0_energy))
print("Total VQE+AC0 energy: {:.6f} Ha".format(vqe.generate_report()["final_value"
    ↪"]+ac0_energy))

print("AC0 correction using CASSCF PDMs: {:.6f} Ha".format(ac0_energy_casscf))
print("Total VQE+AC0 energy: {:.6f} Ha".format(vqe.generate_report()["final_value"
    ↪"]+ac0_energy_casscf))

VQE Energy: -14.888722 Ha
VQE correction to HF: -0.023726 Ha
NEVPT2 correction using VQE PDMs: -0.002049 Ha
Total VQE+NEVPT2 energy: -14.890771 Ha
NEVPT2 correction using CASSCF PDMs: -0.0001986 Ha
Total VQE+NEVPT2 energy: -14.890708 Ha
AC0 correction using VQE PDMs: -0.003039 Ha
Total VQE+AC0 energy: -14.891761 Ha
AC0 correction using CASSCF PDMs: -0.002515 Ha
Total VQE+AC0 energy: -14.891237 Ha
```

20.4 Projection-based embedding with energy corrections

This can be viewed as a continuation of the [projection-based embedding tutorial](#). In this tutorial, we will once again utilize projection-based embedding for quantum chemistry calculations within the context of Density Functional Theory (DFT). However, the key distinction here is the incorporation of NEVPT2 and AC0 energy correction methods to enhance the accuracy of our ground-state energy calculation. Incorporating these methods into our workflow requires us to redefine the final InQuanto driver used for this example because our previous approach did not yield the correct energy results. To demonstrate this, we have selected ethanol (C:math:2H_6O) as our test case.

To gain a better understanding of the application of NEVPT2 and AC0 to enhance the precision of quantum computations beyond mean-field methods like Hartree-Fock or density functional theory (DFT), please consult the [NEVPT2+AC0 tutorial](#). For further insights into Projection-based embedding, please refer to [Manby et al \(2012\)](#).

Here are the steps outlined:

- Define the system.
- Define the embedded RHF driver using AVAS.
- Generate a driver object wrapping the active space Hamiltonian.
- Run VQE to get the ground state energy and final parameters.
- Compute the NEVPT2 correction to the energy.

- Compute the AC0 correction to the energy.

The initial step involves creating the InQuanto-PySCF projection-based embedding driver (`inquanto.extensions.pyscf.ChemistryDriverPySCFEmbeddingRHF`). This driver is essential for conducting and storing the outcome of molecular RHF calculations.

Parameters used:

`geometry` – Molecular geometry.

`basis` – Atomic basis set valid for Mole class.

`frozen` – Frozen orbital information.

`transf` – Orbital transformer.

`functional` – KS functional to use for the system calculation, or None if RHF is desired.

In order to reduce hardware resource requirements in multi-configuration and multireference electronic structure calculations, the Atomic Valence Active Space (AVAS) approximation in the PySCF extension of InQuanto was used. AVAS is an automated technique for creating active orbital spaces, particularly useful for describing electronic configurations arising from specific atomic valence orbitals, like metal d orbitals in coordination complexes. The variable `aolabels` contains the atomic orbital (AO) labels for the AO active space. The variables `threshold` and `threshold_vir` specify the truncation thresholds for the AO-projector. These thresholds determine which AOs are retained within the active space for occupied and virtual orbitals, respectively, based on their values. More detailed information can be found in Sayfutyarova et al (2017) and in [the Fe4N2 system preparation tutorial](#).

```
[1]: import warnings
warnings.filterwarnings('ignore')

geometry = [
    ["O", [-1.1867, -0.2472, 0.0000]],
    ["H", [-1.9237, 0.3850, 0.0000]],
    ["H", [-0.0227, 1.1812, 0.8852]],
    ["C", [0.0000, 0.5526, 0.0000]],
    ["H", [-0.0227, 1.1812, -0.8852]],
    ["C", [1.1879, -0.3829, 0.0000]],
    ["H", [2.0985, 0.2306, 0.0000]],
    ["H", [1.1184, -1.0093, 0.8869]],
    ["H", [1.1184, -1.0093, -0.8869]],
]

basis="3-21G",
functional="b3lyp5"
```

```
[2]: from inquanto.extensions.pyscf import ChemistryDriverPySCFEmbeddingRHF, AVAS,
    ↪FromActiveSpace

avas = AVAS(
    aolabels=["0 0 2p", "0 0 3p", "0 0 2s", "0 0 3s", "0 0 3d", "1 H"],
    threshold=0.8,
    threshold_vir=0.5,
)

driver = ChemistryDriverPySCFEmbeddingRHF(
    geometry=geometry,
```

(continues on next page)

(continued from previous page)

```

    basis=basis,
    functional=functional,
    transf=avas,
    frozen=avas.frozenf,
)

```

To incorporate dynamic energy correlation methods, it is necessary to create a new driver object derived from the effective WF-in-DFT Hamiltonian. Hence, a PySCF driver object that encapsulates the Hamiltonian of the current active space is created using the InQuanto `get_subsystem_driver` function. This function accepts two arguments: the frozen orbital information (`frozen`), and the orbital transformer (`transf`).

`FromActiveSpace` aids in determining the frozen orbital list based on the information provided about the active space. In this context, `ncas` represents the number of active orbitals, and `nelecas` represents the number of active electrons. We could also perform postprocessing of orbitals using the Complete Active Space Self-Consistent Field (CASSCF) method to construct molecular integrals by enabling the `casscf_transform` parameter.

The `get_system` function is responsible for computing the fermionic Hamiltonian operator, Fock space, and Hartree Fock state. The `qubit_encode` function carries out qubit encoding, utilizing the mapping class associated with the current integral operator. The default mapping is Jordan-Wigner (used throughout this tutorial).

```
[3]: from inquanto.extensions.pyscf._transf import CASSCF

ncas, nelecas = 4, 4
casscf_transform = False

if casscf_transform:
    driver_embedded = driver.get_subsystem_driver(frozen=FromActiveSpace(ncas,_
→nelecas), transf=CASSCF(ncas, nelecas))
else:
    driver_embedded = driver.get_subsystem_driver(frozen=FromActiveSpace(ncas,_
→nelecas))

chem_hamiltonian, space, state = driver_embedded.get_system()
qubit_hamiltonian = chem_hamiltonian.qubit_encode()
```

When we examine the imaginary-coefficient terms of our Hamiltonian by using the `hermitian_factorisation` property, we observe that they are negligible for this active space. Consequently, we have constructed our qubit Hamiltonian by utilizing the hermitian part which exclusively includes terms with real coefficients.

```
[4]: hermitian_part, antihermitian_part = qubit_hamiltonian.hermitian_factorisation()
print(antihermitian_part)
qubit_hamiltonian_hermitian = qubit_hamiltonian.hermitian_part()

(-3.492467502263369e-10, Y4 Z5 X6), (3.492467502263369e-10, X4 Z5 Y6), (-3.
→4924675000949645e-10, Y5 Z6 X7), (3.4924675000949645e-10, X5 Z6 Y7)
```

To construct our ansatz for the specified fermion space and fermion state, we have employed the Chemically Aware Unitary Coupled Cluster with singles and doubles excitations (UCCSD).

```
[5]: from inquanto.ansatze import FermionSpaceAnsatzChemicallyAwareUCCSD

ansatz = FermionSpaceAnsatzChemicallyAwareUCCSD(space, state)
```

Here, we carry out a straightforward VQE experiment to obtain the ground state energy of our system. For a more extensive guide on executing VQE calculations with InQuanto on quantum computers, we suggest referring to the [VQE](#)

tutorial. To define where the computation is performed we set the backend to `AerStateBackend()`.

```
[6]: from inquanto.express import run_vqe
from pytket.extensions.qiskit import AerStateBackend
from inquanto.minimizers import MinimizerRotosolve

backend = AerStateBackend()

vqe = run_vqe(
    ansatz,
    qubit_hamiltonian_hermitian,
    backend=backend,
    with_gradient=False,
    minimizer=MinimizerRotosolve(),
)
# TIMER BLOCK-0 BEGINS AT 2024-07-08 12:41:56.512236
# TIMER BLOCK-0 ENDS - DURATION (s): 92.4722578 [0:01:32.472258]
```

The InQuanto `symmetry_operators_z2` function is employed to retrieve a list of symmetry operators applicable to our system. These symmetry operators are associated with the point group, spin parity, and particle number parity \mathbb{Z}_2 symmetries that uphold a specific symmetry sector.

```
[7]: from inquanto.spaces import QubitSpace

symmetry_operators = QubitSpace(space.n_spin_orb).symmetry_operators_z2(
    qubit_hamiltonian_hermitian
)
```

In InQuanto, we can use a quantum computer to measure n-particle Reduced Density Matrices (n-RDMs) and employ them in a classical SC-NEVPT2 calculation, replacing the static correlation effects typically handled by CASCI with quantum simulations. The remaining dynamic electron correlation effects are approximated.

The class `inquanto.computables.composite.PDM1234RealComputable` computes the 1st, 2nd, 3rd, and 4th Pre-Density Matrices (PDMs) for a specific state, as determined by the provided ansatz and parameters.

Parameters used:

`space` – Fermion occupation space spanned by this RDM.

`ansatz` – Ansatz state with respect to which expectation values are computed.

`encoding` – Fermion to qubit mapping.

`symmetry_operators` – Z_2 symmetries of the Hamiltonian.

`cas_elec` – Number of active electrons.

`cas_orbs` – Number of active orbitals.

The evaluation of PDMs is carried out using `inquanto.protocols.SparseStatevectorProtocol`, which is designed for sparse statevector calculations while utilizing caching. The `get_evaluator` method is employed to generate and provide a function (evaluator) that receives a specific quantum computable and computes it based on its type.

The `get_nevpt2_correction` function is used to compute the strongly contracted NEVPT2 correction to the energy using the provided density matrices.

For more extensive information about this method, please refer to Krompiec & Muñoz Ramo (2022).

Note that the next cell can take about 15 mins or more to run.

```
[8]: from inquanto.computables.composite import PDM1234RealComputable
from inquanto.mappings import QubitMappingJordanWigner
from inquanto.protocols import SparseStatevectorProtocol

pdm_computables = PDM1234RealComputable(
    space=space,
    ansatz=ansatz,
    encoding=QubitMappingJordanWigner(),
    symmetry_operators=symmetry_operators,
    cas_elec=ncas,
    cas_orbs=nelecas,
)

pdms = pdm_computables.evaluate(
    SparseStatevectorProtocol(backend).get_evaluator(vqe.final_parameters)
)
```

```
[9]: nevpt2_energy = driver_embedded.get_nevpt2_correction(pdms)
```

As an alternative approach, we can calculate the AC0 correction to the energy using the provided density matrices. Here, one- and two-particle reduced density matrices are employed to account for electron correlation effects. The `inquanto.computables.composite.SpinlessNBodyRDMArrayRealComputable` class was employed to calculate a general n-body RDM.

Parameters used:

`n` – n-body RDM.

`fermion_space` – Fermion space where the operators are defined.

`ansatz` – Ansatz with respect to which the expectation values are computed.

`encoding` – Qubit encoding from fermion space to qubit space.

`symmetry_operators` – List of Z2 symmetries of the Hamiltonian.

`taperer` – Optional taperer object.

The `inquanto.protocols.SparseStatevectorProtocol` class and its `get_evaluator` method are utilized in the same way as previously.

The `get_ac0_correction` function is used to calculate the AC0 correction to the energy from the provided density matrices.

To obtain both practical and theoretical insights, you can refer to the [NEVPT2+AC0 tutorial](#), which provides an in-depth exploration of NEVPT2 and AC0 corrections within InQuanto.

```
[10]: from inquanto.computables.composite import SpinlessNBodyRDMArrayRealComputable
from inquanto.computables import ComputableList

rdm_computable = ComputableList(
    [
        SpinlessNBodyRDMArrayRealComputable(
            n=n,
            fermion_space=space,
            ansatz=ansatz,
```

(continues on next page)

(continued from previous page)

```

        encoding=QubitMappingJordanWigner(),
        symmetry_operators=symmetry_operators,
        taperer=None,
    )
    for n in (1, 2)
]
)

rdm_computed = rdm_computable.evaluate(
    SparseStatevectorProtocol(backend).get_evaluator(vqe.final_parameters)
)

dm1 = rdm_computed[0]
dm2 = rdm_computed[1]

ac0_energy = driver_embedded.get_ac0_correction((dm1, dm2))

```

As the final step, we can conduct a comparison between our results and those obtained from classical calculations. Upon applying energy correction methods to the VQE result, it becomes evident that there is a very slight difference between the quantum and classical ground state energies.

```
[11]: print("FCI-in-DFT reference: {:.6f} Ha".format(-153.76095170040134))

print("VQE Energy: {:.6f} Ha".format(vqe.final_value))

print("NEVPT2 correction: {:.6f} Ha".format(nevpt2_energy))
print("Total VQE+NEVPT2 energy: {:.6f} Ha".format(vqe.final_value+nevpt2_energy))

print("AC0 correction using VQE PDMs: {:.6f} Ha".format(ac0_energy))
print("Total VQE+AC0 energy: {:.6f} Ha".format(vqe.final_value+ac0_energy))

FCI-in-DFT reference: -153.760952 Ha
VQE Energy: -153.754823 Ha
NEVPT2 correction: -0.004565 Ha
Total VQE+NEVPT2 energy: -153.759388 Ha
AC0 correction using VQE PDMs: -0.004790 Ha
Total VQE+AC0 energy: -153.759613 Ha
```

CHAPTER
TWENTYONE

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. Examples using the extensions are found *elsewhere*.

21.1 algorithms/adapt

Examples of ADAPT-VQE.

File	Description
algorithm_fermionic_adapt.py	A simulation of H ₂ in STO-3G using the ADAPT algorithm with fermionic helper functions.
algorithm_iqeb.py	A simulation of H ₂ in STO-3G using IQEB algorithm.
algorithm_adapt.py	A simulation of H ₂ in STO-3G using the ADAPT algorithm.

21.2 algorithms/qse

Examples of Quantum Subspace Expansion.

File	Description
algorithm_qse.py	A simulation of H ₂ in STO-3G using the QSE algorithm

21.3 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 using express methods
vqs_real_example_paper_phased.py	A custom equation of motion VQS time evolution simulation.
vqs_real_example_paper.py	A AlgorithmMcLachlanRealTime time evolution simulation for a small system.
imtime_evolution_example.py	An exact imaginary time evolution simulation using express methods

21.4 algorithms/vqd

Examples of Variational Quantum Deflation.

File	Description
algorithm_vqd.py	Variational Quantum Deflation using computables.

21.5 algorithms/vqe

Examples of canonical Variational Quantum Eigensolver usage.

File	Description
algorithm_vqe_uccsd.py	A canonical VQE simulation of H ₂ in STO-3G using a UCCSD Ansatz.
algorithm_vqe_shots_minimiz	Use of minimizers in shot-based calculations.
py	
algorithm_vqe_hea.py	A canonical VQE simulation of H ₂ in STO-3G using a hardware-efficient Ansatz.
algorithm_vqe_varCI.py	A canonical VQE simulation of H ₂ in STO-3G using a variational configuration interaction Ansatz.

21.6 ansatzes

Examples demonstrating usage of Ansatz classes.

File	Description
multiconfig_with_symbols.	Use of a linear combination of fermionic occupation states as a variational ansatz.
py	
layered_hea.py	Use of a layered hardware efficient Ansatz.
multiconfstate_6qubit.	Preparation of a 6 qubit linear combination of fermionic occupation states with fixed coefficients.
py	
multiconfstatebox_6qubi	Preparation of a 6 qubit linear combination of fermionic occupation states with fixed coefficients.
py	
multiconfstatebox_4qubi	Preparation of a 4 qubit linear combination of fermionic occupation states with fixed coefficients.
py	
multiconfstate_4qubit.	Preparation of a 4 qubit linear combination of fermionic occupation states with fixed coefficients.
py	
fermion_space_ansatze.py	Use of a general Fermionic Ansatz.

21.7 computables

Examples demonstrating usage of computables classes.

File	Description
computable_protocol_tutorial.py	Demonstrates the general use of computables and protocols.

21.8 computables/atomic

Examples demonstrating usage of atomic computable classes.

File	Description
expval_noisy_backend.py	Computable expression example (ExpectationValue)
derivatives_methods.py	An example showing how to use evaluate gradients
expval_pauli.py	Protocol observable averaging to calculate expectation values.
finite_difference.py	An example showing how to use evaluate gradients and compare against finite differences
computable_h2_symbolic_deriva py	Use of computables classes for STO-3G H2 expectation value and gradients with symbolic protocol.
computable_h2.py	Use of computables classes for STO-3G H2 expectation value measurement.
computable_overlap_squared. py	Computable expression example (OverlapSquared)
using_protocol_list.py	An example using protocol list for finite differences
expval_symbolic.py	Use of symbolic protocol for expectation value.

21.9 computables/composite

Examples demonstrating usage of composite computable classes.

File	Description
nonorthogonal_matrices.py	Basic CI calculation with NonOrthogonal computable.
qcm4_computable.py	Estimating energy using QCM4 computable.
qse_matrices.py	An example running QSE using computables
overlap_matrix.py	Simple overlap matrix calculation.

21.10 computables/composite/gf

Examples demonstrating usage of Green's function computable classes.

File	Description
krylov_-1.py	Example of building your own Lanczos routine, then comparing it to KrylovSubspaceComputable.
krylov_0.py	Example of evaluating LanczosCoefficientsComputable to get elements of tridiagonal matrix.
gf_measurenorm_element_shots_py	Computation (shots for Lanczos, statevector for ground state) one element of GF of Hubbard dimer.
krylov_2.py	Use of KrylovSubspaceComputable to get Green's function element from shot-based protocol from 1st Lanczos vector.
krylov_4.py	Example of evaluating LanczosMatrixComputable with sandwiched moments from shot-based protocol.
gf_measurenorm_matrix_mbqf_sp.py	Computation of the full GF matrix of the Hubbard dimer, and plot the spectral function A(omega).
gf_measurenorm_matrix_mbqf.py	Computation (noiseless statevector) the full GF matrix of the Hubbard dimer at a single frequency.
krylov_1.py	Example of evaluating LanczosMatrixComputable to get tridiagonal matrix for given dimension of Krylov space.
gf_measurenorm_matrix_mbqf_sp.py	Computation of the full GF matrix of the Hubbard dimer with shots, and plot the spectral function A(omega).
gf_measurenorm_element_shots_py	Computation (shots for Lanczos, shots for ground state) one element of GF of Hubbard dimer.
gf_measurenorm_element.py	Noiseless statevector computation of one element of GF of Hubbard dimer by sandwiching Hamiltonian moments.
gf_measurenorm_matrix_mbqf_ci.py	Generation of the circuits required for the full GF matrix of the Hubbard dimer.
krylov_3.py	Example of evaluating KrylovSubspaceComputable with sandwiched moments from shot-based protocol.

21.11 computables/composite/rdm

Examples demonstrating usage of reduced density matrix computable classes.

File	Description
rdm_computable.py	An example for constructing an RDM using computables
rdm_nevpt2.py	Test of NEVPT2 using RDMs from

21.12 computables/primitive

Examples demonstrating usage of primitive computable classes.

File	Description
basic_primitives.py	An example showing how to use quantum computables.

21.13 core

Examples demonstrating usage of logging and debugging functionality.

File	Description
context_printing.py	Using context for logging standard outputs
symbols.py	Simple demonstrations of manipulating Symbols and SymbolDict in inquanto

21.14 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_vq.py	An example impurity DMET simulation of a 3-dihydrogen ring using the express module.
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.

21.15 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.

21.16 mappings

Examples of fermion-qubit mapping functionality.

File	Description
mapping_bk.py	Use of the Bravyi-Kitaev mapping from fermions to qubits.
mapping_paraparticle.py	Use and comparison of the paraparticle mapping from fermions to qubits
mapping_jw.py	Use of the Jordan-Wigner mapping from fermions to qubits.

21.17 minimizers

Examples of classical minimizers in InQuanto.

File	Description
integrators.py	Symbolic evaluation and comparing integrator methods.
minimizers.py	Use of minimizers.

21.18 operators

Examples of functionality of operator and state classes.

File	Description
double_factorization_h2.py	Simple energy calculation using a double factorized hamiltonian.
chemistry_integral_operator.py	Creation of ChemistryRestrictedIntegralOperator (CRI) 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.

21.19 protocols

Examples demonstrating usage of protocol classes.

File	Description
expval_spam.py	Running PauliAveraging with inquanto's noise mitigation.
factorized_overlap.py	Calculating complex overlaps with the FactorizedOverlap protocols and comparing to HadamardTestOverlap.
qermit_spam_zne.py	Running protocols via Qermit's MitRes and MitEx.
hadamard_test_overlap.py	Calculating complex overlaps with the HadamardTestOverlap protocol.
hadamard_direct_df_grad_expl.py	Use of phase shift derivative for STO-3G H2 expectation value measurement.
overlap_squared_protocols.py	Compare different shot based protocols for calculating an overlap squared.
pauli_hadamard_protocols.py	Usage of the PauliAveraging and HadamardTest protocols.
feature_partition_strategy.py	Partition measurement reduction strategies for PauliAveraging protocol.
phase_shift_df_grad_expr.py	Use of phase shift derivative for STO-3G H2 expectation value measurement.
phase_shift_df_grad_heap.py	Use of phase shift derivative for STO-3G H2 expectation value measurement.
feature_observable_averaging.py	Protocol observable averaging to calculate expectation values.
expval_pmsv.py	Example using PMSV error mitigation to calculate expectation values.
feature_projective_measurement.py	Example using a simple ProjectiveMeasurements protocol.
feature_credits_cost.py	Protocol observable averaging to calculate expectation values.
expval_mitigation_methods+.py	Running protocols via Qermit's MitRes and MitEx and inquanto's noise mitigation.

21.20 protocols/phase_estimation

Examples demonstrating usage of the quantum phase estimation protocols.

File	Description
hadamard_test_iqpe_quantinuum.py	Demonstration of the generation of IPEA circuits with the specialized Quantinuum protocols.
hadamard_test_iqpe_statevector.py	Demonstration of the generation of IPEA unitaries for statevector simulation.
hadamard_test_iqpe.py	Demonstration of the generation of IPEA circuits, and their use for Hadamard test sampling.

21.21 spaces

Examples of space classes for generating operators and states.

File	Description
qubit_spaces.py	Use of QubitSpace object to generate qubit operators.
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.

21.22 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
TWENTYTWO

INQUANTO-EXTENSIONS

Several interfaces to third-party chemistry programs are available for InQuanto, known as InQuanto extensions.

Currently, these extensions consist of:

- *InQuanto-PySCF* : utilizes the PySCF library for advanced quantum chemistry simulations and analysis.
- *InQuanto-NGLView* : seamlessly integrates with the NGLView Jupyter widget, providing an interactive platform for molecular visualization within the context of quantum chemistry simulations.
- *InQuanto-Phayes* : provides an interface to the Phayes package, streamlining Bayesian quantum phase estimation with a focus on one specific flavor.

These extensions collectively contribute to the comprehensive capabilities of InQuanto, offering users enhanced tools for quantum chemistry research, interactive molecular visualization, and specialized Bayesian quantum phase estimation tasks.

CHAPTER
TWENTYTHREE

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.

23.1 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 H₂ 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
```

(continues on next page)

(continued from previous page)

3	0.336424	F1^ F0^ F0	F1
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).

23.2 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.

23.3 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 `in quanto-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 `ChemistryDriverPySCFGammaRHF` driver for a Pd (111) surface with a 2x2x1 slab and 20 Å of vacuum:

```
from pyscf.pbc import gto as pgto
from in quanto.extensions.pyscf import ChemistryDriverPySCFGammaRHF

cell_pd221=[
    [5.50129075763134, 0.0, 0.0],
    [2.75064537881567, 4.764257549713281, 0.0],
    [0.0, 0.0, 20.0]
]

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",
```

(continues on next page)

(continued from previous page)

```

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
)

```

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: 14080944.278478837
```

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.

23.4 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",

```

(continues on next page)

(continued from previous page)

```

)
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	-1.12654503453569
MP2	-1.1440347834365328

CCSD	-1.1516885473648506
CASCI	-1.151688547516609

23.5 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`

23.6 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:sub: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
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],
```

(continues on next page)

(continued from previous page)

```
)
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(
```

(continues on next page)

(continued from previous page)

```

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']
```

```
aolabels_vir = ['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 [0.96867354]
```

```
Inactive from occupied = 1
```

```
Active from unoccupied = 1 , eig [0.98544091]
```

```
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.

23.7 Energy correction with NEVPT2 and AC0

InQuanto introduces an innovative approach merging quantum and classical techniques for strongly contracted N-electron Valence State 2nd-order Perturbation Theory (SC-NEVPT2), replacing CASCI's static correlation with quantum computer-simulated n-particle Reduced Density Matrices (n-RDMs) measurements [61]. These measurements are then employed in a classical SC-NEVPT2 calculation to approximate the remaining dynamic electron correlation effects, marking a pioneering hybrid quantum-classical calculation for multi-reference perturbation theory, utilizing a quantum computer for the quantum component.

CASSCF calculations are initially performed using the PySCF package to obtain Pre-Density Matrices (PDMs) [62] and RDMs. The `inquanto-pyscf` driver sets up the calculation, and `from_mf()` initializes the driver from a PySCF mean-field object. The example involves an Li₂ molecule with 6-31g atomic basis set. HF orbital coefficients are replaced with those from CASSCF to enhance the orbital description and system representation. The `get_nevpt2_correction` function is employed to calculate the strongly contracted NEVPT2 correction to the energy using the provided density matrices. The `make_dm1234` function in PySCF calculates the PDMs, which do not directly correspond to the 1-, 2-, 3-, and 4-particle density matrices.

```
from inquanto.extensions.pyscf import FromActiveSpace, ChemistryDriverPySCFMolecularRHF
from pyscf import gto, scf, fci, mcscf

geometry = [["Li", [0.0, 0.0, 0.0]], ["Li", [2.63, 0.0, 0.0]]]
ncas, nelecas = 4, 2

mf = gto.M(atom=geometry, basis="6-31g").apply(scf.RHF).run()
mc = mcscf.CASSCF(mf, ncas, nelecas)
mc.run()
mf.mo_coeff=mc.mo_coeff
```

(continues on next page)

(continued from previous page)

```
(pdm1, pdm2, pdm3, pdm4) = fci.rdm.make_dm1234(
    "FCI4pdm_kern_sf", mc.ci, mc.ci, ncas, nelecas
)

driver_casscf = ChemistryDriverPySCFMolecularRHF.from_mf(
    mf, frozen=FromActiveSpace(ncas, nelecas)(mf)
)

nevpt2_energy = driver_casscf.get_nevpt2_correction(rdm1, pdm2, pdm3, pdm4))
```

converged SCF energy = -14.8652654987828

CASSCF energy = -14.8887816132791

CASCI E = -14.8887816132791 E(CI) = -0.619242400904721 S^2 = 0.0000000

Sr (-1)', E = 0.000000000000064

Si (+1)', E = -0.00001592230011

Sijrs (0)', E = -0.00022778403082

Sijr (+1)', E = -0.00007024184177

Srsi (-1)', E = -0.00023752538104

Srs (-2)', E = -0.00127421589128

Sij (+2)', E = -0.00002668012632

Sir (0)', E = -0.00013325634708

Nevpt2 Energy = -0.001985625917782

The first-order expansion of the Adiabatic Connection for Complete Active Space wave functions (AC0-CAS) method [63], incorporating 1- and 2-particle reduced density matrices, utilizes the AC method and CAS approach within quantum chemistry. By employing the same driver, one can calculate the AC0 energy correction, utilizing the `get_ac0_correction` function where we specify the actual 1- and 2-particle density matrices converted from PDMs using the `reorder_dm12` function.

```
rdm1, rdm2 = fci.rdm.reorder_dm12(pdm1, pdm2)

ac0_energy = driver_casscf.get_ac0_correction((rdm1, rdm2))
```

Note

To perform the AC0 calculation, it is necessary to install pyscf-AC0. See here: [here](#).

An alternative approach would be to perform VQE calculations to obtain the PDMs and RDMs. A detailed tutorial including both approaches can be found [here](#).

23.8 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 types 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.

23.9 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.

23.9.1 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,
    ImpurityDMETROHFFragmentPySCFAactive
)

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)
```

23.9.2 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)

```

```
# STARTING CHEMICAL POTENTIAL 0.0
# STARTING CORR POTENTIAL PARAMETERS []
```

```
# FULL SCF ITERATION 0
```

```
# NEWTON ITERATION - CHEMICAL POTENTIAL 0.0
```

```
# FRAGMENT 0 - None: <H>=-301.7956772445178 EFR=-114.78345837272751 Q=-0.
→ 049838818654826866
```

```
# FRAGMENT 1 - None: <H>=-301.90654101502605 EFR=-152.88306683468392 Q=-0.
↪005690013605963884
```

```
# FRAGMENT 2 - None: <H>=-301.9101535819732 EFR=-150.83827026648663 Q=-0.
↪0014970562591898329
```

```
# FRAGMENT 3 - None: <H>=-301.94448653913275 EFR=-153.44135729451716 Q=-0.
↪00474978212623256
# NEWTON ITERATION - CHEMICAL POTENTIAL 0.0001
```

```
# FRAGMENT 0 - None: <H>=-301.79657624354576 EFR=-114.78406015411821 Q=-0.
↪049731861287206414
```

```
# FRAGMENT 1 - None: <H>=-301.90783538928997 EFR=-152.8847864254817 Q=-0.
↪005423346447219757
```

```
# FRAGMENT 2 - None: <H>=-301.9115539401107 EFR=-150.8396128507544 Q=-0.
↪0012572548818727824
```

```
# FRAGMENT 3 - None: <H>=-301.9458871008663 EFR=-153.44262085161134 Q=-0.
↪004532549348830628
# NEWTON ITERATION - CHEMICAL POTENTIAL 0.007436949983551479
```

```
# FRAGMENT 0 - None: <H>=-301.8625639773624 EFR=-114.82804988188985 Q=-0.
↪04191500418083649
```

```
# FRAGMENT 1 - None: <H>=-302.0028761208678 EFR=-153.0108811892299 Q=0.
↪014138160575370762
```

```
# FRAGMENT 2 - None: <H>=-302.01436028231785 EFR=-150.9380540703192 Q=0.
↪0163327315090811
```

```
# FRAGMENT 3 - None: <H>=-302.04870529435425 EFR=-153.53530162450159 Q=0.
↪011407964500470769
# CHEMICAL POTENTIAL 0.007441304209978171
# FINAL PARAMETERS: []
# FINAL CHEMICAL POTENTIAL: 0.007441304209978171
# FINAL ENERGY: -302.25869925300924
(-302.25869925300924, 0.007441304209978171, array([], dtype=float64))
```

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. However for `DMETRHF` it is important the the set of fragments completely cover the entire molecule.

23.10 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, `in quanto-pyscf` provides an easy way for a user to define a fragment solver class.

To create a DMET fragment, we subclass the `in quanto.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 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:

```
from in quanto.extensions.pyscf import ImpurityDMETROHFFragmentPySCFActive
from in quanto.operators import ChemistryRestrictedIntegralOperator
from in quanto.spaces import FermionSpace
from in quanto.states import FermionState
from in quanto.mappings import QubitMappingJordanWigner
from in quanto.ansatzes import FermionSpaceAnsatzUCCSD
from in quanto.computables import ExpectationValue
from in quanto.minimizers import MinimizerScipy
from in quanto.algorithms import AlgorithmVQE
from in quanto.protocols import SparseStatevectorProtocol
from pytket.extensions.qiskit import AerStateBackend

class MyFragment(ImpurityDMETROHFFragmentPySCFActive):

    def solve_active(
                      self,
                      hamiltonian_operator: ChemistryRestrictedIntegralOperator,
                      fermion_space: FermionSpace,
                      fermion_state: FermionState,
                      ):
        jw = QubitMappingJordanWigner()
```

(continues on next page)

(continued from previous page)

```

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_
↪objective=SparseStatevectorProtocol())

vqe.run()

energy = vqe.final_value

return energy

```

23.11 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 `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*.

23.12 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 `inquanto.extensions.pyscf.fmo.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.ansatzen 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 = [
```

(continues on next page)

(continued from previous page)

```

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)

```

```

# FIRST MONOMER ENERGIES: [-3.518038894864244, -2.886013952482359, -1.
˓→6863458716694653]

```

```

# CONVERGENCE REACHED!
# CONVERGED MONOMER ENERGIES: [-1.8009644337834727, -1.8134981363881528, -1.
˓→8009642542761393]
# TIMER BLOCK-0 BEGINS AT 2024-10-30 10:06:04.992607

```

```

# TIMER BLOCK-0 ENDS - DURATION (s): 0.5415383 [0:00:00.541538]
# TIMER BLOCK-1 BEGINS AT 2024-10-30 10:06:05.591074

```

```

# TIMER BLOCK-1 ENDS - DURATION (s): 0.3137772 [0:00:00.313777]
# TIMER BLOCK-2 BEGINS AT 2024-10-30 10:06:05.985275

```

```

# TIMER BLOCK-2 ENDS - DURATION (s): 0.5454638 [0:00:00.545464]
# FINAL MONOMER ENERGIES: [-1.8244970693394005, -1.83732927761619, -1.
˓→8244970519622772]

```

```
# TIMER BLOCK-3 BEGINS AT 2024-10-30 10:06:07.154235
```

```
# TIMER BLOCK-3 ENDS - DURATION (s): 12.8534136 [0:00:12.853414]
```

```
# TIMER BLOCK-4 BEGINS AT 2024-10-30 10:06:20.522593
```

```
# TIMER BLOCK-4 ENDS - DURATION (s): 11.7584136 [0:00:11.758414]
```

```
# TIMER BLOCK-5 BEGINS AT 2024-10-30 10:06:32.975592
```

```

# TIMER BLOCK-5 ENDS - DURATION (s): 17.2086885 [0:00:17.208689]
# FINAL DIMER ENERGIES: [-4.745456689215111, -4.189591290931866, -4.745456662426454]
# FINAL DIMER CORRECTION ENERGIES: [-1.0836303422595202, -0.5405971696301886, -1.
˓→0836303328479868]
# TOTAL ENERGY: -3.3512648044323488

```

```
-3.3512648044323488
```

23.13 QM/MM

In QM/MM methods, the energy is partitioned into quantum mechanical (QM) and molecular mechanical (MM) contributions [64] 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” [65]. In InQuanto the additive approach is adopted, which means the total energy takes the form

$$E_{\text{tot}} = E_{\text{QM}} + E_{\text{QM/MM}} + E_{\text{MM}} \quad (23.1)$$

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

$$h_{i,j} \rightarrow h_{i,j}^{\text{QM/MM}} = \int d\mathbf{x} \phi_i^*(\mathbf{x}) \left(\frac{-\nabla^2}{2} - \sum_A \frac{Z_A}{|\mathbf{R}_A - \mathbf{r}|} - \sum_M \frac{q_M}{|\mathbf{R}_M - \mathbf{r}|} \right) \phi_j(\mathbf{x}) \quad (23.2)$$

where $\mathbf{x} = \{\mathbf{r}, s\}$ with \mathbf{r} (s) the electronic position (spin), and the modified classical Coulomb interaction for nuclei becomes

$$V_{\text{nuc}} \rightarrow V_{\text{nuc}}^{\text{QM/MM}} = \sum_{A,B} \frac{Z_A Z_B}{|\mathbf{R}_A - \mathbf{R}_B|} + \sum_{A,M} \frac{Z_A q_M}{|\mathbf{R}_A - \mathbf{R}_M|} \quad (23.3)$$

where M labels the MM point charges, with charge q_M and position \mathbf{R}_M .

In `inquito-pyscf` the specialized driver class: `ChemistryDriverPySCFMolecularRHFQMMMCOSMO`, uses the QM/MM scheme of PySCF to modify the one-body integrals and nuclear Coulomb interaction. With this driver, QM/MM embedding is performed as follows; we consider an H₂ molecule surrounded by ten random point charges:

```
from inquito.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 numpy.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.

23.14 COSMO

The COSMO (COnductor-like Screening MOdel) scheme [66] 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 utilized. In this scheme, the mean-field SCF problem is solved with a modified Hamiltonian

$$E_{\text{SCF}} = \langle \Psi | \hat{H} + \hat{V}_{\text{ddCOSMO}} | \Psi \rangle \quad (23.4)$$

Where \hat{V}_{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

$$E_{\text{ddCOSMO}} = \frac{1}{2} f(\epsilon_s) \int_{\Omega} \rho(\mathbf{r}) W(\mathbf{r}) d\mathbf{r} \quad (23.5)$$

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 [67].

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:

$$E_{\text{tot}} = \langle \Psi | \hat{H} + \hat{V}_{\text{ddCOSMO}} | \Psi \rangle - \text{Tr}(D \hat{V}_{\text{ddCOSMO}}) + E_{\text{ddCOSMO}} \quad (23.6)$$

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 proceed as usual. The energy `mf_solvent_e_tot` corresponds to the mean field energy with the COSMO corrections as specified in (23.6).

It is also possible to combine COSMO with other embedding schemes in InQuanto by setting `do_cosmo=True` in the driver as above.

CHAPTER TWENTYFOUR

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.

24.1 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:

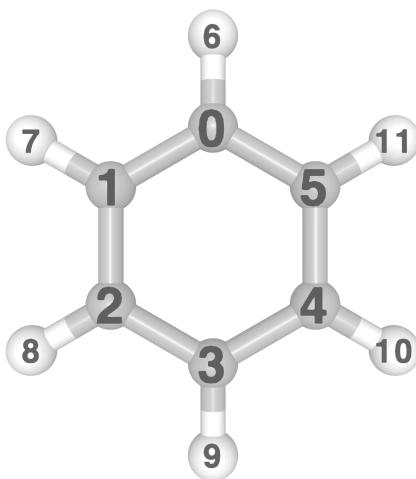
```
from inquanto.geometries import GeometryMolecular
from inquanto.extensions.nglview import VisualizerNGL

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

# 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()

```

24.2 Visualizing Fragments

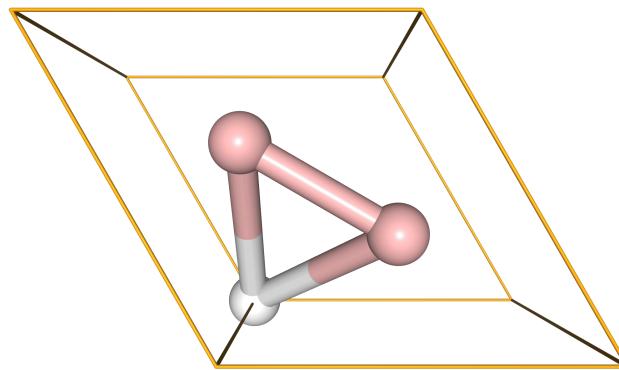
There are several *fragmentation methods* available in InQuanto; to visualize a fragmentation scheme defined in a `GeometryMolecular` object, use the `visualize_fragmentation()` method:

```

c6h6_geom.set_groups(
    "fragments",
    {
        "ch1": [0, 6],
        "ch2": [5, 11],
        "ch3": [4, 10],
    }
)

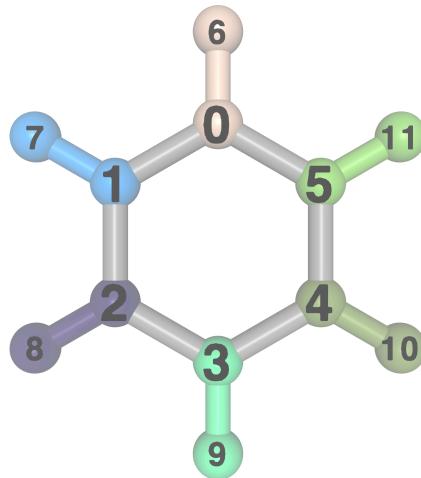
```

(continues on next page)



(continued from previous page)

```
"ch4": [3, 9],  
"ch5": [2, 8],  
"ch6": [1, 7]  
}  
}  
visualizer.visualize_fragmentation("fragments", atom_labels="index")
```

**Note**

Visualizing fragments is not yet supported for periodic geometries.

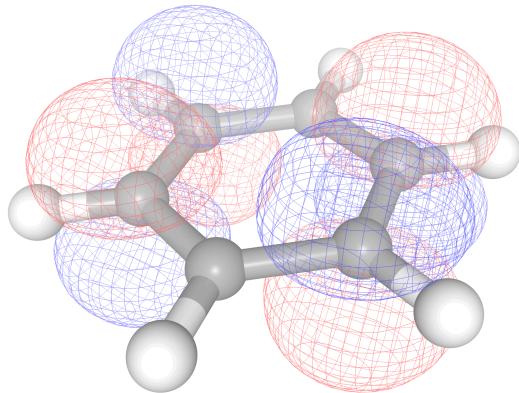
24.3 Visualizing Orbitals

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]
```



Finally we note that extended NGLView options can be exposed by rendering its graphical user interface (GUI) using the following option:

```
g = GeometryMolecular(xyz)
visualizer = VisualizerNGL(g)
visualizer.visualize_molecule().display(gui=True)
```

CHAPTER
TWENTYFIVE

INQUANTO-PHAYES

The *InQuanto-Phayes* extension provides an interface to `phayes` package that performs one flavor of Bayesian quantum phase estimation (QPE) introduced by K. Yamamoto, S. Duffield, Y. Kikuchi, and D. Muñoz Ramo [20] using the QPE circuit protocols implemented into InQuanto. See [Iterative Phase Estimation Algorithms](#) for the introduction to the other iterative QPE algorithms available in InQuanto.

The basic idea of Bayesian QPE is, in general, to update the prior distribution of the phase ϕ by the Bayesian update rule

$$P(\phi|m, k, \beta) \propto P(m|\phi, k, \beta)P(\phi), \quad (25.1)$$

where $P(\phi|m, k, \beta)$ and $P(m|\phi, k, \beta)$ denote the posterior and likelihood, respectively. The parameters k and β are generated for each update. Bayesian QPE may be more efficient than the maximum likelihood approach, as the parameter selection can reflect the prior $P(\phi)$. The Bayesian QPE implemented as `AlgorithmBayesianQPE` has the following features:

1. Representation of the prior $P(\phi)$ switches from Fourier to von Mises (wrapped Gaussian) automatically.
2. The parameters k and β are selected optimally from the prior.

See Ref. [20] for the technical details. Below is a simple example demonstrating the usage of `AlgorithmBayesianQPE`.

As in the usage of the other iterative QPE algorithm classes, we prepare the `IterativePhaseEstimationStatevector` object to handle the quantum circuits generated using the chemistry input.

```
from inquanto.express import get_system
from inquanto.mappings import QubitMappingJordanWigner
from inquanto.ansatzes import FermionSpaceAnsatzUCCSD
from inquanto.protocols import IterativePhaseEstimationStatevector

# Generate the spin Hamiltonian from the molecular Hamiltonian.
target_data = "h2_sto3g.h5"
fermion_hamiltonian, fermion_fock_space, fermion_state = get_system(target_data)
mapping = QubitMappingJordanWigner()
qubit_hamiltonian = mapping.operator_map(fermion_hamiltonian)

# Generate a list of qubit operators as exponents to be trotterized.
qubit_operator_list = qubit_hamiltonian.trotterize(trotter_number=1)

# The parameter `t` is physically recognized as the duration time in atomic units.
time = 0.25

# Preliminary calculated parameters.
ansatz_parameters = [-0.107, 0., 0.]
```

(continues on next page)

(continued from previous page)

```
# Generate a non-symbolic ansatz.
ansatz = FermionSpaceAnsatzUCCSD(fermion_fock_space, fermion_state, mapping)
parameters = dict(zip(ansatz.state_symbols, ansatz_parameters))
state_prep = ansatz.subs(parameters)

# Construct the protocol to handle the iterative QPE circuits.
protocol = IterativePhaseEstimationStatevector(
).build(
    state=state_prep,
    evolution_operator_exponents=qubit_operator_list * time,
)
```

Then, we prepare the algorithm class.

```
import phayes
from inquanto.extensions.phayes import AlgorithmBayesianQPE
# This is needed for the numerical stability.
from jax import config
config.update("jax_enable_x64", True)

# Prepare the phayes state.
phayes_state = phayes.init(J=1000)
k_max = 300

algorithm = AlgorithmBayesianQPE(
    phayes_state=phayes_state,
    k_max=k_max,
).build(protocol)
```

`phayes_state` represents the prior distribution, either in the Fourier or von Mises representation. `k_max` designates the cap of k . The algorithm will not attempt to choose a higher value of k than this setting, in order to limit the maximum circuit depth. In contrast to `AlgorithmInfoTheoryQPE` and `AlgorithmKitaevQPE`, `AlgorithmBayesianQPE` requires a feedback loop. Driving such a feedback loop simply requires invoking the `run()` method as many times as the number of desired Bayesian updates.

```
import numpy
import matplotlib.pyplot as plt

# Number of Bayesian updates.
n_update = 50

# Execute the Bayesian QPE
for i in range(n_update):

    # Perform the single Bayesian update with `n_shots` of measurement samples.
    algorithm.run()

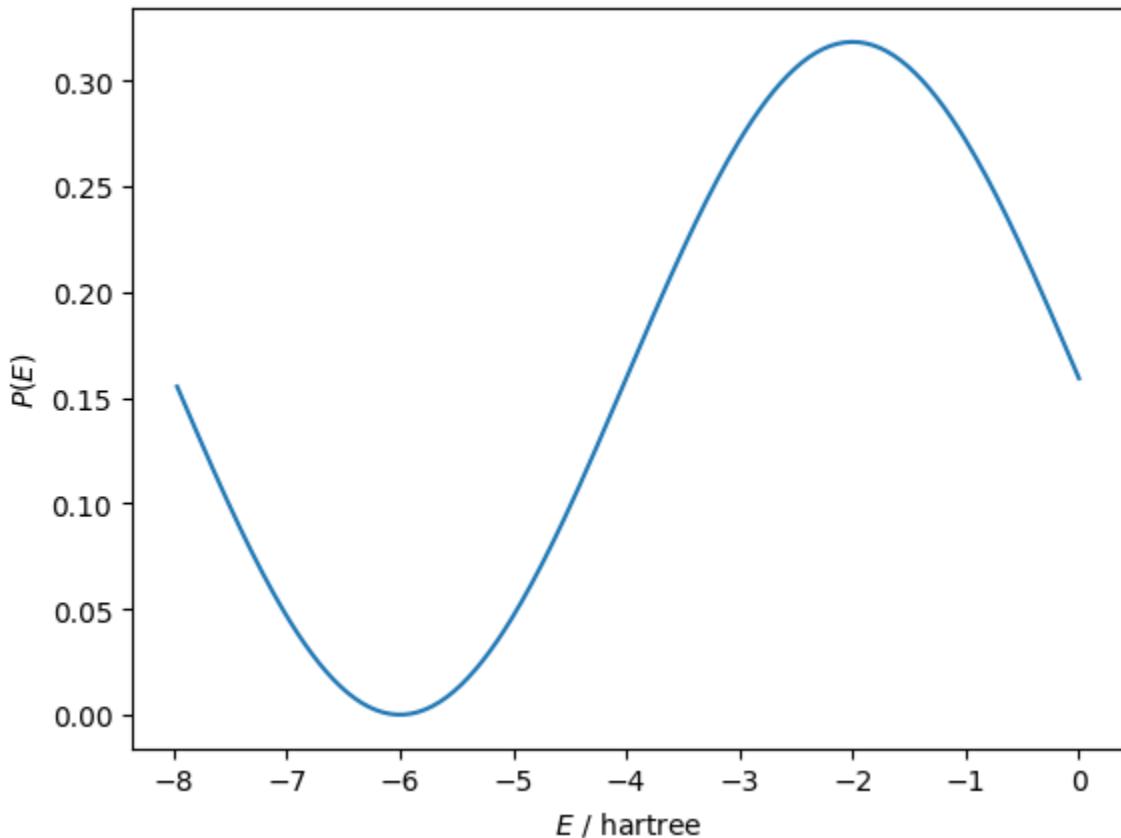
    # Show the current prior.
    if i % 10 == 0:
        mu, sigma = algorithm.final_value()
        mode = {True: 'Fourier', False: 'von Mises'}[bool(algorithm.phayes_state.
        fourier_mode)]
```

(continues on next page)

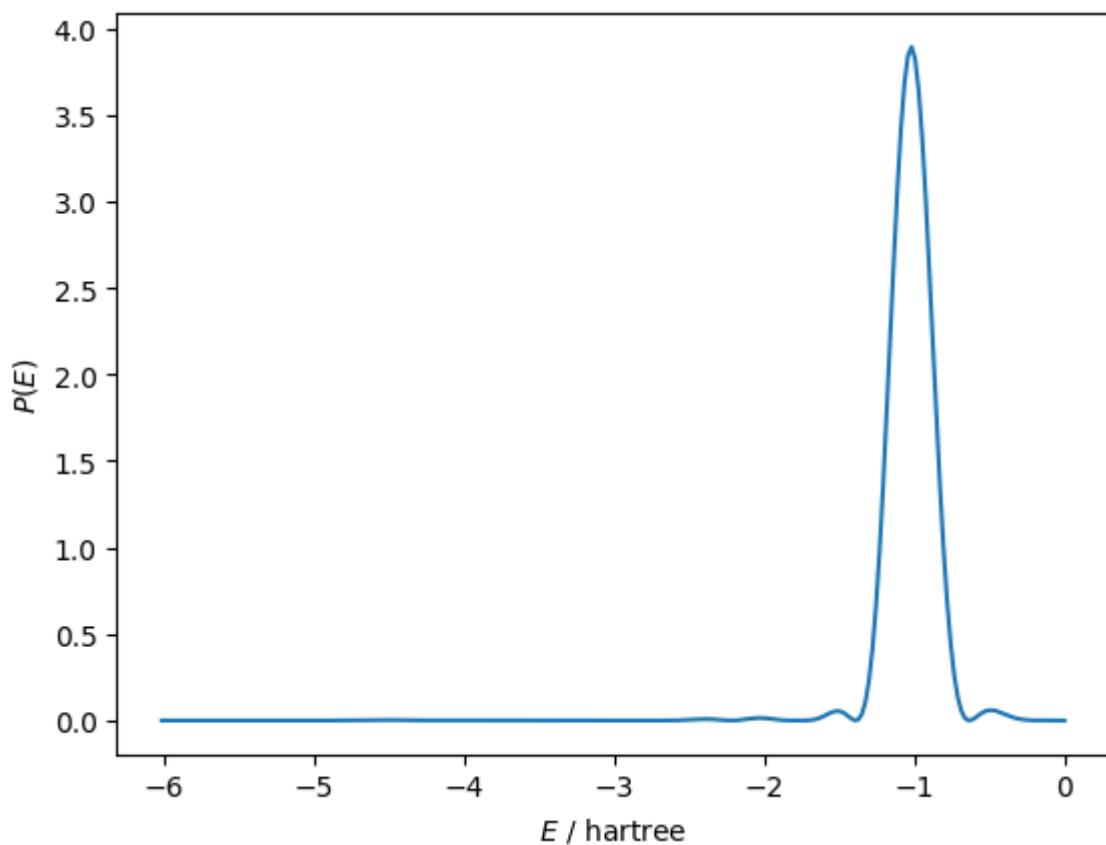
(continued from previous page)

```
print(f"i_update      = {i+1}")
print(f"representation = {mode}")
xmin = max(mu - 30 * sigma, 0)
xmax = min(mu + 30 * sigma, 2)
phi = numpy.linspace(xmin, xmax, 256)[-1]
plt.plot(-phi / time, algorithm.final_pdf(phi))
plt.xlabel("$E$ / hartree")
plt.ylabel("$P(E)$")
plt.show()
```

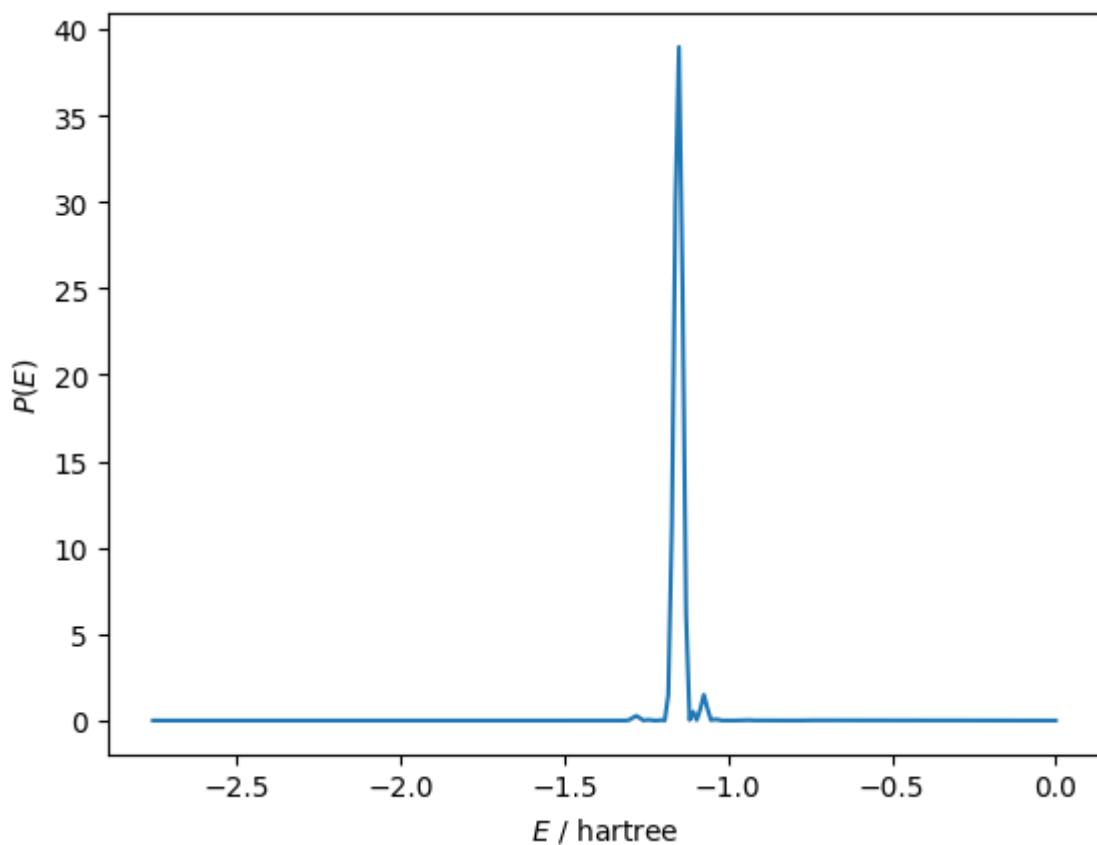
```
i_update      = 1
representation = Fourier
```



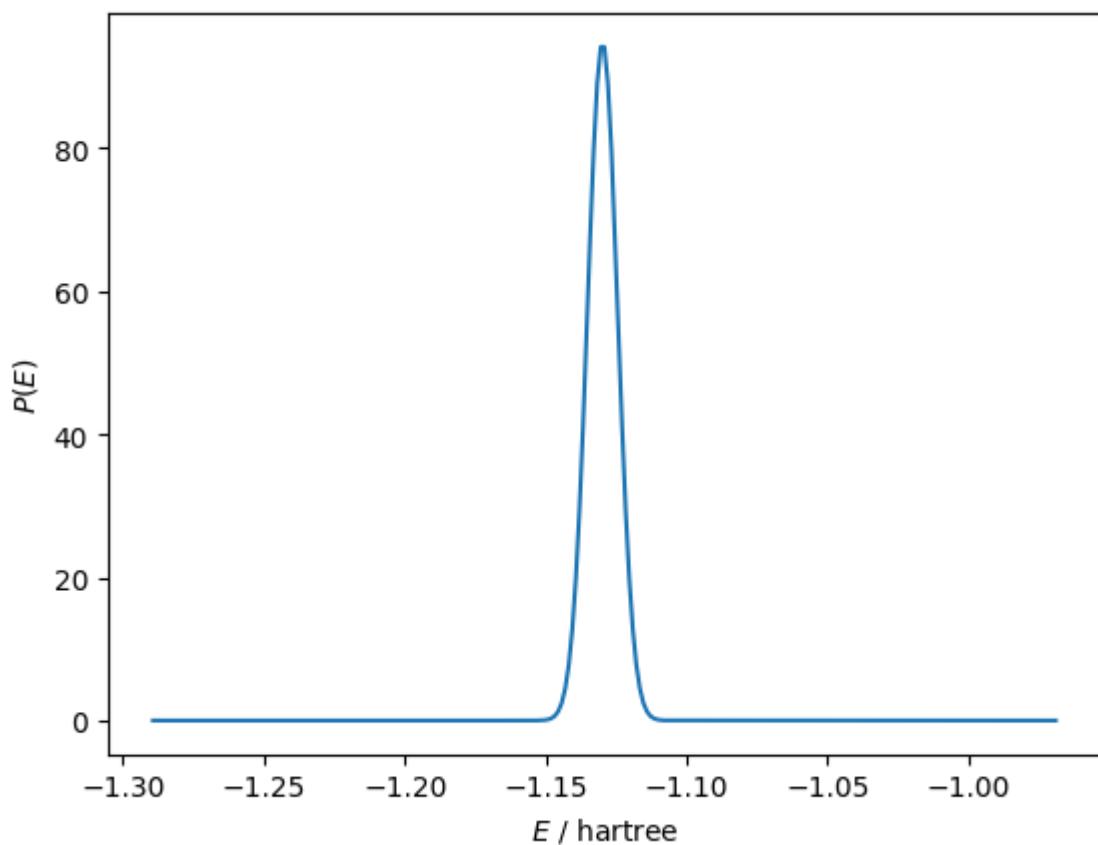
```
i_update      = 11
representation = Fourier
```



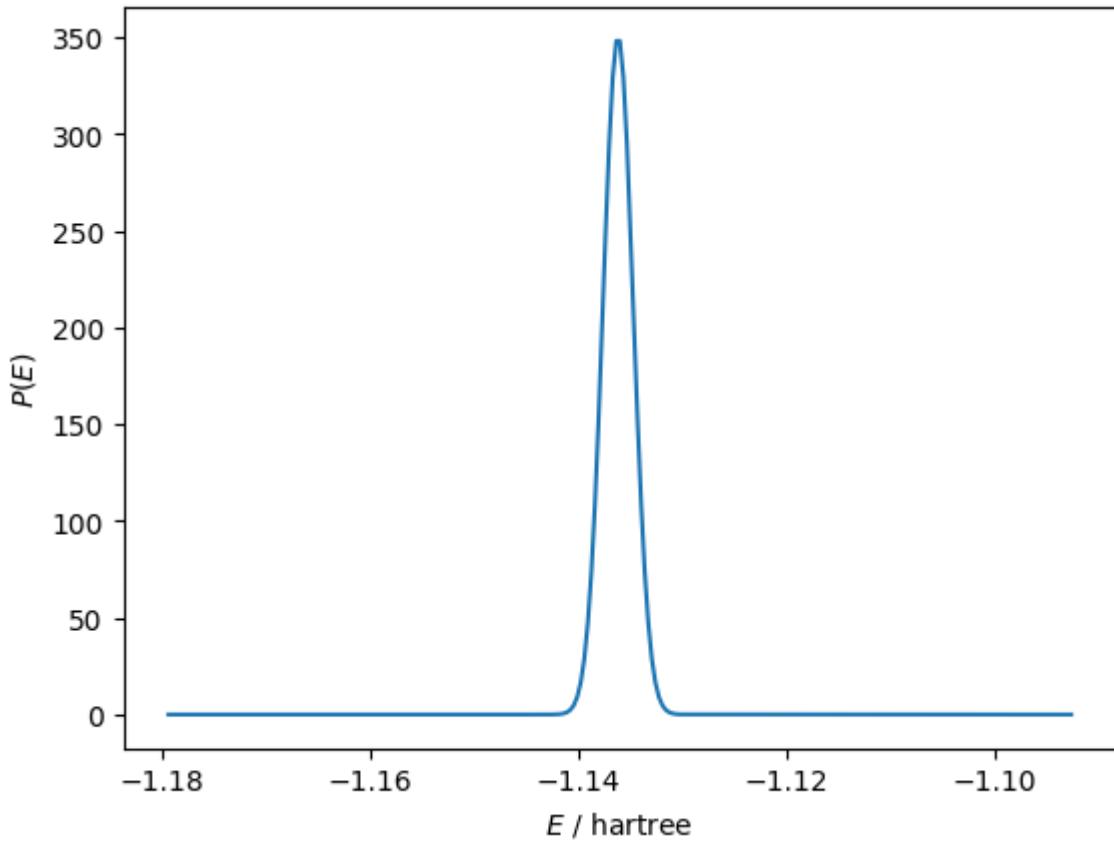
```
i_update      = 21
representation = Fourier
```



```
i_update      = 31
representation = von Mises
```



```
i_update      = 41
representation = von Mises
```



Note that the representation of the prior switches from Fourier to von Mises automatically. Now, the energy estimate is shown as a result of Bayesian QPE.

```
# Display the final results.
mu, sigma = algorithm.final_value()
energy_mu = -mu / time
energy_sigma = sigma / time
mode = {True: 'Fourier', False: 'von Mises'}[bool(algorithm.phayes_state.fourier_
←mode)]
print(f"i_update      = {i+1}")
print(f"Energy(mu)    = {energy_mu:10.6f} hartree")
print(f"Energy(sigma) = {energy_sigma:10.6f} hartree")
print(f"Representation = {mode}")
```

```
i_update      = 50
Energy(mu)    = -1.136461 hartree
Energy(sigma) = 0.001078 hartree
Representation = von Mises
```

OVERVIEW OF INQUANTO EXTENSIONS EXAMPLES

This page lists examples for the InQuanto-PySCF, InQuanto-NGLView and InQuanto-Phayes packages.

26.1 inquanto-pyscf/drivers

Examples demonstrating basic functionality of InQuanto-PySCF drivers.

File	Description
algorithm_vqe_ccsd_amplit.py	A canonical VQE simulation of OH(-) using (classical) CCSD or MP2 amplitudes.
algorithm_vqe_mp2_amplitu.py	A canonical VQE simulation of OH(-) using (classical) MP2 amplitudes.
statevector_accelerated_v.py	LiH VQE simulation with and without gradients to demonstrate acceleration
double_factorization_n2_p.py	Get a double factorized hamiltonian from the PySCF extension, calculate energy, and compare to standard approach.

26.2 inquanto-pyscf/embeddings

Examples using PySCF with InQuanto's DMET classes.

File	Description
impurity_dmet_h7_pyscf.py	An ImpurityDMET example simulating a 7-hydrogen chain.
impurity_dmet_ch2_pyscf.py	An example for running impurity DMET on triplet CH2.
impurity_dmet_h2x3_express_pyscf.py	An Impurity DMET example for simulating a 3-dihydrogen ring.
dmet_one_phenol_vqe_pyscf.py	An example DMET calculation simulating phenol

26.3 inquanto-pyscf/fmo

Examples of usage of InQuanto-PySCF FMO classes.

File	Description
fmo_h2x3_integral_operator.py	An example FMO simulation of a 3-dihydrogen chain.
fmo_h2x3_integral_operator_VQE.py	An example FMO simulation of a 3-dihydrogen chain with custom VQE fragment solver.
fmo_lih_h2x2_sto3g_integral_op.py	An example FMO simulation of a LiH and two H2 with CCSD fragment solver.
fmo_h2x3_631g_integral_operator.py	An example FMO simulation of a 3-dihydrogen chain with 631G basis.
fmo_h2x3_integral_operator_VQE.py	An example Fragment molecular orbital(FMO) simulation of a 3-dihydrogen chain with custom VQE fragment solver.
fmo_li2_lih_h2_sto3g_integral_.py	An example FMO simulation of a LiH and H2.
fmo_h2x3_integral_operator_CCS.py	An example FMO simulation of a 3-dihydrogen chain with CCSD fragment solver.
fmo_lih_h2x2_sto3g_integral_op.py	An example FMO simulation of a LiH and two H2.

26.4 inquanto-pyscf/projection_embedding

Examples of usage of InQuanto-PySCF Projection Embedding classes.

File	Description
pt2o5_vqe_in_b3lyp.py	An example using dft projection based embedding of Pt5O2 with VQE.

26.5 inquanto-pyscf/symmetry

Examples of usage of symmetry in InQuanto-PySCF.

File	Description
tapering_pmsv.py	Using Hamiltonian symmetries to reduce the qubit count with tapering, and perform noise mitigation with PMSV.

26.6 inquanto-phayes/algorithm

Using InQuanto-Phayes for Bayesian QPE

File	Description
algorithm_bayesian_noise_aware.py	BQPE calculation using a noise aware prior.
algorithm_bayesian.py	Bayesian QPE protocol for noiseless backends
algorithm_bayesian_quatinuum_iceberg.py	BQPE calculation using H1 device and the iceberg detection code.

26.7 inquanto-nglview/nglview

Vizualizing structures and orbitals with InQuanto-NGLView

File	Description
<code>example_unitcell.py</code>	Example inquanto-nglview visualization of periodic systems
<code>example_molecule.py</code>	Example inquanto-nglview visualization of molecular systems

INQUANTO API REFERENCE

27.1 inquanto.algorithms

27.1.1 Variational

```
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 a 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 [Trotter-Ansatz](#) 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** ([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(*protocol_expectation*, *protocol_pool_metric*, *protocol_gradient*=`None`)

Build the algorithm using the provided protocols.

Parameters

- **protocol_expectation** ([EvaluatorRunnerMixin](#)) – The protocol used for expectation value calculation.

- **protocol_pool_metric** (`EvaluatorRunnerMixin`) – The protocol used to determine the excitation selection metric.
- **protocol_gradient** (`Optional[EvaluatorRunnerMixin]`, default: `None`) – The protocol used for gradient calculation.

Returns`AlgorithmAdaptVQE` – self**property final_parameters: `SymbolDict`**

Returns the optimised ansatz parameters.

generate_report()

Return a dictionary giving experimental results.

Return type`dict`**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, `FermionSpaceStateExp` or `FermionSpaceStateExpChemicallyAware`.

Returns`Union[FermionSpaceStateExp, FermionSpaceStateExpChemicallyAware]` – A new ansatz object.**get_exponents_with_symbols()**

Returns a symbol-containing sublist of the fermionic pool operator list.

Return type`FermionOperatorList`**run()**

Perform the ADAPT experiment.

After execution, a `TrotterAnsatz` instance, optimized energy and final parameters will be available.**Return type**`None`

```
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 to 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** (*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 (*protocol_expectation*, *protocol_pool_metric*, *protocol_gradient=None*)

Build the algorithm using the provided protocols.

Parameters

- **protocol_expectation** (*EvaluatorRunnerMixin*) – The protocol used for expectation value calculation.
- **protocol_pool_metric** (*EvaluatorRunnerMixin*) – The protocol used to determine the excitation selection metric.
- **protocol_gradient** (*Optional[EvaluatorRunnerMixin]*, default: `None`) – The protocol used for gradient calculation.

Returns

AlgorithmAdaptVQE – self

property final_parameters: *SymbolDict*

Returns the optimised ansatz parameters.

generate_report()

Return a dictionary giving experimental results.

Return type

dict

run()

Perform the ADAPT experiment.

After execution, a *TrotterAnsatz* instance, optimized energy and final parameters will be available.

Return type

None

```
class AlgorithmIQEB(pool, state, hamiltonian, minimizer, 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.

The Iterative Qubit-Excitation-Based (IQEB) 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 the pool of operators. The ParaFermionSpace generators are used with this class.

Parameters

- **pool** (*QubitOperatorList*) – Holds the pool of qubit excitation 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.
- **n_iterations** (*int*, default: 100) – Number of iterations before termination.
- **n_grads** (*int*, default: 10) – Number of terms to narrow down the pool, based on gradients. Each IQEB iteration will run *n_grads* VQEs, one VQE for each of the *n_grads* largest gradient terms.
- **energy_tolerance** (*float*, default: $1.0e-10$) – Condition of termination of IQEB algorithm. Corresponds to threshold energy difference between IQEB iterations. The 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 (*protocol_expectation*, *protocol_pool_metric*, *protocol_gradient=None*)

Build the algorithm using the provided protocols.

Parameters

- **protocol_expectation** (*EvaluatorRunnerMixin*) – The protocol used for expectation value calculation.
- **protocol_pool_metric** (*EvaluatorRunnerMixin*) – The protocol used to determine the excitation selection metric.
- **protocol_gradient** (*Optional[EvaluatorRunnerMixin]*, default: `None`) – The protocol used for gradient calculation.

Returns

AlgorithmAdaptVQE – self

property final_parameters: *SymbolDict*

Returns the optimised ansatz parameters.

generate_report()

Return a dictionary of results after running the algorithm.

Return type

dict

run (*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 VQE 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.

Each IQEB iteration is indexed by m , while each VQE (inside an IQEB iteration) is indexed by p .

After execution, a `TrotterAnsatz` instance, optimized energy and final parameters will be available.

The parameter symbols (r) of terms in the ansatz are labelled as $r_m_p(\text{sc})$, where ‘sc’ stands for spin complement.

Parameters

- `compiler_passes` (`Optional[BasePass]`, default: `None`) – Circuit optimisation pass applied at each iteration of the algorithm.

Return type

`None`

```
class AlgorithmVQE(minimizer, objective_expression, *, gradient_expression=None, initial_parameters=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` (`Optional[SymbolDict]`, default: `None`) – A set of initial ansatz parameters.
- `gradient_expression` (`Optional[ExpectationValueDerivative]`, default: `None`) – An expression to evaluate the gradient of the objective function.
- `auxiliary_expression` (`Optional[ComputableNode]`, default: `None`) – Additional expressions to evaluate alongside the energy.

```
build(protocol_objective, protocol_gradient=None)
```

Build the VQE experiment.

Provide objects required to estimate the expectation value of the ansatz, and gradient estimation used for minimization.

Parameters

- `protocol_objective` (`EvaluatorRunnerMixin`) – The protocol used for energy expectation value estimation.
- `protocol_gradient` (`Optional[EvaluatorRunnerMixin]`, default: `None`) – The protocol used for energy gradient estimation.

Returns

`AlgorithmVQE` – self

```
property final_evaluated_auxiliary_expression: tuple[float, RestrictedOneBodyRDM]
```

Expectation value result of VQE experiment on auxiliary expression.

```
property final_evaluated_objective_expression: float
```

Expectation value of the objective function using the final set of parameters.

```

property final_parameters: SymbolDict
    Parameters used to evaluate final expectation value.

property final_value: float
    Expectation value result of VQE experiment.

generate_report()
    Return a dictionary giving experimental results.

    Return type
        dict

run()
    Perform the VQE experiment.

    Results may be queried with generate_report()

    Return type
        AlgorithmVQE

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 (*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

- **objective_protocol** (*EvaluatorRunnerMixin*) – The protocol used for energy expectation value estimation.

- **weight_protocol** (`EvaluatorRunnerMixin`) – The protocol used to estimate the weighting of eigenstates of the Hamiltonian.
- **overlap_protocol** (`EvaluatorRunnerMixin`) – 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: list[SymbolDict]**

Parameters used to evaluate final expectation value.

property final_values: list[float]

Expectation value result of VQD experiment.

generate_report()

Return a dictionary giving experimental results.

Return type`dict`**run()**

Perform the VQD experiment.

Results can be queried with `generate_report()`.**Returns**`AlgorithmVQD` – self

27.1.2 Non-variational

class AlgorithmQSE(computable_qse_matrices, parameters)Bases: `object`

Quantum Subspace Expansion algorithm.

The implementation here is based on work in arXiv:1603.05681.

Parameters

- **computable_qse_matrices** (`QSEMatricesComputable`) – Computable to return the matrix representation of a hermitian operator, H and overlap matrix, S .
- **parameters** (`SymbolDict`) – Circuit parameters from which the subspace is generated.

build(protocol)

Build the algorithm using the provided protocols.

Parameters`protocol` (`EvaluatorRunnerMixin`) – The protocol to evaluate matrix elements.**Returns**`AlgorithmQSE` – self**property final_states**

Final states in QSE computable.

property final_values

Final expectation values of QSE computable.

```
generate_report ()
    Return a dictionary giving experimental results.

    Return type
        dict

run ()
    Perform the QSE experiment. Results can be queried with generate_report ().

    Return type
        None

class AlgorithmSCEOM (computable_sceom_matrix, parameters=None)
Bases: object
Quantum Self Consistent Equation of Motion.

The implementation here is based on work in https://doi.org/10.1039/D2SC05371C.
```

Parameters

- **computable_sceom_matrix** (*SCEOMMatrixComputable*) – Computable to return the matrix representation of a Hermitian operator with respect to the correlated excited states, M matrix.
- **parameters** (`Optional[SymbolDict]`, default: `None`) – Circuit parameters from VQE run.

build(protocol)

Build the algorithm using the provided protocols.

Parameters

- **objective_protocol** – The protocol to evaluate matrix elements.
- **protocol** (`EvaluatorRunnerMixin`)

Returns

AlgorithmSCEOM – self

property final_states

Final states in QSCEOM computable.

property final_values

Final eigenvalues of QSCEOM computable.

generate_report ()

Return a dictionary giving experimental results.

Return type

dict

get_dataframe_sceom_analysis ()

Returns a dataframe containing expectation values and overlaps with SCEOM states. :rtype: DataFrame

 **Note**

The expectation values and overlaps are calculated with the `default_evaluate ()` computable method.

⚠ Warning

This feature is under development due to numerical instabilities issues.

Raises

`RuntimeError` – If `:meth:final_states` is None.

`print_sceom_states()`

Prints the SCEOM excited states.

Return type

`None`

`run()`

Perform the SCEOM experiment. Results can be queried with `generate_report()`.

Return type

`None`

27.1.3 Phase estimation

`class AlgorithmDeterministicQPE(Ansatz, evolution_operator_exponents)`

Bases: `object`

Quantum Phase Estimation (QPE), for estimating the eigenstate energies of molecular Hamiltonians.

This class corresponds to “deterministic” forms of QPE - those which do not rely on stochasticity or statistical inference based on measurement outcomes. Examples include canonical QPE and certain variants of iterative QPE.

To use this algorithm, provide an ansatz object and a `QubitOperatorList` corresponding to the unitary evolution operator of the Hamiltonian to be simulated. The ansatz object is used for state preparation, and should be a state which is likely to have high overlap with the true eigenstate of interest. The `QubitOperatorList` must have terms within it corresponding to individual exponents of an exponential product which may be simulated on a quantum computer. Typically, this will mean that the `QubitOperatorList` will be generated through Trotterization. The `QubitOperatorList` must be scaled through multiplying by a total evolution time t , such that the obtained phase will be in the interval $(0, 2\pi]$.

Once created, the computational primitives must be built with the `build()` method. Here, a protocol must be specified providing detail of how to create the specific quantum circuits corresponding to a particular form of QPE. For instance, `CanonicalPhaseEstimation` will create canonical QPE circuits. See the protocol documentation, or the examples, for details as to which protocols are available.

Please see the documentation for example usage.

Parameters

- `ansatz` (`GeneralAnsatz`) – An ansatz state to be used for the state preparation.
- `evolution_operator_exponents` (`QubitOperatorList`) – A list of exponents corresponding to an exponential product representing the evolution operator of the Hamiltonian of interest. This will typically be generated through Trotterization.

Notes

The value of the parameter t of the time evolution operator e^{-iHt} must be given in a unit of half turn, i.e., $t = \text{time} * \pi$ to be compatible with the pytket convention.

`build(protocol, phase_calculation_protocol=None)`

Build the QPE algorithm using the specified protocols and computational details.

Here, a Protocol object must be specified identifying how the relevant quantum circuits are to be constructed. These will typically correspond to a particular “flavour” of quantum phase estimation - for instance, `CanonicalPhaseEstimation` for canonical phase estimation. Optionally, a second Protocol derived from `PhaseEstimator` may be specified, to control how a phase is determined from the observed experimental results. By default, this will proceed by taking the most likely measurement outcome.

Parameters

- `protocol` (`ProtocolQPE`) – A computational protocol defining how the QPE circuits are to be built, typically corresponding to a “flavour” of QPE.
- `phase_calculation_protocol` (`Optional[PhaseEstimator]`, default: `None`) – An optional protocol, derived from `PhaseEstimator`, used for determining the phase from the raw measurement outcomes. By default, this will calculate the phase from the most common measurement outcome.

Returns

`AlgorithmDeterministicQPE` – self.

`final_energy(time, phase_estimator_protocol=None)`

Returns the energy, given a specified total evolution time.

Parameters

- `time` (`float`) – The total evolution time of the original unitary evolution operator.
- `phase_estimator_protocol` (`Optional[PhaseEstimator]`, default: `None`) – An optional protocol used to override how the phase is derived from raw measurement outcomes. By default, this will use the protocol specified at build-time.

Returns

`float` – The calculated energy, given the provided total evolution time.

`final_phase(phase_estimator_protocol=None)`

Returns the final phase estimate given by the algorithm run.

Parameters

- `phase_estimator_protocol` (`Optional[PhaseEstimator]`, default: `None`) – An optional protocol used to override how the phase is derived from raw measurement outcomes. By default, this will use the protocol specified at build-time.

Returns

The calculated phase.

`generate_report()`

Return a dictionary giving experimental results.

Return type

`dict[str, Any]`

`launch_experiment()`

Launch the jobs through the protocol object asynchronously.

This method may be useful when the job is run on a remote backend.

Returns

`list[ResultHandle]` – List of result handles.

retrieve_experiment (handles_list=None)

Retrieve the backend results through the protocol.

Parameters

- `handles_list` (`Union[list[tuple[int, float, list[ResultHandle]]], list[tuple[int, float, list[BackendResult]]], None]`, default: `None`) – An optional list of experiment handles - if not passed, stored handles from this
- `used.` (*object will be*)

Returns

`AlgorithmDeterministicQPE` – self.

Raises

`RuntimeError` – If the experiment has not been launched.

run ()

Perform the QPE experiment.

Results may be queried with `generate_report()`, `final_phase()` or `final_energy()`.

Returns

`AlgorithmDeterministicQPE` – self.

run_experiment ()

Perform the QPE experiment.

Results may be queried with `generate_report()`, `final_phase()` or `final_energy()`.

Returns

`AlgorithmDeterministicQPE` – self.

class AlgorithmInfoTheoryQPE (resolution, k_max, n_samples, prior=None, error_rate=None, verbose=0)

Bases: `object`

Execute information theory QPE algorithm.

Reference:

- K. M. Svore, M. B. Hastings, and M. Freedman, [arXiv:1304.0741](#).

This algorithm class performs maximum likelihood estimation of the phase ϕ .

The noise-aware likelihood function is expressed as

$$P(m|\phi, k, \beta) = \frac{1 + (1 - q(k))(-1)^m \cos(k\phi + \beta)}{2} \quad (27.1)$$

where $q(k) \in [0, 1]$ is the error rate as a function of k .

Parameters

- `resolution` (`int`) – Number of grid points to resolve the phase.
- `k_max` (`int`) – Cap of the circuit depth parameter k .
- `n_samples` (`int`) – Number of measurement samples.
- `prior` (`Optional[ndarray]`, default: `None`) – Prior distribution. If not given, uniform distribution is used.

- **error_rate** (`Optional[Callable[[int], float]]`, default: `None`) – Error rate used for the noise-aware likelihood. If not given, $q(k) = 0$ is used.
- **verbose** (`int`, default: 0) – Control verbosity.

`build(protocol)`

Set the protocol and build the algorithm.

Parameters

`protocol` (`BaseIterativePhaseEstimation`) – Iterative QPE protocol.

Returns

`AlgorithmInfoTheoryQPE` – self.

`final_pdf(phi)`

Return the PDF as a function of phase.

Parameters

`phi` (`ndarray`) – Grid representation of the phase in [0, 2) (pytket convention).

Returns

`ndarray` – Probability distribution function.

`final_value()`

Return the phase estimate in the unit of half turn.

Returns

`tuple[float, float]` – Mean μ and standard deviation σ .

`join(handles_list)`

Retrieve the results through the protocol.

Parameters

`handles_list` (`Union[list[tuple[int, float, list[ResultHandle]]], list[tuple[int, float, list[BackendResult]]]]`) – List of (k, beta, handles/handles).

Return type

`None`

`run()`

Run the algorithm.

`run_async()`

Launch the jobs through the protocol object asynchronously.

This API may be useful when the job is run on a remote backend.

Returns

`list[tuple[int, float, list[ResultHandle]]]` – List of (k, beta, handles/results).

`class AlgorithmKitaevQPE(n_bits, n_extra_bits=2, verbose=0)`

Bases: `object`

Execute Kitaev's QPE algorithm.

- A. Yu. Kitaev, [arXiv:quant-ph/9511026](https://arxiv.org/abs/quant-ph/9511026).

The parameter `n_bits` governs the highest unitary power that will be applied in the iterative procedure i.e. if `n_bits` = k , then the largest circuit will involve the application of U^{2^k} . Obtaining a more precise estimate of the phase is possible with the `n_extra_bits` parameter; in this case the circuits will be sampled repeatedly to obtain a result with `n_bits` + `n_extra_bits` bits of precision. Note that sampling in this way scales unfavourably, and many shots may be required to obtain extra precision in this manner.

Parameters

- **n_bits** (`int`) – Number of bits to be estimated.
- **n_extra_bits** (`int`, default: 2) – Number of extra bits to be estimated.
- **verbose** (`int`, default: 0) – Verbosity level.

build(protocol)

Set the protocol and build the algorithm.

Parameters

protocol (`BaseIterativePhaseEstimation`) – Iterative QPE protocol.

Returns

`AlgorithmKitaevQPE` – self

final_value()

Final value of the phase estimate.

Returns

`tuple[float, float]` – Phase estimate and the precision.

join(handles_mapping)

Retrieve the backend results through the protocol.

Parameters

- **handles_list** – List of (k, beta, handles).
- **handles_mapping** (`Union[list[tuple[int, float, list[ResultHandle]]], list[tuple[int, float, list[BackendResult]]]]`)

Return type

`None`

run()

Run the algorithm.

Return type

`None`

run_async()

Run the jobs asynchronously.

This method may be useful when the job is run on a remote backend.

Returns

`list[tuple[int, float, list[ResultHandle]]]` – List of job IDs (k, beta, handles/results).

27.1.4 Time evolution

`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** (*ComputableNode*) – 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(*protocol*)

Executes the build method of a computable expression.

Parameters

- **backend** – Backend for the computable protocols.
- **protocol** (*EvaluatorRunnerMixin*) – Protocol to build computable with.

Returns

AlgorithmVQS – self.

property final_parameters: SymbolDict

Parameters used to evaluate final expectation value.

final_propagation_evaluation(*runner*)

Evaluates input computable expression for the last step of the propagation.

Parameters

- **runner** (*Callable[[SymbolDict], Any]*) – Computable expression to be evaluated.
- ****kwargs** – Keyword arguments to be passed to computable `run()` method.

Returns

ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]] – Evaluated computable expression for the last step of the propagation.

post_propagation_evaluation(*runner*, **args*, ***kwargs*)

Evaluates input computable expression along propagation trajectory.

This is done after propagation has been done.

Parameters

- **runner** (*Callable[[SymbolDict], Any]*) – 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 expressions 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_co, bound= generic, covariant=True)]] – The solution to the 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 \(2019\)](#).

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.

Notes

The same protocol is used in `build()` method to build the metric tensor, [MetricTensorReal](#), and the derivative, [ExpectationValueBraDerivativeReal](#).

build(protocol)

Executes the build method of a computable expression.

Parameters

- **backend** – Backend for the computable protocols.
- **protocol** ([EvaluatorRunnerMixin](#)) – Protocol to build computable with.

Returns

[AlgorithmVQS](#) – self.

property final_parameters: SymbolDict

Parameters used to evaluate final expectation value.

final_propagation_evaluation(runner)

Evaluates input computable expression for the last step of the propagation.

Parameters

- **runner** ([Callable\[\[SymbolDict\], Any\]](#)) – Computable expression to be evaluated.
- ****kwargs** – Keyword arguments to be passed to computable `run()` method.

Returns

`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – Evaluated computable expression for the last step of the propagation.

post_propagation_evaluation(runner, *args, **kwargs)

Evaluates input computable expression along propagation trajectory.

This is done after propagation has been done.

Parameters

- **runner** ([Callable\[\[SymbolDict\], Any\]](#)) – 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 expressions 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_co, bound= generic, covariant=True)]]` – The solution to the 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).

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.

 **Notes**

Note that the same protocol is used in `build()` method to build the metric tensor, `MetricTensorImag`, and the derivative, `ExpectationValueBraDerivativeImag`.

`build(protocol)`

Executes the build method of a computable expression.

Parameters

- `backend` – Backend for the computable protocols.
- `protocol` (`EvaluatorRunnerMixin`) – Protocol to build computable with.

Returns

`AlgorithmVQS` – self.

`property final_parameters: SymbolDict`

Parameters used to evaluate final expectation value.

`final_propagation_evaluation(runner)`

Evaluates input computable expression for the last step of the propagation.

Parameters

- `runner` (`Callable[[SymbolDict], Any]`) – Computable expression to be evaluated.
- `**kwargs` – Keyword arguments to be passed to computable `run()` method.

Returns

`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – Evaluated computable expression for the last step of the propagation.

post_propagation_evaluation(runner, *args, **kwargs)

Evaluates input computable expression along propagation trajectory.

This is done after propagation has been done.

Parameters

- **runner** (`Callable[[SymbolDict], Any]`) – 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 expressions 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_co, bound= generic, covariant=True)]]` – The solution to the ODE experiment.

Raises

`RuntimeError` – If member variable `_backend` is not initialised.

27.2 inquanto.ansatzes

27.2.1 Basic ansatzes

class GeneralAnsatz(reference, *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]`) – A reference state circuit or any valid initializer for `reference_circuit_builder()`.
- **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)`

default_pass()

Get the default compiler pass for the ansatz type.

Returns

`BasePass` – A tket pass object.

df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

 **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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

df_symbolic(symbol_map=None, *, space=None, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

 **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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

`abstract free_symbols()`

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

`free_symbols_ordered()`

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None)`

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

`get_circuit_no_ref(symbol_map=None, compiler_pass=None)`

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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** (`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.

property n_symbols: int
Returns the number of free symbols in the object.

reference_qubit_state()
Create a symbolic `QubitState` representation of the reference state.

Returns
`QubitState` – Reference state as a `QubitState`.

reset_reference(reference)
Resetting the reference state of the ansatz in place.

Note

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

reference (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit
Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet
Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

abstract symbol_substitution(symbol_map=None)
Performs an in-place symbol substition 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_CircuitAnsatz (`symbol_map=None, compiler_pass=None, ignore_default_pass=False`)

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- `symbol_map` (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol substitution map.
- `compiler_pass` (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation applied
- `ignore_default_pass` (`bool`, default: `False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

to_QubitState ()

Create a symbolic QubitState representation of the ansatz.

Returns

`QubitState` – Ansatz as a QubitState.

class CircuitAnsatz (`circuit, reference=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()`.

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)`

default_pass()

Get the default compiler pass for the ansatz type.

Returns

`BasePass` – A tket pass object.

df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

 **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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

df_symbolic(symbol_map=None, *, space=None, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

 **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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

`free_symbols()`

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

`free_symbols_ordered()`

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None)`

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

`get_circuit_no_ref(symbol_map=None, compiler_pass=None)`

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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** (`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.

property n_symbols: int
Returns the number of free symbols in the object.

reference_qubit_state()
Create a symbolic `QubitState` representation of the reference state.

Returns
`QubitState` – Reference state as a `QubitState`.

reset_reference(reference)
Resetting the reference state of the ansatz in place.

Note

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

reference (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit
Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet
Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

symbol_substitution(symbol_map=None)
Performs an in-place symbol substastion 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_CircuitAnsatz (`symbol_map=None`, `compiler_pass=None`, `ignore_default_pass=False`)

Cast the ansatz as `CircuitAnsatz` with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- `symbol_map` (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol substitution map.
- `compiler_pass` (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation applied
- `ignore_default_pass` (`bool`, default: `False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

to_QubitState ()

Create a symbolic `QubitState` representation of the ansatz.

Returns

`QubitState` – Ansatz as a `QubitState`.

class ComposedAnsatz (*`ansatzes`, `reference=None`)

Bases: `CircuitAnsatz`

Composes circuit ansatzes into one circuit ansatz.

Note

The composed circuit is built by applying the arguments of the `ComposedAnsatz` constructor in reverse order. i.e. for two instances of `CircuitAnsatz` A, B, the composed circuit of `ComposedAnsatz(A, B)` represents the state $|AB\rangle$, hence B is applied first.

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()`.

`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)`

`default_pass()`

Get the default compiler pass for the ansatz type.

Returns

`BasePass` – A tket pass object.

`df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)`

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

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. Passed to `get_numeric_representation()`.
- **space** (*Any*, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.

- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

df_symbolic (`symbol_map=None, *, space=None, tol=1e-10`)

Returns a `pandas.DataFrame` representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

 **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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

free_symbols()

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None`)

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

get_circuit_no_ref(`symbol_map=None, compiler_pass=None`)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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` (`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.

property n_symbols: int

Returns the number of free symbols in the object.

reference_qubit_state()

Create a symbolic `QubitState` representation of the reference state.

Returns

`QubitState` – Reference state as a `QubitState`.

reset_reference(`reference`)

Resetting the reference state of the ansatz in place.

 **Note**

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

`reference` (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet

Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

symbol_substitution (symbol_map=None)

Performs an in-place symbol substastion 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_CircuitAnsatz (symbol_map=None, compiler_pass=None, ignore_default_pass=False)

Cast the ansatz as `CircuitAnsatz` with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None`) – Optional symbol substitution map.
- **compiler_pass** (`Optional[BasePass], default: None`) – Optional compiler pass for circuit compilation applied
- **ignore_default_pass** (`bool, default: False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

to_QubitState()

Create a symbolic `QubitState` representation of the ansatz.

Returns

`QubitState` – Ansatz as a `QubitState`.

class TrotterAnsatz(exponents, reference=None)

Bases: `GeneralAnsatz`

Ansatz representing a state built from a product of Pauli-exponentials.

This is at the core of the UCC family of ansatzes in InQuanto.

Note: This class requires numerical operators within the input `QubitOperatorList`.

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()`.

i Examples

```
>>> from inquanto.states import QubitState
>>> exponents = QubitOperatorList.from_string("a [(1j, Y0 X2)], b [(1j, Y1 X3)]"
    <input>")
>>> ref = QubitState([1, 1, 0, 0])
>>> ansatz = TrotterAnsatz(exponents, reference=ref)
>>> ansatz.free_symbols_ordered() # returns an lexicographically ordered set_
    <input>of symbols
SymbolSet([a, b])
>>> ansatz.free_symbols() == {Symbol("a"), Symbol("b")} # returns a set of_
    <input>symbols
True
>>> ansatz.subs("new_{ }").free_symbols() == {Symbol("new_a"), Symbol("new_b")}
    <input># 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
```

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)
```

default_pass()

Get the default compiler pass for the ansatz type.

Returns

A tket pass object.

df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

 **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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

df_symbolic(symbol_map=None, *, space=None, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

 **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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

property exponents: `QubitOperatorList`

Returns the qubit operator exponents.

free_symbols()

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None`)

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

get_circuit_no_ref(`symbol_map=None, compiler_pass=None`)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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** (`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.

property n_symbols: int
Returns the number of free symbols in the object.

reference_qubit_state()
Create a symbolic `QubitState` representation of the reference state.

Returns
`QubitState` – Reference state as a `QubitState`.

reset_reference(reference)
Resetting the reference state of the ansatz in place.

Note

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

reference (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit
Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet
Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

symbol_substitution(symbol_map=None)
Performs an in-place symbol substastion 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_CircuitAnsatz (`symbol_map=None`, `compiler_pass=None`, `ignore_default_pass=False`)

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- `symbol_map` (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol substitution map.
- `compiler_pass` (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation applied
- `ignore_default_pass` (`bool`, default: `False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

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.

```
reference_circuit_builder(initializer)
```

Building a non-symbolic reference circuit.

Parameters

```
initializer(Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit, None]) –
```

A non-symbolic initializer circuit or an object that can be converted to one.

- If `initializer` is an `int`, an empty initializer circuit is created with `initializer` number of qubits.
- If `initializer` is a `list` of qubit-s, an empty initializer circuit is created with the qubits.
- **If `initializer` is a list of 0 or 1-s, an initializer circuit is created and x gate is added for indices where the `initializer[index] == 1`.**
- **If `initializer` is a `QubitSpace`, an empty initializer circuit is created with qubits in the `QubitSpace`.**
- **If `initializer` is a non-symbolic `QubitState`, an initializer circuit is created that represents the initializer state.**

Returns

```
Optional[Circuit] – A non-symbolic reference circuit.
```

27.2.2 Fermion Space ansatzes

```
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 initializes state trotterization.

Qubit tapering can optionally be performed. To enable qubit tapering, pass a `TapererZ2` object to `taperer`. Tapering behavior 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 is built, 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** (`TapererZ2`, default: `None`) – The taperer object used to control how the 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": Tests each exponent and throws an exception if any exponent 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": Tests each exponent and discards 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.

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)`

default_pass()

Get the default compiler pass for the ansatz type.

Returns

A ket pass object.

df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

**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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.

- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

df_symbolic (`symbol_map=None`, *, `space=None`, `tol=1e-10`)

Returns a `pandas.DataFrame` representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

⚠️ 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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

property exponents: QubitOperatorList

Returns the qubit operator exponents.

property fermion_operator_exponents: FermionOperatorList

Returns the list of exponents of the exponential product included in the ansatz.

free_symbols()

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None)

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

get_circuit_no_ref(symbol_map=None, compiler_pass=None)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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` (`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.

`property n_symbols: int`

Returns the number of free symbols in the object.

`reference_qubit_state()`

Create a symbolic `QubitState` representation of the reference state.

Returns

`QubitState` – Reference state as a `QubitState`.

`reset_reference(reference)`

Resetting the reference state of the ansatz in place.

Note

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

`reference` (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

`property state_circuit: Circuit`

Returns the symbolic state circuit with a default compilation.

`property state_symbols: SymbolSet`

Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

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_CircuitAnsatz (symbol_map=None, compiler_pass=None, ignore_default_pass=False)

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None`) – Optional symbol substitution map.
- **compiler_pass** (`Optional[BasePass], default: None`) – Optional compiler pass for circuit compilation applied
- **ignore_default_pass** (`bool, default: False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

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).

Builds ansatz for a given `fermion_space` and `fermion_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.

`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)`

`default_pass()`

Get the default compiler pass for the ansatz type.

Returns

A tket pass object.

`df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)`

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

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. Passed to `get_numeric_representation()`.
- `space` (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- `backend` (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- `tol` (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

`df_symbolic(symbol_map=None, *, space=None, tol=1e-10)`

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

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. Passed to `get_symbolic_representation()`.

- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

property exponents: `QubitOperatorList`

Returns the qubit operator exponents.

property fermion_operator_exponents: `FermionOperatorList`

Returns the list of exponents of the exponential product included in the ansatz.

free_symbols()

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None`)

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

get_circuit_no_ref (`symbol_map=None, compiler_pass=None`)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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` (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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` (`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.

property n_symbols: int

Returns the number of free symbols in the object.

reference_qubit_state()

Create a symbolic `QubitState` representation of the reference state.

Returns

`QubitState` – Reference state as a `QubitState`.

reset_reference(reference)

Resetting the reference state of the ansatz in place.

Note

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

`reference` (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet

Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

symbol_substitution(symbol_map=None)

Performs an in-place symbol substastion 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_CircuitAnsatz(symbol_map=None, compiler_pass=None, ignore_default_pass=False)`

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- `symbol_map` (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol substitution map.
- `compiler_pass` (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation applied
- `ignore_default_pass` (`bool`, default: `False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

`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).

Builds ansatz for a given `fermion_space` and `fermion_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.

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)`

default_pass()

Get the default compiler pass for the ansatz type.

Returns

A ket pass object.

df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

 **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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.

- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

df_symbolic (`symbol_map=None`, *, `space=None`, `tol=1e-10`)

Returns a `pandas.DataFrame` representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

⚠️ 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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

property exponents: QubitOperatorList

Returns the qubit operator exponents.

property fermion_operator_exponents: FermionOperatorList

Returns the list of exponents of the exponential product included in the ansatz.

free_symbols()

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None)

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

get_circuit_no_ref(symbol_map=None, compiler_pass=None)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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` (`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.

`property n_symbols: int`

Returns the number of free symbols in the object.

`reference_qubit_state()`

Create a symbolic `QubitState` representation of the reference state.

Returns

`QubitState` – Reference state as a `QubitState`.

`reset_reference(reference)`

Resetting the reference state of the ansatz in place.

Note

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

`reference` (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

`property state_circuit: Circuit`

Returns the symbolic state circuit with a default compilation.

`property state_symbols: SymbolSet`

Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

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_CircuitAnsatz (symbol_map=None, compiler_pass=None, ignore_default_pass=False)

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None`) – Optional symbol substitution map.
- **compiler_pass** (`Optional[BasePass], default: None`) – Optional compiler pass for circuit compilation applied
- **ignore_default_pass** (`bool, default: False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

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)

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. Assumes input exponents are ordered as single exponents first, followed by double exponents.
- **fermion_state** (*FermionState*) – Initial fermionic reference state.

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)

default_pass()

Get the default compiler pass for the ansatz type.

Returns

BasePass – A tket pass object.

df_numeric (symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses *get_numeric_representation()* to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

`df_symbolic(symbol_map=None, *, space=None, tol=1e-10)`

Returns a `pandas.DataFrame` representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

property fermion_operator_exponents: *FermionOperatorList*

Returns the list of exponents of the exponential product included in the ansatz.

free_symbols()

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order 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*, *compiler_pass=None*)

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

get_circuit_no_ref(*symbol_map=None*, *compiler_pass=None*)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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** (`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.

property n_symbols: int

Returns the number of free symbols in the object.

reference_qubit_state()

Create a symbolic `QubitState` representation of the reference state.

Returns

`QubitState` – Reference state as a `QubitState`.

reset_reference (*reference*)

Resetting the reference state of the ansatz in place.

Note

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

reference (Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet

Returns the ordered parameter symbols this state uses.

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]) – A mapping for substitution of free symbols.

Returns

TypeVar(SYMBOLICTYPE, bound= Symbolic) – A copy of self with symbols substituted according to the provided map.

symbol_substitution (*symbol_map=None*)

Performs an in-place symbol substastion 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_CircuitAnsatz (*symbol_map=None, compiler_pass=None, ignore_default_pass=False*)

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- `symbol_map` (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol substitution map.
- `compiler_pass` (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation applied
- `ignore_default_pass` (`bool`, default: `False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

`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).

Described in <https://doi.org/10.1063/5.0144680>.

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.

`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)`

default_pass()

Get the default compiler pass for the ansatz type.

Returns

BasePass – A tket pass object.

df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

 **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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

DataFrame – A dataframe representing the object.

df_symbolic(symbol_map=None, *, space=None, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

 **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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

property fermion_operator_exponents: FermionOperatorList

Returns the list of exponents of the exponential product included in the ansatz.

free_symbols()

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None)

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

get_circuit_no_ref(symbol_map=None, compiler_pass=None)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.

- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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** (`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.

property n_symbols: int

Returns the number of free symbols in the object.

reference_qubit_state()

Create a symbolic `QubitState` representation of the reference state.

Returns

`QubitState` – Reference state as a `QubitState`.

reset_reference(reference)

Resetting the reference state of the ansatz in place.

 **Note**

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

reference (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet

Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

symbol_substitution(symbol_map=None)

Performs an in-place symbol substastion 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_CircuitAnsatz (`symbol_map=None`, `compiler_pass=None`, `ignore_default_pass=False`)

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- `symbol_map` (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol substitution map.
- `compiler_pass` (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation applied
- `ignore_default_pass` (`bool`, default: `False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

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 generalized spin-paired doubles excitations, no singles (k-UpCCGD).

Here, generalized means occupied and virtual orbital subspaces are undistinguished, hence occupied-occupied and virtual-virtual excitations are included. 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.

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)`

default_pass()

Get the default compiler pass for the ansatz type.

Returns

A tket pass object.

df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

 **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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

`df_symbolic(symbol_map=None, *, space=None, tol=1e-10)`

Returns a `pandas.DataFrame` representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

 **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. Passed to `get_symbolic_representation()`.
- `space` (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- `tol` (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

`property exponents: QubitOperatorList`

Returns the qubit operator exponents.

`property fermion_operator_exponents: FermionOperatorList`

Returns the list of exponents of the exponential product included in the ansatz.

`free_symbols()`

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

`free_symbols_ordered()`

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None)

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

get_circuit_no_ref(symbol_map=None, compiler_pass=None)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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` (`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.

`property n_symbols: int`

Returns the number of free symbols in the object.

`reference_qubit_state()`

Create a symbolic `QubitState` representation of the reference state.

Returns

`QubitState` – Reference state as a `QubitState`.

`reset_reference(reference)`

Resetting the reference state of the ansatz in place.

Note

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

`reference` (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

`property state_circuit: Circuit`

Returns the symbolic state circuit with a default compilation.

`property state_symbols: SymbolSet`

Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

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_CircuitAnsatz (symbol_map=None, compiler_pass=None, ignore_default_pass=False)

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None`) – Optional symbol substitution map.
- **compiler_pass** (`Optional[BasePass], default: None`) – Optional compiler pass for circuit compilation applied
- **ignore_default_pass** (`bool, default: False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

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.

Ansatz consisting of `k` factors of variationally independent unitary coupled cluster operators with fully generalized singles and generalized spin-paired doubles excitations (k-UpCCGSD).

Here, generalized means occupied and virtual orbital subspaces are undistinguished, hence occupied-occupied and virtual-virtual excitations are included. 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.

`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)`

default_pass()

Get the default compiler pass for the ansatz type.

Returns

A tket pass object.

df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

 **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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

df_symbolic(symbol_map=None, *, space=None, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

 **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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

property exponents: QubitOperatorList

Returns the qubit operator exponents.

property fermion_operator_exponents: FermionOperatorList

Returns the list of exponents of the exponential product included in the ansatz.

free_symbols()

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None)

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

get_circuit_no_ref(symbol_map=None, compiler_pass=None)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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** (`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.

property n_symbols: int
Returns the number of free symbols in the object.

reference_qubit_state()
Create a symbolic `QubitState` representation of the reference state.

Returns
`QubitState` – Reference state as a `QubitState`.

reset_reference(reference)
Resetting the reference state of the ansatz in place.

Note

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

reference (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit
Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet
Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

symbol_substitution(symbol_map=None)
Performs an in-place symbol substastion 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_CircuitAnsatz (`symbol_map=None`, `compiler_pass=None`, `ignore_default_pass=False`)

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- `symbol_map` (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol substitution map.
- `compiler_pass` (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation applied
- `ignore_default_pass` (`bool`, default: `False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

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 FermionSpaceAnsatzkUpCCGSDSsinglet (fermion_space, fermion_state, k_input,
                                             qubit_mapping=QubitMappingJordanWigner(), *args,
                                             **kwargs)
```

Bases: *FermionSpaceStateExp*

k-UpCCGSDSinglet ansatz.

Ansatz consisting of k factors of variationally independent unitary coupled cluster operators with fully generalized singles and generalized spin paired doubles excitations.

Here, generalized means occupied and virtual orbital subspaces are undistinguished, hence occupied-occupied and virtual-virtual excitations are included. See <https://arxiv.org/abs/1810.02327> for more details.

Adapted to singlets, so 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.
- **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.

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)`

default_pass()

Get the default compiler pass for the ansatz type.

Returns

A tket pass object.

df_numeric (`symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10`)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

`df_symbolic(symbol_map=None, *, space=None, tol=1e-10)`

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

`property exponents: QubitOperatorList`

Returns the qubit operator exponents.

property fermion_operator_exponents: FermionOperatorList

Returns the list of exponents of the exponential product included in the ansatz.

free_symbols()

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None)

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

get_circuit_no_ref(symbol_map=None, compiler_pass=None)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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** (`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.

property n_symbols: int

Returns the number of free symbols in the object.

reference_qubit_state()

Create a symbolic `QubitState` representation of the reference state.

Returns

`QubitState` – Reference state as a `QubitState`.

reset_reference(`reference`)

Resetting the reference state of the ansatz in place.

Note

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

reference (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet

Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

symbol_substitution (`symbol_map=None`)

Performs an in-place symbol substition 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_CircuitAnsatz (`symbol_map=None, compiler_pass=None, ignore_default_pass=False`)

Cast the ansatz as `CircuitAnsatz` with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol substitution map.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation applied
- **ignore_default_pass** (`bool`, default: `False`) – Prevents the ansatz self.default_pass being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

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`

Unitary coupled cluster with fully generalized doubles excitations, no singles (UCCGD).

Here, generalized means occupied and virtual orbital subspaces are undistinguished, hence occupied-occupied and virtual-virtual excitations are included. 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.

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)`

default_pass()

Get the default compiler pass for the ansatz type.

Returns

A tket pass object.

df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

 **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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

df_symbolic(symbol_map=None, *, space=None, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

property exponents: `QubitOperatorList`

Returns the qubit operator exponents.

property fermion_operator_exponents: `FermionOperatorList`

Returns the list of exponents of the exponential product included in the ansatz.

free_symbols()

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None`)

Constructs a single 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`) – Optional symbol mapping for substitution.

- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

get_circuit_no_ref (`symbol_map=None, compiler_pass=None`)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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** (`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.

property n_symbols: int

Returns the number of free symbols in the object.

reference_qubit_state()

Create a symbolic `QubitState` representation of the reference state.

Returns

`QubitState` – Reference state as a `QubitState`.

reset_reference(reference)

Resetting the reference state of the ansatz in place.

 **Note**

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

reference (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet

Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

symbol_substitution(symbol_map=None)

Performs an in-place symbol substitution 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_CircuitAnsatz(symbol_map=None, compiler_pass=None, ignore_default_pass=False)

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol substitution map.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation applied
- **ignore_default_pass** (`bool`, default: `False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

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`

Unitary coupled cluster with fully generalized singles and doubles excitations (UCCGSD).

Here, generalized means occupied and virtual orbital subspaces are undistinguished, hence occupied-occupied and virtual-virtual excitations are included. 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.

`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)`

`default_pass()`

Get the default compiler pass for the ansatz type.

Returns

A tket pass object.

```
df_numeric (symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)
```

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

`df_symbolic(symbol_map=None, *, space=None, tol=1e-10)`

Returns a `pandas.DataFrame` representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

property exponents: *QubitOperatorList*

Returns the qubit operator exponents.

property fermion_operator_exponents: *FermionOperatorList*

Returns the list of exponents of the exponential product included in the ansatz.

free_symbols()

Returns the free symbols in the object.

Returns

Set[Symbol] – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None)

Constructs a single 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*) – Optional symbol mapping for substitution.
- **compiler_pass** (*Optional[BasePass]*, default: *None*) – Optional compiler pass for circuit compilation.

Returns

Circuit – A circuit that represent the state.

get_circuit_no_ref(symbol_map=None, compiler_pass=None)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (*Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]*, default: *None*) – Optional symbol mapping for substitution.
- **compiler_pass** (*BasePass*, default: *None*) – Optional compiler pass for circuit compilation.

Returns

Circuit – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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** (`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.

property n_symbols: int

Returns the number of free symbols in the object.

reference_qubit_state()

Create a symbolic `QubitState` representation of the reference state.

Returns

`QubitState` – Reference state as a `QubitState`.

reset_reference (*reference*)

Resetting the reference state of the ansatz in place.

Note

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

reference (Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet

Returns the ordered parameter symbols this state uses.

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]) – A mapping for substitution of free symbols.

Returns

TypeVar(SYMBOLICTYPE, bound= Symbolic) – A copy of self with symbols substituted according to the provided map.

symbol_substitution (*symbol_map=None*)

Performs an in-place symbol substastion 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_CircuitAnsatz (*symbol_map=None, compiler_pass=None, ignore_default_pass=False*)

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- `symbol_map` (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol substitution map.
- `compiler_pass` (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation applied
- `ignore_default_pass` (`bool`, default: `False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

`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).

Builds ansatz for a given `fermion_space` and `fermion_state`.

Adapted to singlets, so 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.

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)`

default_pass()

Get the default compiler pass for the ansatz type.

Returns

A ket pass object.

df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

 **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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.

- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

df_symbolic (`symbol_map=None`, *, `space=None`, `tol=1e-10`)

Returns a `pandas.DataFrame` representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

⚠️ 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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

property exponents: QubitOperatorList

Returns the qubit operator exponents.

property fermion_operator_exponents: FermionOperatorList

Returns the list of exponents of the exponential product included in the ansatz.

free_symbols()

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None)

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

get_circuit_no_ref(symbol_map=None, compiler_pass=None)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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` (`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.

`property n_symbols: int`

Returns the number of free symbols in the object.

`reference_qubit_state()`

Create a symbolic `QubitState` representation of the reference state.

Returns

`QubitState` – Reference state as a `QubitState`.

`reset_reference(reference)`

Resetting the reference state of the ansatz in place.

Note

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

`reference` (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

`property state_circuit: Circuit`

Returns the symbolic state circuit with a default compilation.

`property state_symbols: SymbolSet`

Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

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_CircuitAnsatz (symbol_map=None, compiler_pass=None, ignore_default_pass=False)

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol substitution map.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation applied
- **ignore_default_pass** (`bool`, default: `False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

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.

27.2.3 Multi-configurational ansatzes

```
class MultiConfigurationAnsatz(configurations, reference=None)
```

Bases: `GeneralAnsatz`

Variational ansatz consisting of selected occupation configurations, using Givens rotations.

Consists of occupation configurations which have symbolic coefficients.

Follows logic of <https://quantum-journal.org/papers/q-2022-06-20-742/>, with in-house developed procedures to control Givens rotations.

Finds necessary controls of Givens rotation boxes using method based on hamming distances (and their qubit-wise decompositions) of all configurations relative to the first.

For single excitations (2 qubits), hamming distance = 2, the 1-body Givens rotation (G1) in Figure 12 of <https://quantum-journal.org/papers/q-2022-06-20-742/> is used.

For double excitations (4 qubits), hamming distance = 4, the 2-body Givens rotation (G2) in Figure 11 of <https://quantum-journal.org/papers/q-2022-06-20-742/> is used.

Excitations that are higher than doubles are achieved by nesting a controlled G1 inside a ladder of controlled-SWAPs (G1(>2)).

 **Note**

Supports only basis configurations which have the same number of 1s in the bit string. This corresponds to configurations with the same particle number as Jordan-Wigner encoding is assumed.

Parameters

- **configurations** (`List[QubitStateString]`) – List of `QubitStateString` objects providing the basis configurations 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()`.

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)`

default_pass()

Get the default compiler pass for the ansatz type.

Returns

A tket pass object.

df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

 **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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

df_symbolic(symbol_map=None, *, space=None, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

`free_symbols()`

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

`free_symbols_ordered()`

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None)`

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

get_circuit_no_ref(symbol_map=None, compiler_pass=None)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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** (`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.

property n_symbols: int

Returns the number of free symbols in the object.

reference_qubit_state()

Create a symbolic `QubitState` representation of the reference state.

Returns

`QubitState` – Reference state as a `QubitState`.

reset_reference(reference)

Resetting the reference state of the ansatz in place.

 **Note**

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

reference (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet

Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

symbol_substitution(symbol_map=None)

Performs an in-place symbol substition 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

`MultiConfigurationAnsatz` – self, with symbols substituted.

`to_CircuitAnsatz(symbol_map=None, compiler_pass=None, ignore_default_pass=False)`

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- `symbol_map` (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol substitution map.
- `compiler_pass` (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation applied
- `ignore_default_pass` (`bool`, default: `False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

`to_QubitState()`

Create a symbolic `QubitState` representation of the ansatz.

Returns

`QubitState` – Ansatz as a `QubitState`.

`class MultiConfigurationState(input_states)`

Bases: `GeneralAnsatz`

State preparation for a selected linear combination of occupation configurations with fixed coefficients.

Configuration coefficients must be real-valued.

Follows logic of <https://quantum-journal.org/papers/q-2022-06-20-742/>, with in-house developed procedures to control Givens rotations.

Finds necessary controls of Givens rotation boxes using method based on hamming distances (and their qubit-wise decompositions) of all configurations relative to the first.

For single excitations (2 qubits), hamming distance = 2, the 1-body Givens rotation (G1) in Figure 12 of <https://quantum-journal.org/papers/q-2022-06-20-742/> is used.

For double excitations (4 qubits), hamming distance = 4, the 2-body Givens rotation (G2) in Figure 11 of <https://quantum-journal.org/papers/q-2022-06-20-742/> is used.

Excitations that are higher than doubles are achieved by nesting a controlled G1 inside a ladder of controlled-SWAPs (G1(>2)).

Note

Supports only basis configurations which have the same number of 1s in the bit string. This corresponds to configurations with the same particle number as Jordan-Wigner encoding is assumed.

Parameters

`input_states` (*QubitState*) – Multi-reference qubit state input, providing basis configurations and numerical coefficients. Coefficients must be real and normalised.

`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)`

`default_pass()`

Get the default compiler pass for the ansatz type.

Returns

A tket pass object.

`df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)`

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

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. Passed to `get_numeric_representation()`.
- `space` (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.

- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

df_symbolic (`symbol_map=None, *, space=None, tol=1e-10`)

Returns a `pandas.DataFrame` representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

 **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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

free_symbols()

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None`)

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[Transform]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

get_circuit_no_ref(`symbol_map=None, compiler_pass=None`)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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` (`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.

property n_symbols: int

Returns the number of free symbols in the object.

reference_qubit_state()

Create a symbolic `QubitState` representation of the reference state.

Returns

`QubitState` – Reference state as a `QubitState`.

reset_reference(`reference`)

Resetting the reference state of the ansatz in place.

 **Note**

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

`reference` (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet

Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

symbol_substitution (symbol_map=None)

Performs an in-place symbol substition 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_CircuitAnsatz (symbol_map=None, compiler_pass=None, ignore_default_pass=False)

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None`) – Optional symbol substitution map.
- **compiler_pass** (`Optional[BasePass], default: None`) – Optional compiler pass for circuit compilation applied
- **ignore_default_pass** (`bool, default: False`) – Prevents the ansatz `self.default_pass` being applied

Returns

CircuitAnsatz – A new `CircuitAnsatz` ansatz.

`to_QubitState()`

Create a symbolic `QubitState` representation of the ansatz.

Returns

`QubitState` – Ansatz as a `QubitState`.

`class MultiConfigurationStateBox(input_states)`

Bases: `GeneralAnsatz`

State preparation for a selected linear combination of occupation configurations with fixed coefficients.

Serves as a wrapper for pytket's `StatePreparationBox`.

Converts input `QubitState` to ndarray state vector (exponentially scaling), passes this to `StatePreparationBox` which prepares the state vector using multiplexed-RY and multiplexed-Rz gates.

See <https://cqcl.github.io/tket/pytket/api/circuit.html#pytket.circuit.StatePreparationBox>.

Parameters

`input_states` (`QubitState`) – Multi-reference `QubitState` input, providing basis configurations and numerical coefficients.

`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)`

`default_pass()`

Get the default compiler pass for the ansatz type.

Returns

A tket pass object.

`df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)`

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

 **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. Passed to `get_numeric_representation()`.

- `space` (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- `backend` (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- `tol` (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

`df_symbolic(symbol_map=None, *, space=None, tol=1e-10)`

Returns a `pandas.DataFrame` representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

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. Passed to `get_symbolic_representation()`.
- `space` (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- `tol` (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

`free_symbols()`

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None)

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[Transform]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

Circuit – A circuit that represent the state.

get_circuit_no_ref(symbol_map=None, compiler_pass=None)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

Circuit – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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** (`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.

property n_symbols: int

Returns the number of free symbols in the object.

reference_qubit_state()

Create a symbolic `QubitState` representation of the reference state.

Returns

`QubitState` – Reference state as a `QubitState`.

reset_reference(`reference`)

Resetting the reference state of the ansatz in place.

 **Note**

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

- **reference** (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet

Returns the ordered parameter symbols this state uses.

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])` – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

symbol_substitution(symbol_map=None)

Performs an in-place symbol substition 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_CircuitAnsatz(symbol_map=None, compiler_pass=None, ignore_default_pass=False)

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None`) – Optional symbol substitution map.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation applied

- `ignore_default_pass` (`bool`, default: `False`) – Prevents the ansatz self.default_pass being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

to_QubitState()

Create a symbolic `QubitState` representation of the ansatz.

Returns

`QubitState` – Ansatz as a `QubitState`.

27.2.4 Rotational ansatzes

class RealGeneralizedBasisRotationAnsatz(reference)

Bases: `BaseRealBasisRotationAnsatz`

Basis rotation ansatz that allows for generalized real unitary rotations.

Constructs the ansatz $\hat{R} |\Psi\rangle$ where \hat{R} is given by the Thouless theorem: $\hat{R} = \exp \left[\sum_{ij} [\ln U]_{ij} a_i^\dagger a_j \right]$ and U is a single-particle basis rotation matrix (real, unitary). See <https://arxiv.org/abs/1711.04789>.

Note

Supports generalized rotations which may mix spin channels.

Note

Uses a Jordan-Wigner mapping for encoding rotations.

Parameters

`reference` (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Initial reference state, $|\Psi\rangle$ above. Can be any valid initializer for `reference_circuit_builder()`.

ansatz_parameters_from_unitary(rotation_unitary)

Converts a real unitary rotation to ansatz parameters.

Performs a QR decomposition of the rotation matrix to find ansatz parameters.

Parameters

`rotation_unitary` (`ndarray[Any, dtype[float]]`) – A real unitary matrix with shape $(N_{\text{so}}, N_{\text{so}})$ where N_{so} is the number of spin-orbitals/qubits. Assumes the spin-orbital encoding: [0a, 0b, 1a, 1b, 2a...].

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)`

default_pass()

Get the default compiler pass for the ansatz type.

Returns

`DecomposeBoxes` – A tket pass object.

df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

 **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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

df_symbolic(symbol_map=None, *, space=None, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

 **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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

`free_symbols()`

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

`free_symbols_ordered()`

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None)`

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

`get_circuit_no_ref(symbol_map=None, compiler_pass=None)`

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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** (`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.

property n_symbols: int
Returns the number of free symbols in the object.

reference_qubit_state()
Create a symbolic `QubitState` representation of the reference state.

Returns
`QubitState` – Reference state as a `QubitState`.

reset_reference(reference)
Resetting the reference state of the ansatz in place.

Note

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

reference (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet

Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

symbol_substitution(symbol_map=None)

Performs an in-place symbol substastion 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

`BaseRealBasisRotationAnsatz` – self, with symbols substituted.

to_CircuitAnsatz (`symbol_map=None`, `compiler_pass=None`, `ignore_default_pass=False`)

Cast the ansatz as `CircuitAnsatz` with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- `symbol_map` (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol substitution map.
- `compiler_pass` (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation applied
- `ignore_default_pass` (`bool`, default: `False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

to_QubitState ()

Create a symbolic `QubitState` representation of the ansatz.

Returns

`QubitState` – Ansatz as a `QubitState`.

class RealRestrictedBasisRotationAnsatz (reference)

Bases: `BaseRealBasisRotationAnsatz`

Basis rotation ansatz that allows for restricted-spin, real unitary rotations.

Constructs the ansatz $\hat{R}|\Psi\rangle$ where \hat{R} is given by the Thouless theorem: $\hat{R} = \exp\left[\sum_{ij}[\ln U]_{ij}a_i^\dagger a_j\right]$ and U is a single-particle basis rotation matrix (real, unitary). See <https://arxiv.org/abs/1711.04789>.

Note

Supports rotations which act on both spin channels equivalently, and independently.

Note

Uses a Jordan-Wigner mapping for encoding rotations.

Parameters

reference (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`)
– Initial reference state, $|\Psi\rangle$ above.

ansatz_parameters_from_unitary (*rotation_unitary*)

Converts a real unitary rotation to ansatz parameters.

Performs a QR decomposition of the rotation matrix to find ansatz parameters.

Parameters

rotation_unitary (`ndarray[Any, dtype[float]]`) – A real unitary matrix with shape $(n_{\text{orb}}, n_{\text{orb}})$ where n_{orb} is the of number of spatial orbitals. This rotation is applied to both spin channels.

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)`

default_pass()

Get the default compiler pass for the ansatz type.

Returns

`DecomposeBoxes` – A tket pass object.

df_numeric (*symbol_map=None*, *, *space=None*, *backend=None*, *dtype=complex*, *tol=1e-10*)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

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.
Passed to `get_numeric_representation()`.

- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

df_symbolic (`symbol_map=None, *, space=None, tol=1e-10`)

Returns a `pandas.DataFrame` representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.



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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

free_symbols()

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None`)

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

get_circuit_no_ref(`symbol_map=None, compiler_pass=None`)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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` (`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.

property n_symbols: int

Returns the number of free symbols in the object.

reference_qubit_state()

Create a symbolic `QubitState` representation of the reference state.

Returns

`QubitState` – Reference state as a `QubitState`.

reset_reference(`reference`)

Resetting the reference state of the ansatz in place.

 **Note**

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

`reference` (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet

Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

symbol_substitution(symbol_map=None)

Performs an in-place symbol substastion 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

`BaseRealBasisRotationAnsatz` – self, with symbols substituted.

to_CircuitAnsatz(symbol_map=None, compiler_pass=None, ignore_default_pass=False)

Cast the ansatz as `CircuitAnsatz` with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None`) – Optional symbol substitution map.
- **compiler_pass** (`Optional[BasePass]`, `default: None`) – Optional compiler pass for circuit compilation applied
- **ignore_default_pass** (`bool`, `default: False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

to_QubitState()

Create a symbolic `QubitState` representation of the ansatz.

Returns

`QubitState` – Ansatz as a `QubitState`.

class RealUnrestrictedBasisRotationAnsatz(reference)

Bases: `BaseRealBasisRotationAnsatz`

Basis rotation ansatz that allows for unrestricted-spin, real unitary rotations.

Constructs the ansatz $\hat{R}|\Psi\rangle$ where \hat{R} is given by the Thouless theorem: $\hat{R} = \exp\left[\sum_{ij}[\ln U]_{ij}a_i^\dagger a_j\right]$ and U is a single-particle basis rotation matrix (real, unitary). See <https://arxiv.org/abs/1711.04789>.

 **Note**

Supports rotations which transform each spin channel independently.

 **Note**

Uses a Jordan-Wigner mapping for encoding rotations.

Parameters

`reference` (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`)
– Initial reference state, $|\Psi\rangle$ above.

ansatz_parameters_from_unitary(rotation_unitary_a, rotation_unitary_b)

Converts a real unitary rotation to ansatz parameters.

Performs a QR decomposition of the rotation matrix to find ansatz parameters.

Parameters

- `rotation_unitary_a` (`ndarray[Any, dtype[float]]`) – A real unitary matrix with shape $(n_{\text{orb}}, n_{\text{orb}})$ where n_{orb} is the number of spatial orbitals. This rotation is applied to the alpha spin channel.
- `rotation_unitary_b` (`ndarray[Any, dtype[float]]`) – A real unitary matrix with shape $(n_{\text{orb}}, n_{\text{orb}})$ where n_{orb} is the number of spatial orbitals. This rotation is applied to the beta spin channel. Must be the same shape as `rotation_unitary_a`.

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)
```

default_pass()

Get the default compiler pass for the ansatz type.

Returns

DecomposeBoxes – A tket pass object.

df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

 **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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

df_symbolic(symbol_map=None, *, space=None, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

 **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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

`free_symbols()`

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

`free_symbols_ordered()`

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None)`

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

`get_circuit_no_ref(symbol_map=None, compiler_pass=None)`

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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** (`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.

property n_symbols: int
Returns the number of free symbols in the object.

reference_qubit_state()
Create a symbolic `QubitState` representation of the reference state.

Returns
`QubitState` – Reference state as a `QubitState`.

reset_reference(reference)
Resetting the reference state of the ansatz in place.

Note

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

reference (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit

Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet

Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

symbol_substitution(symbol_map=None)

Performs an in-place symbol substastion 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

`BaseRealBasisRotationAnsatz` – self, with symbols substituted.

to_CircuitAnsatz (`symbol_map=None`, `compiler_pass=None`, `ignore_default_pass=False`)

Cast the ansatz as `CircuitAnsatz` with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- `symbol_map` (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol substitution map.
- `compiler_pass` (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation applied
- `ignore_default_pass` (`bool`, default: `False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

to_QubitState ()

Create a symbolic `QubitState` representation of the ansatz.

Returns

`QubitState` – Ansatz as a `QubitState`.

generalized_basis_rotation_to_circuit (`rotation_unitary`)

Converts a real unitary basis rotation to a circuit.

Supports generalized real rotations, which may couple and mix spin channels.

Note

Uses a Jordan-Wigner mapping for encoding rotations.

Parameters

- `rotation_unitary` (`ndarray[Any, dtype[float]]`) – A real unitary matrix with shape $(N_{\{so\}}, N_{\{so\}})$ where $N_{\{so\}}$ is the number of spin-orbitals/qubits. Assumes the spin-orbital encoding: [0a, 0b, 1a, 1b, 2a...].

Returns

A circuit representing the basis rotation matrix, the number of qubits equals to the number of spin-orbitals.

restricted_basis_rotation_to_circuit(*rotation_unitary*)

Converts a real unitary basis rotation to a circuit.

Rotation is applied to both alpha and beta spin channels.

 **Note**

Uses a Jordan-Wigner mapping for encoding rotations.

Parameters

rotation_unitary (ndarray[*Any*, dtype[*float*]]) – A real unitary matrix with shape (*n_{orb}*, *n_{orb}*) where *n_{orb}* is the of number of spatial orbitals.

Returns

Circuit – A circuit representing the basis rotation matrix, the number of qubits equals to the number of spin-orbitals.

unrestricted_basis_rotation_to_circuit(*rotation_unitary_a*, *rotation_unitary_b*)

Converts a pair of real unitary basis rotations to a circuit.

Each rotation matrix corresponds to a spin channel.

 **Note**

Uses a Jordan-Wigner mapping for encoding rotations.

Parameters

- **rotation_unitary_a** (ndarray[*Any*, dtype[*float*]]) – A real unitary matrix with shape (*n_{orb}*, *n_{orb}*) where *n_{orb}* is the of number of spatial orbitals. This rotation is applied to the alpha spin channel.
- **rotation_unitary_b** (ndarray[*Any*, dtype[*float*]]) – A real unitary matrix with shape (*n_{orb}*, *n_{orb}*) where *n_{orb}* is the of number of spatial orbitals. This rotation is applied to the alpha spin channel. Must be the same shape as *rotation_unitary_a*.

Returns

A circuit representing the basis rotation matrix, the number of qubits equals to the number of spin-orbitals.

rotate_ansatz_restricted(*ansatz*, *rotation_unitary*)

Rotate an ansatz by a real unitary basis rotation.

Rotation is applied to both alpha and beta spin channels. Uses *restricted_basis_rotation_to_circuit()* in combination with *CircuitAnsatz* and *ComposedAnsatz* to build a new state.

 **Note**

Uses a Jordan-Wigner mapping for encoding rotations.

Parameters

- **ansatz** (*GeneralAnsatz*) – Input ansatz to be rotated. Must be encoded with the Jordan-Wigner mapping.
- **rotation_unitary** (*ndarray[Any, dtype[float]]*) – A real unitary matrix with shape $(n_{\text{orb}}, n_{\text{orb}})$ where n_{orb} is the number of spatial orbitals.

Returns

A new ansatz with the rotation appended to the state preparation circuit.

27.2.5 Other ansatzes

```
class HamiltonianVariationalAnsatz(fermion_state, hamiltonian_operator=FermionOperator(),
                                    qubit_mapping=QubitMappingJordanWigner(), s=1, *args, **kwargs)
```

Bases: *TrotterAnsatz*

Hamiltonian Variational Ansatz introduced in <https://arxiv.org/abs/1507.08969>.

Parameters

- **fermion_state** (*FermionState*) – Spin orbital occupations. If applied to molecules, this should represent a single configuration mean field state.
- **hamiltonian_operator** (*FermionOperator*, default: *FermionOperator()*) – The hamiltonian operator produced by driver.
- **qubit_mapping** (*QubitMapping*, default: *QubitMappingJordanWigner()*) – How to map Fock state operators and states to qubit operators and circuits.
- **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.

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)

default_pass()

Get the default compiler pass for the ansatz type.

Returns

A tket pass object.

df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)

Returns a `pandas.DataFrame` representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

`df_symbolic(symbol_map=None, *, space=None, tol=1e-10)`

Returns a `pandas.DataFrame` representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

property `exponents: QubitOperatorList`

Returns the qubit operator exponents.

free_symbols()

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None)

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

get_circuit_no_ref(symbol_map=None, compiler_pass=None)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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** (`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.

property n_symbols: int

Returns the number of free symbols in the object.

reference_qubit_state()

Create a symbolic `QubitState` representation of the reference state.

Returns

`QubitState` – Reference state as a `QubitState`.

reset_reference(`reference`)

Resetting the reference state of the ansatz in place.

Note

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

reference (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

`property split_hamiltonian: Tuple[FermionOperator]`

Split hamiltonian into diagonal and non-diagonal elements.

Hamiltonian is separated into three terms H_{diag} , H_{hop} , and H_{ex} , each one a `FermionOperator`. These will be exponentiated to build the Hamiltonian Variational Ansatz.

Diagonal terms:

$$H_{diag} = \sum_p (e_p a_p^\dagger a_p) + \sum_{p,q} (h_{pq} a_p^\dagger a_p a_q^\dagger a_q)$$

in terms of `FermionOperator`:

$$((p, 1), (p, 0)) : e_p \text{ and } ((p, 1), (q, 1), (p, 0), (q, 0)) : h_{pq}$$

Hopping terms:

$$H_{hop} = \sum_{p,q} (e_{p,q} a_p^\dagger a_q) + \sum_{p,r,q} (h_{pr} a_p^\dagger a_q a_r^\dagger a_r)$$

in terms of `FermionOperator`:

$$((p, 1), (q, 0)) : e_p, q \text{ and } ((p, 1), (q, 1), (q, 0), (r, 0)) : h_{pq}$$

Exchange terms:

$$H_{ex} = \sum_{p,q,r,s} (h_{pqrs} a_p^\dagger a_q^\dagger a_r a_s)$$

in terms of `FermionOperator`:

$$(((p, 1), (q, 1), (r, 0), (s, 0))) : h_{pqrs}$$

Returns

Diagonal, hopping, and exchange operators extracted from the Hamiltonian.

`property state_circuit: Circuit`

Returns the symbolic state circuit with a default compilation.

`property state_symbols: SymbolSet`

Returns the ordered parameter symbols this state uses.

`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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

`symbol_substitution(symbol_map=None)`

Performs an in-place symbol substitution 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_CircuitAnsatz(symbol_map=None, compiler_pass=None, ignore_default_pass=False)`

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- `symbol_map (Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None], default: None)` – Optional symbol substitution map.
- `compiler_pass (Optional[BasePass], default: None)` – Optional compiler pass for circuit compilation applied
- `ignore_default_pass (bool, default: False)` – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

`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)
```

Bases: `GeneralAnsatz`

Parent class for ansatzes with layered replicas of a group of circuit primitives (such as HEA).

Hierarchy:

- Block (`CustomGateDef`): Consists of a collection of parametric `pytket.circuit.OpType` operations added to a circuit.
- Layer (`CircBox`): Consists of a rotation block and (possibly) an entangler block. If `inter_block_entangler=True` (default), rotation blocks between layers are separated by a block of two-qubit operations (entangler).

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])` – A reference state circuit or any valid initializer for `reference_circuit_builder()`.
- `n_layers (int, default: 1)` – Number of layers to be added to circuit.
- `inter_block_entangler (bool, default: True)` – If True add a block of two-qubit gate primitives between rotation blocks within a layer.
- `entangler_def (CustomGateDef, default: None)` – Definition of the two-qubit entangler primitive.
- `cap_layers (bool, default: True)` – If True end circuit by adding one extra block of rotations.
- `cap_def (List[CustomGateDef], default: None)` – A list of circuit primitives as `CustomGateDefs` for the cap.

`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)`

default_pass()

Get the default compiler pass for the ansatz type.

Returns

A tket pass object.

df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

 **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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

df_symbolic(symbol_map=None, *, space=None, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

 **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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

free_symbols()

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None)

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

get_circuit_no_ref(symbol_map=None, compiler_pass=None)

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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** (`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.

property n_symbols: int
    Returns the number of free symbols in the object.

reference_qubit_state()
    Create a symbolic QubitState representation of the reference state.

Returns
    QubitState – Reference state as a QubitState.

reset_reference(reference)
    Resetting the reference state of the ansatz in place.
```

ⓘ Note

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

reference (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

>Returns

Returns self with the modified reference.

```
property state_circuit: Circuit
    Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet
    Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

>Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

```
symbol_substitution(symbol_map=None)
    Performs an in-place symbol substition 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_CircuitAnsatz (*symbol_map=None*, *compiler_pass=None*, *ignore_default_pass=False*)

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

 **Note**

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the *CircuitAnsatz* using *ignore_default_pass*. This can be combined with the user defined *compiler_pass* for full control. If the result of *self.default_pass* is not ignored and a *compiler_pass* is defined then both sets of passes will be combined and applied.

Parameters

- **symbol_map** (*Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]*, default: *None*) – Optional symbol substitution map.
- **compiler_pass** (*Optional[BasePass]*, default: *None*) – Optional compiler pass for circuit compilation applied
- **ignore_default_pass** (*bool*, default: *False*) – Prevents the ansatz *self.default_pass* being applied

Returns

CircuitAnsatz – A new *CircuitAnsatz* ansatz.

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*)

Bases: *LayeredAnsatz*

Hardware Efficient Ansatz as defined in <https://arxiv.org/abs/1704.05018>.

Consists of *n_layers* layers of circuit primitives. Each layer consists of a rotation block and an entangler block.

Parameters

- **rotation_operations** (*List[OpType]*) – Define blocks of rotations as Ry, RxRy or RxRyRz.
- **reference** (*Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]*) – Initial state of qubit register. A reference state circuit or any valid initializer for *reference_circuit_builder()*.
- **n_layers** (*int*) – Number of layers.
- **entangler_operation** (*OpType*, default: *OpType.CX*) – The operation to build the entangler block. Default value is a CX gate.

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)`**default_pass()**

Get the default compiler pass for the ansatz type.

Returns

A tket pass object.

df_numeric(symbol_map=None, *, space=None, backend=None, dtype=complex, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_numeric_representation()` to generate a numeric vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

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. Passed to `get_numeric_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_numeric_representation()`.
- **backend** (`Optional[Backend]`, default: `None`) – An optional backend to use to build the representation. Passed to `get_numeric_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. Passed to `get_numeric_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

df_symbolic(symbol_map=None, *, space=None, tol=1e-10)

Returns a pandas.DataFrame representation of the ansatz state.

Uses `get_symbolic_representation()` to generate a symbolic vector for the ansatz state, and returns a dataframe with coefficients alongside their corresponding computational basis states.

Assumes lexicographical ordering of basis states.

Symbolic coefficients are simplified before being added to dataframe.

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. Passed to `get_symbolic_representation()`.
- **space** (`Any`, default: `None`) – Basis information to represent the object. Passed to `get_symbolic_representation()`.
- **tol** (`float`, default: `1e-10`) – Absolute tolerance below which terms in the ansatz will be omitted from the dataframe. If negative, no terms are discarded.

Returns

`DataFrame` – A dataframe representing the object.

`free_symbols()`

Returns the free symbols in the object.

Returns

`Set[Symbol]` – Unordered set of symbols.

`free_symbols_ordered()`

Returns the free symbols in increasing lexicographic order 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, compiler_pass=None)`

Constructs a single 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`) – Optional symbol mapping for substitution.
- **compiler_pass** (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the state.

`get_circuit_no_ref(symbol_map=None, compiler_pass=None)`

Constructs a single state circuit without the reference state.

Parameters

- **symbol_map** (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol mapping for substitution.
- **compiler_pass** (`BasePass`, default: `None`) – Optional compiler pass for circuit compilation.

Returns

`Circuit` – A circuit that represent the referenceless 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** (`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[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], 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** (`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.

property n_symbols: int
Returns the number of free symbols in the object.

reference_qubit_state()
Create a symbolic `QubitState` representation of the reference state.

Returns
`QubitState` – Reference state as a `QubitState`.

reset_reference(reference)
Resetting the reference state of the ansatz in place.

Note

The number of qubits in the new reference has to match with the already existing reference state.

Parameters

reference (`Union[int, List[Qubit], List[int], QubitSpace, QubitState, Circuit]`) – Any reference that can be converted into a non-symbolic reference state circuit.

Returns

Returns self with the modified reference.

property state_circuit: Circuit
Returns the symbolic state circuit with a default compilation.

property state_symbols: SymbolSet
Returns the ordered parameter symbols this state uses.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

symbol_substitution(symbol_map=None)
Performs an in-place symbol substastion 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_CircuitAnsatz (`symbol_map=None`, `compiler_pass=None`, `ignore_default_pass=False`)

Cast the ansatz as CircuitAnsatz with optional symbol substitution and compilation control.

Note

Some ansatzes have built in tket compiler passes. These can be ignored when casting to the `CircuitAnsatz` using `ignore_default_pass`. This can be combined with the user defined `compiler_pass` for full control. If the result of `self.default_pass` is not ignored and a `compiler_pass` is defined then both sets of passes will be combined and applied.

Parameters

- `symbol_map` (`Union[SymbolDict, Dict[Symbol, Expr], Dict[Symbol, float], Dict[Symbol, Union[float, complex, Expr]], Callable[[Symbol], Expr], str, None]`, default: `None`) – Optional symbol substitution map.
- `compiler_pass` (`Optional[BasePass]`, default: `None`) – Optional compiler pass for circuit compilation applied
- `ignore_default_pass` (`bool`, default: `False`) – Prevents the ansatz `self.default_pass` being applied

Returns

`CircuitAnsatz` – A new `CircuitAnsatz` ansatz.

to_QubitState ()

Create a symbolic `QubitState` representation of the ansatz.

Returns

`QubitState` – Ansatz as a `QubitState`.

27.3 inquanto.computables

27.3.1 inquanto.computables.atomic

Submodule for quantum computable expressions that interact directly with InQuanto Protocols.

class ExpectationValue (`state, kernel`)

Bases: `ComputableNode[float]`

Represents the expectation value of a Hermitian operator kernel with a state.

$\langle \Psi | H | \Psi \rangle$

Parameters

- `state` (`GeneralAnsatz`) – Input state.
- `kernel` (`QubitOperator`) – Hermitian operator kernel.

add_label(*label*, *label_children=False*)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with *label*.

Parameters

- **label** (`str`) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with *label*. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

children()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

default_evaluate(*parameters*, *protocol=None*)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

evaluate(*evaluator=None*)

Evaluate this object using the provided evaluator function.

Parameters

evaluator (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.

Returns

`Union[TypeVar(EvaluatableType, bound= Evaluatable), TypeVar(EvaluatedType)]` – The computed result.

free_symbols()

Returns free symbols in the state $|\Psi\rangle$.

Return type

`Set[Symbol]`

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

kernel: *QubitOperator***label: `str` = `None`****print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

state: *GeneralAnsatz***walk(*depth*=0)**

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

- `depth` (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class ExpectationValueBraDerivativeImag(*state*, *kernel*, *symbols*)

Bases: `IExpectationValueDerivative`

Represents the imaginary part of the bra derivatives of an expectation value of a Hermitian operator.

$\Im\langle\partial_\theta\Psi(\theta)|H|\Psi(\theta)\rangle$

Parameters

- `state` (`GeneralAnsatz`) – Ansatz state $|\Psi(\theta)\rangle$.
- `kernel` (`QubitOperator`) – Qubit operator kernel H .
- `symbols` (`Set[Symbol]`) – Symbols with respect to which the derivatives are computed.

add_label(*label*, *label_children*=`False`)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label` (`str`) – Label string to be assigned to node. Overwrites any existing label.
- `label_children` (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

children()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

default_evaluate (*parameters*, *protocol*=*None*)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

evaluate (*evaluator*=*None*)

Evaluate this object using the provided evaluator function.

Parameters

evaluator (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.

Returns

`Union[TypeVar(EvaluatableType, bound= Evaluatable), TypeVar(EvaluatedType)]` – The computed result.

free_symbols ()

Returns free symbols in the ansatz state.

Return type

`Set[Symbol]`

free_symbols_ordered ()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf ()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

kernel: `QubitOperator`

label: `str` = `None`

print_tree ()

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

state: `GeneralAnsatz`

symbols: `Set[Symbol]`

walk (depth=0)
Generator method to traverse the computable expression tree in a depth-first manner.

Parameters
`depth (int, default: 0)` – The initial depth of the tree. Default is 0.

Yields
A tuple containing the current computable node and its depth in the tree.

Return type
`Iterator[Tuple[ComputableNode, int]]`

class ExpectationValueBraDerivativeReal (state, kernel, symbols)
Bases: `IExpectationValueDerivative`
Represents the real part of the bra derivatives of an expectation value of a Hermitian operator.
 $\Re\langle\partial_\theta\Psi(\theta)|H|\Psi(\theta)\rangle$

Parameters

- `state (GeneralAnsatz)` – Ansatz state $|\Psi(\theta)\rangle$.
- `kernel (QubitOperator)` – Qubit operator kernel H .
- `symbols (Set[Symbol])` – Symbols with respect to which the derivatives are computed.

add_label (label, label_children=False)
Assign a label to the current computable.
Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label (str)` – Label string to be assigned to node. Overwrites any existing label.
- `label_children (bool, default: False)` – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns
`ComputableNode` – Self.

children ()
Generator method that yields the child computable nodes of the current computable node.

Yields
An iterator over the child computable nodes of the current computable node.

Return type
`Iterator[ComputableNode]`

default_evaluate (parameters, protocol=None)
Evaluate the final results immediately for return.
If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- `parameters (SymbolDict)` – `SymbolDict` or dict to map symbols to values.
- `protocol (Optional[Any], default: None)` – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluatable object.

evaluate (evaluator=None)

Evaluate this object using the provided evaluator function.

Parameters

`evaluator` (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.

Returns

`Union[TypeVar(EvaluatableType, bound=Evaluatable), TypeVar(EvaluatedType)]` – The computed result.

free_symbols()

Returns free symbols in the ansatz state.

Return type

`Set[Symbol]`

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

kernel: QubitOperator**label: str = None****print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

state: GeneralAnsatz**symbols: Set[Symbol]****walk (depth=0)**

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

`depth` (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

```
class ExpectationValueDerivative(state, kernel, symbols)
```

Bases: IExpectationValueDerivative

Represents the derivatives of the expectation value of a Hermitian operator.

$$\partial_\theta \langle \Psi(\theta) | H | \Psi(\theta) \rangle$$

Parameters

- **state** (*GeneralAnsatz*) – Ansatz state $|\Psi(\theta)\rangle$.
- **kernel** (*QubitOperator*) – Qubit operator kernel H .
- **symbols** (*Set[Symbol]*) – Symbols with respect to which the derivatives are computed.

```
add_label(label, label_children=False)
```

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with *label*.

Parameters

- **label** (*str*) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (*bool*, default: `False`) – If `True`, all child nodes of this computable are labeled with *label*. If `False`, children remain unlabeled.

Returns

ComputableNode – Self.

```
children()
```

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

Iterator[ComputableNode]

```
default_evaluate(parameters, protocol=None)
```

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- **parameters** (*SymbolDict*) – *SymbolDict* or dict to map symbols to values.
- **protocol** (*Optional[Any]*, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

Union[Evaluatable, Any] – Final value of the evaluable object.

```
evaluate(evaluator=None)
```

Evaluate this object using the provided evaluator function.

Parameters

evaluator (*Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]*, default: `None`) – A callable evaluator that is called on the instance.

Returns

Union[TypeVar(EvaluatableType, bound= Evaluatable), TypeVar(EvaluatedType)] – The computed result.

free_symbols()
Returns free symbols in the ansatz state.

Return type
`Set[Symbol]`

free_symbols_ordered()
Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns
`SymbolSet` – Ordered free symbols in object.

is_leaf()
Check if the current computable node is a leaf (i.e., it has no children).

Returns
`bool` – True if the computable node is a leaf, `False` otherwise.

kernel: `QubitOperator`

label: `str = None`

print_tree()
Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type
`None`

state: `GeneralAnsatz`

symbols: `Set[Symbol]`

walk (`depth=0`)
Generator method to traverse the computable expression tree in a depth-first manner.

Parameters
`depth` (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields
A tuple containing the current computable node and its depth in the tree.

Return type
`Iterator[Tuple[ComputableNode, int]]`

class ExpectationValueKetDerivativeImag(state, kernel, symbols)
Bases: `IExpectationValueDerivative`

Represents the imaginary part of the ket derivatives of an expectation value of a Hermitian operator.
 $\Im\langle\Psi(\theta)|H|\partial_\theta\Psi(\theta)\rangle$

Parameters

- **state** (`GeneralAnsatz`) – Ansatz state $|\Psi(\theta)\rangle$.
- **kernel** (`QubitOperator`) – Qubit operator kernel H .
- **symbols** (`Set[Symbol]`) – Symbols with respect to which the derivatives are computed.

add_label (`label, label_children=False`)
Assign a label to the current computable.
Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- **label** (`str`) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

children()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

default_evaluate(parameters, protocol=None)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

evaluate(evaluator=None)

Evaluate this object using the provided evaluator function.

Parameters

evaluator (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.

Returns

`Union[TypeVar(EvaluatableType, bound= Evaluatable), TypeVar(EvaluatedType)]` – The computed result.

free_symbols()

Returns free symbols in the ansatz state.

Return type

`Set[Symbol]`

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

```
kernel: QubitOperator
```

```
label: str = None
```

```
print_tree()
```

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

```
None
```

```
state: GeneralAnsatz
```

```
symbols: Set[Symbol]
```

```
walk(depth=0)
```

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

- **depth** (*int*, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

```
Iterator[Tuple[ComputableNode, int]]
```

```
class ExpectationValueKetDerivativeReal(state, kernel, symbols)
```

Bases: IExpectationValueDerivative

Represents the real part of the ket derivatives of an expectation value of a Hermitian operator.

```
 $\Re \langle \Psi(\theta) | H | \partial_\theta \Psi(\theta) \rangle$ 
```

Parameters

- **state** (*GeneralAnsatz*) – Ansatz state $|\Psi(\theta)\rangle$.
- **kernel** (*QubitOperator*) – Qubit operator kernel H .
- **symbols** (*Set[Symbol]*) – Symbols with respect to which the derivatives are computed.

```
add_label(label, label_children=False)
```

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with *label*.

Parameters

- **label** (*str*) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (*bool*, default: False) – If True, all child nodes of this computable are labeled with *label*. If False, children remain unlabeled.

Returns

```
ComputableNode – Self.
```

```
children()
```

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

```
Iterator[ComputableNode]
```

default_evaluate(parameters, protocol=None)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from [pytket-extensions](#). First, it will try the `AerStateBackend` from [pytket-qiskit](#), and then the `QulacsBackend` from [pytket-qulacs](#).

Parameters

- **parameters** (*SymbolDict*) – *SymbolDict* or dict to map symbols to values.
- **protocol** (*Optional[Any]*, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

evaluate(evaluator=None)

Evaluate this object using the provided evaluator function.

Parameters

evaluator (*Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]*, default: `None`) – A callable evaluator that is called on the instance.

Returns

`Union[TypeVar(EvaluatableType, bound= Evaluatable), TypeVar(EvaluatedType)]` – The computed result.

free_symbols()

Returns free symbols in the ansatz state.

Return type

`Set[Symbol]`

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

kernel: *QubitOperator***label**: `str` = `None`**print_tree**()

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

state: *GeneralAnsatz***symbols**: `Set[Symbol]`

walk (*depth=0*)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

- **depth** (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class ExpectationValueNonHermitian(state, kernel)

Bases: `ComputableNode[complex]`

Represents the expectation value of a non-Hermitian operator kernel with a state.

$\langle \Psi | H | \Psi \rangle$

Parameters

- **state** (`GeneralAnsatz`) – Input state.
- **kernel** (`QubitOperator`) – Operator kernel.

add_label (*label, label_children=False*)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- **label** (`str`) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (`bool`, default: False) – If True, all child nodes of this computable are labeled with label. If False, children remain unlabeled.

Returns

`ComputableNode` – Self.

children()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

default_evaluate (*parameters, protocol=None*)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: None) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

evaluate (*evaluator=None*)

Evaluate this object using the provided evaluator function.

Parameters

evaluator (`Optional[Callable[[Evaluable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.

Returns

`Union[TypeVar(EvaluatableType, bound= Evaluatable), TypeVar(EvaluatedType)]` – The computed result.

free_symbols()

Returns free symbols in the state $|\Psi\rangle$.

Return type

`Set[Symbol]`

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

kernel: QubitOperator**label: str = None****print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

state: GeneralAnsatz**walk(depth=0)**

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

depth (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class MetricTensorImag(state, symbols)

Bases: `IMetricTensor`

Represents the imaginary part of the metric tensor.

Calculates: $\Im\langle \partial_{\theta_i} \Psi(\theta) | \partial_{\theta_j} \Psi(\theta) \rangle$ for all i, j .

Parameters

- **state** (`GeneralAnsatz`) – Ansatz state $|\Psi(\theta)\rangle$.

- **symbols** (`Set[Symbol]`) – Symbols with respect to which the derivatives are computed.

add_label (`label, label_children=False`)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- **label** (`str`) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

children()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

default_evaluate (`parameters, protocol=None`)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

evaluate (`evaluator=None`)

Evaluate this object using the provided evaluator function.

Parameters

evaluator (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.

Returns

`Union[TypeVar(EvaluatableType, bound=Evaluatable), TypeVar(EvaluatedType)]` – The computed result.

free_symbols()

Returns free symbols in the state.

Return type

`Set[Symbol]`

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

label: str = None**print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

state: GeneralAnsatz**symbols: Set[Symbol]****walk(depth=0)**

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

- `depth (int, default: 0)` – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class MetricTensorReal(state, symbols)

Bases: IMetricTensor

Represents the real part of the metric tensor.

Calculates: $\Re\langle \partial_{\theta_i} \Psi(\theta) | \partial_{\theta_j} \Psi(\theta) \rangle$ for all i, j .

Parameters

- `state (GeneralAnsatz)` – Ansatz state $|\Psi(\theta)\rangle$.
- `symbols (Set[Symbol])` – Symbols with respect to which the derivatives are computed.

add_label(label, label_children=False)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label (str)` – Label string to be assigned to node. Overwrites any existing label.
- `label_children (bool, default: False)` – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

children()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type`Iterator[ComputableNode]`**default_evaluate**(parameters, protocol=None)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.**Parameters**

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns`Union[Evaluatable, Any]` – Final value of the evaluatable object.**evaluate**(evaluator=None)

Evaluate this object using the provided evaluator function.

Parameters`evaluator` (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.**Returns**`Union[TypeVar(EvaluatableType, bound=Evaluatable), TypeVar(EvaluatedType)]` – The computed result.**free_symbols**()

Returns free symbols in the state.

Return type`Set[Symbol]`**free_symbols_ordered**()Returns the free symbols in increasing lexicographic order as `SymbolSet`.**Returns**`SymbolSet` – Ordered free symbols in object.**is_leaf**()

Check if the current computable node is a leaf (i.e., it has no children).

Returns`bool` – True if the computable node is a leaf, `False` otherwise.**label: str = None****print_tree**()

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type`None`**state: GeneralAnsatz****symbols: Set[Symbol]**

walk (*depth=0*)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

- **depth** (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type
`Iterator[Tuple[ComputableNode, int]]`
class Overlap (*bra_state*, *ket_state*, *kernel=<factory>*)

Bases: `IOOverlap`

Represents the overlap of two states with a Hermitian kernel operator.

 $\langle \Phi | H | \Psi \rangle$
Parameters

- **bra_state** (`GeneralAnsatz`) – Bra state $|\Phi\rangle$.
- **ket_state** (`GeneralAnsatz`) – Ket state $|\Psi\rangle$.
- **kernel** (`Union[QubitOperator, QubitOperatorString]`, default: `<factory>`) – Qubit operator kernel H .

add_label (*label*, *label_children=False*)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- **label** (`str`) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns
`ComputableNode` – Self.
bra_state: `GeneralAnsatz`**children** ()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type
`Iterator[ComputableNode]`
default_evaluate (*parameters*, *protocol=None*)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.

- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

evaluate (evaluator=None)

Evaluate this object using the provided evaluator function.

Parameters

evaluator (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.

Returns

`Union[TypeVar(EvaluatableType, bound=Evaluatable), TypeVar(EvaluatedType)]` – The computed result.

free_symbols ()

Returns free symbols in both bra and ket states.

Return type

`Set[Symbol]`

free_symbols_ordered ()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf ()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

kernel: Union[QubitOperator, QubitOperatorString]**ket_state: GeneralAnsatz****label: str = None****print_tree ()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk (depth=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

depth (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

```
class OverlapImag(bra_state, ket_state, kernel=<factory>)
```

Bases: IOverlap

Represents the imaginary part of the overlap of two states with a Hermitian kernel operator.

$\Im\langle\Phi|H|\Psi\rangle$

Parameters

- **bra_state** (*GeneralAnsatz*) – Bra state $|\Phi\rangle$.
- **ket_state** (*GeneralAnsatz*) – Ket state $|\Psi\rangle$.
- **kernel** (*Union[QubitOperator, QubitOperatorString]*, default: `<factory>`) – Qubit operator kernel H .

add_label (*label, label_children=False*)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- **label** (*str*) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (*bool*, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

ComputableNode – Self.

bra_state: *GeneralAnsatz*

children()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

Iterator[ComputableNode]

default_evaluate (*parameters, protocol=None*)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- **parameters** (*SymbolDict*) – `SymbolDict` or dict to map symbols to values.
- **protocol** (*Optional[Any]*, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

Union[Evaluatable, Any] – Final value of the evaluable object.

evaluate (*evaluator=None*)

Evaluate this object using the provided evaluator function.

Parameters

- **evaluator** (*Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]*, default: `None`) – A callable evaluator that is called on the instance.

Returns

`Union[TypeVar(EvaluatableType, bound= Evaluatable), TypeVar(EvaluatedType)]` –
The computed result.

free_symbols()

Returns free symbols in both bra and ket states.

Return type

`Set[Symbol]`

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

kernel: Union[QubitOperator, QubitOperatorString]**ket_state: GeneralAnsatz****label: str = None****print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk(depth=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

`depth (int, default: 0)` – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class OverlapReal(bra_state, ket_state, kernel=<factory>)

Bases: `IOOverlap`

Represents the real part of the overlap of two states with a Hermitian kernel operator.

$\Re\langle\Phi|H|\Psi\rangle$

Parameters

- `bra_state (GeneralAnsatz)` – Bra state $|\Phi\rangle$.
- `ket_state (GeneralAnsatz)` – Ket state $|\Psi\rangle$.
- `kernel (Union[QubitOperator, QubitOperatorString], default: <factory>)` – Qubit operator kernel H .

add_label (*label*, *label_children=False*)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with *label*.

Parameters

- **label** (`str`) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with *label*. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

bra_state: `GeneralAnsatz`

children ()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

default_evaluate (*parameters*, *protocol=None*)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluatable object.

evaluate (*evaluator=None*)

Evaluate this object using the provided evaluator function.

Parameters

evaluator (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.

Returns

`Union[TypeVar(EvaluatableType, bound= Evaluatable), TypeVar(EvaluatedType)]` – The computed result.

free_symbols ()

Returns free symbols in both bra and ket states.

Return type

`Set[Symbol]`

free_symbols_ordered ()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

kernel: Union[QubitOperator, QubitOperatorString]**ket_state: GeneralAnsatz****label: str = None****print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk(depth=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

`depth (int, default: 0)` – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class OverlapSquared(bra_state, ket_state, kernel=<factory>)

Bases: `IOOverlap`

Represents the overlap squared of two states with a kernel operator.

$$|\langle \Phi | P | \Psi \rangle|^2$$

 **Note**

The kernel operator must be a single Pauli string.

Parameters

- `bra_state (GeneralAnsatz)` – Bra state $|\Phi\rangle$.
- `ket_state (GeneralAnsatz)` – Ket state $|\Psi\rangle$.
- `kernel (Union[QubitOperator, QubitOperatorString], default: <factory>)` – Qubit operator kernel H .

add_label(label, label_children=False)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label (str)` – Label string to be assigned to node. Overwrites any existing label.

- **label_children** (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns`ComputableNode` – Self.**bra_state**: `GeneralAnsatz`**children**()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type`Iterator[ComputableNode]`**default_evaluate**(*parameters*, *protocol*=`None`)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.**Parameters**

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns`Union[Evaluatable, Any]` – Final value of the evaluable object.**evaluate**(*evaluator*=`None`)

Evaluate this object using the provided evaluator function.

Parameters`evaluator` (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.**Returns**`Union[TypeVar(EvaluatableType, bound= Evaluatable), TypeVar(EvaluatedType)]` – The computed result.**free_symbols**()

Returns free symbols in both bra and ket states.

Return type`Set[Symbol]`**free_symbols_ordered**()Returns the free symbols in increasing lexicographic order as `SymbolSet`.**Returns**`SymbolSet` – Ordered free symbols in object.**is_leaf**()

Check if the current computable node is a leaf (i.e., it has no children).

Returns`bool` – True if the computable node is a leaf, `False` otherwise.

```
kernel: Union[QubitOperator, QubitOperatorString]
```

```
ket_state: GeneralAnsatz
```

```
label: str = None
```

```
print_tree()
```

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

None

```
walk(depth=0)
```

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

depth (int, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

Iterator[Tuple[ComputableNode, int]]

27.3.2 inquanto.computables.primitive

Primitive objects for constructing quantum computable expressions.

```
class ComputableFunction(func, *args)
```

Bases: ComputableNode[EvaluatedType]

Class representing a function applied to computable nodes in the expression tree.

Parameters

- **func** (Callable[..., TypeVar(EvaluatedType)]) – Callable expression to be evaluated.
- **args** (Any) – Computable nodes passed to callable `func`.

Example

```
>>> from inquanto.computables.primitive import ComputableInt
>>> add = ComputableFunction(lambda x, y: x + y, ComputableInt(3),_
>>> ComputableInt(4))
>>> result = add.evaluate()
>>> result
7
```

```
add_label(label, label_children=False)
```

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- **label** (str) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (bool, default: False) – If True, all child nodes of this computable are labeled with `label`. If False, children remain unlabeled.

Returns`ComputableNode` – Self.**args:** `Tuple[Any, ...]`**children()**

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type`Iterator[ComputableNode]`**default_evaluate**(*parameters*, *protocol=None*)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.**Parameters**

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns`Union[Evaluatable, Any]` – Final value of the evaluable object.**evaluate**(*evaluator=None*)

Recursively evaluates the expression tree and returns the computed result.

Parameters`evaluator` (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – Evaluator function passed to the `evaluate()` methods of the child computable nodes recursively.**Returns**`Union[ComputableFunction, TypeVar(EvaluatedType)]` – The computed result of the expression tree.**free_symbols()**

Returns the union of free symbols from all children.

Returns`Set[Symbol]` – A set containing the free symbols from all children.**free_symbols_ordered()**Returns the free symbols in increasing lexicographic order as `SymbolSet`.**Returns**`SymbolSet` – Ordered free symbols in object.**func:** `Callable[..., TypeVar(EvaluatedType)]`**is_leaf()**

Check if the current computable node is a leaf (i.e., it has no children).

Returns`bool` – True if the computable node is a leaf, `False` otherwise.

```
label: str = None
```

```
print_tree()
```

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

```
walk(depth=0)
```

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

`depth (int, default: 0)` – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

```
class ComputableInt(value)
```

Bases: `ComputableNode[int]`

Computable wrapper class for an int, mainly for demonstration purposes.

Parameters

`value (int)` – Integer value.

```
add_label(label, label_children=False)
```

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label (str)` – Label string to be assigned to node. Overwrites any existing label.
- `label_children (bool, default: False)` – If True, all child nodes of this computable are labeled with label. If False, children remain unlabeled.

Returns

`ComputableNode` – Self.

```
children()
```

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

```
default_evaluate(parameters, protocol=None)
```

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- `parameters (SymbolDict)` – `SymbolDict` or dict to map symbols to values.
- `protocol (Optional[Any], default: None)` – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluatable object.

evaluate (evaluator=None)

Evaluates its value.

Parameters

`evaluator (Optional[Callable[[Evaluatable], Any]], default: None)`

Return type

`int`

free_symbols ()

Returns the union of free symbols from all children.

Returns

`Set[Symbol]` – A set containing the free symbols from all children.

free_symbols_ordered ()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf ()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, False otherwise.

label: str = None

print_tree ()

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

value: int

walk (depth=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

`depth (int, default: 0)` – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class ComputableList (iterable=(), /)

Bases: `list, ComputableNode[List]`

Class representing a list of items of any types in the computable expression tree.

Parameters

`iterable` – An iterable which produces elements to initialize the list.

Example

```
>>> from inquanto.computables.primitive import ComputableInt
>>> k_list = ComputableList([ComputableInt(1), ComputableInt(2), 3, "foo"])
>>> k_list
[ComputableInt(value=1), ComputableInt(value=2), 3, 'foo']
>>> len(k_list)
4
>>> k_list.evaluate()
[1, 2, 3, 'foo']
```

`add_label(label, label_children=False)`

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label (str)` – Label string to be assigned to node. Overwrites any existing label.
- `label_children (bool, default: False)` – If True, all child nodes of this computable are labeled with `label`. If False, children remain unlabeled.

Returns

`ComputableNode` – Self.

`append(object, /)`

Append object to the end of the list.

`children()`

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

`clear()`

Remove all items from list.

`copy()`

Return a shallow copy of the list.

`count(value, /)`

Return number of occurrences of value.

`default_evaluate(parameters, protocol=None)`

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- `parameters (SymbolDict)` – `SymbolDict` or dict to map symbols to values.
- `protocol (Optional[Any], default: None)` – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluatable object.

evaluate (evaluator=None)

Evaluates each item in the list and returns the computed results as a list.

If an item is a computable, its `evaluate()` method is called. Otherwise, the item itself is returned.

Parameters

`evaluator` (`Optional[Callable[[Evaluatable], Any]]`, default: `None`) – Callable passed to each item's `evaluate()` method.

Returns

`Union[ComputableList, List]` – The computed results of the items.

extend (iterable, /)

Extend list by appending elements from the iterable.

free_symbols ()

Returns the union of free symbols from all children.

Returns

`Set[Symbol]` – A set containing the free symbols from all children.

free_symbols_ordered ()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

index (value, start=0, stop=9223372036854775807, /)

Return first index of value.

Raises `ValueError` if the value is not present.

insert (index, object, /)

Insert object before index.

is_leaf ()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

label: str = None**pop (index=-1, /)**

Remove and return item at index (default last).

Raises `IndexError` if list is empty or index is out of range.

print_tree ()

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

remove (value, /)

Remove first occurrence of value.

Raises `ValueError` if the value is not present.

reverse()

Reverse *IN PLACE*.

sort(*, key=None, reverse=False)

Sort the list in ascending order and return None.

The sort is in-place (i.e. the list itself is modified) and stable (i.e. the order of two equal elements is maintained).

If a key function is given, apply it once to each list item and sort them, ascending or descending, according to their function values.

The reverse flag can be set to sort in descending order.

walk(depth=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

depth (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

```
class ComputableNDArray(array_like: _SupportsArray[dtype[Any]] |  
    _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes  
    | _NestedSequence[bool | int | float | complex | str | bytes], *args: Any, **kwargs: Any)
```

Bases: `ndarray, ComputableNode[ndarray[Any, dtype[_ScalarType_co]]]`

Class representing a multi-dimensional array of items in a computable expression tree.

This class dresses numpy ndarrays with Computable methods. It can contain computable items as objects, allowing computations to be deferred. It provides an interface to seamlessly integrate within a computational framework that uses Computable-based structures.

The array can be of any dimension, and its constructor signature is the same as a numpy array with the `dtype` set to `object` if a computable element is used. Supports all mathematical operations.

Parameters

- **array_like** – Initial data for the array. It can be a list, an already instantiated numpy array, or a list of Computable objects.
- ***args** – Arguments passed to the numpy array constructor.
- ****kwargs** – Key word arguments passed to the numpy array constructor, for example `dtype=object`

 **Example**

```
>>> from inquanto.computables.primitive import ComputableInt
>>> qc_array = ComputableNDArray([ComputableInt(1), ComputableInt(2),  
    ↪ComputableInt(3), ComputableInt(4)], dtype=object)
>>> qc_array.evaluate()
array([1, 2, 3, 4], dtype=object)
```

```
>>> another_array = ComputableNDArray([ComputableInt(0), 1, ComputableInt(1),  
->ComputableInt(0)])  
>>> result = (qc_array + another_array)  
>>> result.evaluate()  
array([1, 3, 4, 4], dtype=object)
```

Note

The behavior of mathematical operations and functions on this class is determined by the behavior of the underlying numpy arrays. Computable items like `ComputableInt` will be evaluated when the `evaluate()` method is called.

T

View of the transposed array.

Same as `self.transpose()`.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])  
>>> a  
array([[1, 2],  
       [3, 4]])  
>>> a.T  
array([[1, 3],  
       [2, 4]])
```



```
>>> a = np.array([1, 2, 3, 4])  
>>> a  
array([1, 2, 3, 4])  
>>> a.T  
array([1, 2, 3, 4])
```

See also

`transpose`

`add_label` (*label*, *label_children=False*)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label` (`str`) – Label string to be assigned to node. Overwrites any existing label.
- `label_children` (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

ComputableNode – Self.

all (*axis=None, out=None, keepdims=False, *, where=True*)

Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

↳ See also

numpy.all

equivalent function

any (*axis=None, out=None, keepdims=False, *, where=True*)

Returns True if any of the elements of *a* evaluate to True.

Refer to *numpy.any* for full documentation.

↳ See also

numpy.any

equivalent function

argmax (*axis=None, out=None, *, keepdims=False*)

Return indices of the maximum values along the given axis.

Refer to *numpy.argmax* for full documentation.

↳ See also

numpy.argmax

equivalent function

argmin (*axis=None, out=None, *, keepdims=False*)

Return indices of the minimum values along the given axis.

Refer to *numpy.argmin* for detailed documentation.

↳ See also

numpy.argmin

equivalent function

argpartition (*kth, axis=-1, kind='introselect', order=None*)

Returns the indices that would partition this array.

Refer to *numpy.argpartition* for full documentation.

Added in version 1.8.0.

See also

`numpy.argpartition`
equivalent function

`argsort (axis=-1, kind=None, order=None)`

Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

See also

`numpy.argsort`
equivalent function

`astype (dtype, order='K', casting='unsafe', subok=True, copy=True)`

Copy of the array, cast to a specified type.

Parameters

- `dtype (str or dtype)` – Typecode or data-type to which the array is cast.
- `order ({'C', 'F', 'A', 'K'}, optional)` – Controls the memory layout order of the result. ‘C’ means C order, ‘F’ means Fortran order, ‘A’ means ‘F’ order if all the arrays are Fortran contiguous, ‘C’ order otherwise, and ‘K’ means as close to the order the array elements appear in memory as possible. Default is ‘K’.
- `casting ({'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional)` – Controls what kind of data casting may occur. Defaults to ‘unsafe’ for backwards compatibility.
 - ‘no’ means the data types should not be cast at all.
 - ‘equiv’ means only byte-order changes are allowed.
 - ‘safe’ means only casts which can preserve values are allowed.
 - ‘same_kind’ means only safe casts or casts within a kind, like float64 to float32, are allowed.
 - ‘unsafe’ means any data conversions may be done.
- `subok (bool, optional)` – If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.
- `copy (bool, optional)` – By default, astype always returns a newly allocated array. If this is set to false, and the `dtype`, `order`, and `subok` requirements are satisfied, the input array is returned instead of a copy.

Returns

`arr_t (ndarray)` – Unless `copy` is False and the other conditions for returning the input array are satisfied (see description for `copy` input parameter), `arr_t` is a new array of the same shape as the input array, with `dtype`, `order` given by `dtype`, `order`.

Notes

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for “unsafe” casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in ‘safe’ casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

Raises

ComplexWarning – When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1., 2., 2.5])

>>> x.astype(int)
array([1, 2, 2])
```

base

Base object if memory is from some other object.

Examples

The base of an array that owns its memory is `None`:

```
>>> x = np.array([1, 2, 3, 4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with `x`:

```
>>> y = x[2:]
>>> y.base is x
True
```

byteswap(*inplace=False*)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place. Arrays of byte-strings are not swapped. The real and imaginary parts of a complex number are swapped individually.

Parameters

`inplace (bool, optional)` – If `True`, swap bytes in-place, default is `False`.

Returns

`out (ndarray)` – The byteswapped array. If `inplace` is `True`, this is a view to self.

Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> list(map(hex, A))
['0x1', '0x100', '0x2233']
>>> A.byteswap(inplace=True)
array([ 256,       1, 13090], dtype=int16)
>>> list(map(hex, A))
['0x100', '0x1', '0x3322']

Arrays of byte-strings are not swapped
>>> A = np.array([b'ceg', b'fac'])
>>> A.byteswap()
array([b'ceg', b'fac'], dtype='|S3')
```

**A.newbyteorder().byteswap() produces an array with the same values
but different representation in memory**

```
>>> A = np.array([1, 2, 3])
>>> A.view(np.uint8)
array([1, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0,
       0, 0], dtype=uint8)
>>> A.newbyteorder().byteswap(inplace=True)
array([1, 2, 3])
>>> A.view(np.uint8)
array([0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0,
       0, 3], dtype=uint8)
```

children()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

choose(choices, out=None, mode='raise')

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

See also

`numpy.choose`
equivalent function

clip(min=None, max=None, out=None, **kwargs)

Return an array whose values are limited to [min, max]. One of max or min must be given.

Refer to `numpy.clip` for full documentation.

See also

numpy.clip
equivalent function

compress (*condition*, *axis=None*, *out=None*)

Return selected slices of this array along given axis.

Refer to *numpy.compress* for full documentation.

↳ See also

numpy.compress
equivalent function

conj()

Complex-conjugate all elements.

Refer to *numpy.conjugate* for full documentation.

↳ See also

numpy.conjugate
equivalent function

conjugate()

Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

↳ See also

numpy.conjugate
equivalent function

copy (*order='C'*)

Return a copy of the array.

Parameters

order ({'C', 'F', 'A', 'K'}, optional) – Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and *numpy.copy()* are very similar but have different default values for their *order=* arguments, and this function always passes sub-classes through.)

↳ See also

numpy.copy
Similar function with different default behavior

```
numpy.copyto
```

❶ Notes

This function is the preferred method for creating an array copy. The function `numpy.copy()` is similar, but it defaults to using order ‘K’, and will not pass sub-classes through by default.

❶ Examples

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]], order='F')

>>> y = x.copy()

>>> x.fill(0)

>>> x
array([[0, 0, 0],
       [0, 0, 0]])

>>> y
array([[1, 2, 3],
       [4, 5, 6]])

>>> y.flags['C_CONTIGUOUS']
True
```

`ctypes`

An object to simplify the interaction of the array with the `ctypes` module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the `ctypes` module. The returned object has, among others, `data`, `shape`, and `strides` attributes (see Notes below) which themselves return `ctypes` objects that can be used as arguments to a shared library.

Parameters

`None`

Returns

`c` (*Python object*) – Possessing attributes `data`, `shape`, `strides`, etc.

↳ See also

`numpy.ctypeslib`

❶ Notes

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

_ctypes.data

A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as `self._array_interface_['data'][0]`.

Note that unlike `data_as`, a reference will not be kept to the array: code like `ctypes.c_void_p((a + b).ctypes.data)` will result in a pointer to a deallocated array, and should be spelt `(a + b).ctypes.data_as(ctypes.c_void_p)`

_ctypes.shape

A ctypes array of length `self.ndim` where the basetype is the C-integer corresponding to `dtype('P')` on this platform (see `~numpy.ctypeslib.c_intp`). This base-type could be `ctypes.c_int`, `ctypes.c_long`, or `ctypes.c_longlong` depending on the platform. The ctypes array contains the shape of the underlying array.

Type

`(c_intp*self.ndim)`

_ctypes.strides

A ctypes array of length `self.ndim` where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

Type

`(c_intp*self.ndim)`

_ctypes.data_as(obj)

Return the data pointer cast to a particular c-types object. For example, calling `self._as_parameter_` is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: `self.data_as(ctypes.POINTER(ctypes.c_double))`.

The returned pointer will keep a reference to the array.

_ctypes.shape_as(obj)

Return the shape tuple as an array of some other c-types type. For example: `self.shape_as(ctypes.c_short)`.

_ctypes.strides_as(obj)

Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.

If the ctypes module is not available, then the `ctypes` attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the `as_parameter` attribute which will return an integer equal to the `data` attribute.

❶ Examples

```
>>> import ctypes
>>> x = np.array([[0, 1], [2, 3]], dtype=np.int32)
>>> x
array([[0, 1],
       [2, 3]], dtype=int32)
```

```
>>> x.ctypes.data
31962608 # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32))
<__main__.LP_c_uint object at 0x7ff2fc1fc200> # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32)).contents
c_uint(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint64)).contents
c_ulong(4294967296)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1fce60> # may vary
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1ff320> # may vary
```

cumprod(axis=None, dtype=None, out=None)

Return the cumulative product of the elements along the given axis.

Refer to *numpy.cumprod* for full documentation.

 **See also**

numpy.cumprod

equivalent function

cumsum(axis=None, dtype=None, out=None)

Return the cumulative sum of the elements along the given axis.

Refer to *numpy.cumsum* for full documentation.

 **See also**

numpy.cumsum

equivalent function

data

Python buffer object pointing to the start of the array's data.

default_evaluate(parameters, protocol=None)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

diagonal (*offset=0, axis1=0, axis2=1*)

Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to `numpy.diagonal()` for full documentation.

See also

`numpy.diagonal`

equivalent function

`dot()`

`dtype`

Data-type of the array's elements.

Warning

Setting `arr.dtype` is discouraged and may be deprecated in the future. Setting will replace the `dtype` without modifying the memory (see also `ndarray.view` and `ndarray.astype`).

Parameters

`None`

Returns

`d` (*numpy dtype object*)

See also

`ndarray.astype`

Cast the values contained in the array to a new data-type.

`ndarray.view`

Create a view of the same data but a different data-type.

`numpy.dtype`

Examples

```
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

dump (*file*)

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

Parameters

file (*str or Path*) – A string naming the dump file.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

dumps ()

Returns the pickle of the array as a string. pickle.loads will convert the string back to an array.

Parameters

None

evaluate (*evaluator=None*)

Evaluates each item in the array and returns the computed results as an array.

If an item is a computable, its `evaluate()` method is called. Otherwise, the item itself is returned.

Parameters

evaluator (*Optional[Callable[[Evaluable], Any]]*, default: `None`) – Callable passed to each item's `evaluate()` method.

Returns

`Union[ComputableNDArray, ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]]` – The computed results of the items.

fill (*value*)

Fill the array with a scalar value.

Parameters

value (*scalar*) – All elements of *a* will be assigned this value.

Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1., 1.])
```

Fill expects a scalar value and always behaves the same as assigning to a single array element. The following is a rare example where this distinction is important:

```
>>> a = np.array([None, None], dtype=object)
>>> a[0] = np.array(3)
>>> a
array([array(3), None], dtype=object)
>>> a.fill(np.array(3))
>>> a
array([array(3), array(3)], dtype=object)
```

Where other forms of assignments will unpack the array being assigned:

```
>>> a[...] = np.array(3)
>>> a
array([3, 3], dtype=object)
```

flags

Information about the memory layout of the array.

C_CONTIGUOUS (C)

The data is in a single, C-style contiguous segment.

F_CONTIGUOUS (F)

The data is in a single, Fortran-style contiguous segment.

OWNDATA (O)

The array owns the memory it uses or borrows it from another object.

WRITEABLE (W)

The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a `RuntimeError` exception.

ALIGNED (A)

The data and all elements are aligned appropriately for the hardware.

WRITEBACKIFCOPY (X)

This array is a copy of some other array. The C-API function `PyArray_ResolveWritebackIfCopy` must be called before deallocating to the base array will be updated with the contents of this array.

FNC

F_CONTIGUOUS and not C_CONTIGUOUS.

FORC

F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).

BEHAVED (B)

ALIGNED and WRITEABLE.

CARRAY (CA)

BEHAVED and C_CONTIGUOUS.

FARRAY (FA)

BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

 **Notes**

The `flags` object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lower-cased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the WRITEBACKIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling `ndarray.setflags`.

The array flags cannot be set arbitrarily:

- WRITEBACKIFCOPY can only be set `False`.

- `ALIGNED` can only be set `True` if the data is truly aligned.
- `WRITEABLE` can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

`flat`

A 1-D iterator over the array.

This is a `numpy.flatiter` instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

See also

`flatten`

Return a copy of the array collapsed into one dimension.

`flatiter`

Examples

```
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<class 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

flatten(*order='C'*)

Return a copy of the array collapsed into one dimension.

Parameters

order ({'C', 'F', 'A', 'K'}, *optional*) – ‘C’ means to flatten in row-major (C-style) order. ‘F’ means to flatten in column-major (Fortran-style) order. ‘A’ means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. ‘K’ means to flatten *a* in the order the elements occur in memory. The default is ‘C’.

Returns

y (*ndarray*) – A copy of the input array, flattened to one dimension.

↳ **See also**

ravel

Return a flattened array.

flat

A 1-D flat iterator over the array.

➊ **Examples**

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

free_symbols()

Returns the union of free symbols from all children.

Returns

Set[Symbol] – A set containing the free symbols from all children.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as *SymbolSet*.

Returns

SymbolSet – Ordered free symbols in object.

getfield(*dtype*, *offset=0*)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

Parameters

- **dtype** (*str* or *dtype*) – The data type of the view. The dtype size of the view can not be larger than that of the array itself.
- **offset** (*int*) – Number of bytes to skip before beginning the element view.

❶ Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j,  0.+0.j],
       [0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[1.,  0.],
       [0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[1.,  0.],
       [0.,  4.]])
```

`imag`

The imaginary part of the array.

❶ Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0.          ,  0.70710678])
>>> x.imag.dtype
dtype('float64')
```

`is_leaf()`

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

`item(*args)`

Copy an element of an array to a standard Python scalar and return it.

Parameters

`*args` (*Arguments (variable number and type)*) –

- none: in this case, the method only works for arrays with one element (`a.size == 1`), which element is copied into a standard Python scalar object and returned.
- `int_type`: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of `int_types`: functions as does a single `int_type` argument, except that the argument is interpreted as an nd-index into the array.

Returns

`z` (*Standard Python scalar object*) – A copy of the specified element of the array as a suitable Python scalar

i Notes

When the data type of *a* is longdouble or clongdouble, *item()* returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for *item()*, unless fields are defined, in which case a tuple is returned.

item is very similar to *a[args]*, except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

i Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

itemset (*args)

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, *a.itemset(*args)* is equivalent to but faster than *a[args] = item*. The item should be a scalar value and *args* must select a single item in the array *a*.

Parameters

***args (Arguments)** – If one argument: a scalar, only used in case *a* is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

i Notes

Compared to indexing syntax, *itemset* provides some speed increase for placing a scalar into a particular location in an *ndarray*, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using *itemset* (and *item*) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

i Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
```

```
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[2, 2, 6],
       [1, 0, 6],
       [1, 0, 9]])
```

itemsize

Length of one array element in bytes.

Examples

```
>>> x = np.array([1, 2, 3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1, 2, 3], dtype=np.complex128)
>>> x.itemsize
16
```

label: str = None

max (axis=None, out=None, keepdims=False, initial=<no value>, where=True)

Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

See also

numpy.amax

equivalent function

mean (axis=None, dtype=None, out=None, keepdims=False, *, where=True)

Returns the average of the array elements along given axis.

Refer to *numpy.mean* for full documentation.

See also

numpy.mean

equivalent function

min (axis=None, out=None, keepdims=False, initial=<no value>, where=True)

Return the minimum along a given axis.

Refer to *numpy.amin* for full documentation.

 See also

`numpy.amin`
equivalent function

nbytes

Total bytes consumed by the elements of the array.

 Notes

Does not include memory consumed by non-element attributes of the array object.

 See also

`sys.getsizeof`
Memory consumed by the object itself without parents in case view. This does include memory consumed by non-element attributes.

 Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

ndim

Number of array dimensions.

 Examples

```
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

newbyteorder (new_order='S', /)

Return the array with the same data viewed with a different byte order.

Equivalent to:

```
arr.view(arr.dtype.newbyteorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

Parameters

new_order (*string, optional*) – Byte order to force; a value from the byte order specifications below. *new_order* codes can be any of:

- 'S' - swap dtype from current to opposite endian
- {'<', 'little'} - little endian
- {'>', 'big'} - big endian
- {'=', 'native'} - native order, equivalent to `sys.byteorder`
- {'I', 'T'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order.

Returns

new_arr (*array*) – New array object with the dtype reflecting given change to the byte order.

nonzero()

Return the indices of the elements that are non-zero.

Refer to `numpy.nonzero` for full documentation.

↳ **See also**

`numpy.nonzero`
equivalent function

partition (*kth, axis=-1, kind='introselect', order=None*)

Rearranges the elements in the array in such a way that the value of the element in *kth* position is in the position it would be in a sorted array. All elements smaller than the *kth* element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

Added in version 1.8.0.

Parameters

- **kth** (*int or sequence of ints*) – Element index to partition by. The *kth* element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order of all elements in the partitions is undefined. If provided with a sequence of *kth* it will partition all elements indexed by *kth* of them into their sorted position at once.

Deprecated since version 1.22.0: Passing booleans as index is deprecated.

- **axis** (*int, optional*) – Axis along which to sort. Default is -1, which means sort along the last axis.
- **kind** (*{'introselect'}*, *optional*) – Selection algorithm. Default is 'introselect'.
- **order** (*str or list of str, optional*) – When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need to be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

↳ **See also**

numpy.partition
Return a partitioned copy of an array.

argpartition
Indirect partition.

sort
Full sort.

Notes

See `np.partition` for notes on the different algorithms.

Examples

```
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(3)
>>> a
array([2, 1, 3, 4])

>>> a.partition((1, 3))
>>> a
array([1, 2, 3, 4])
```

print_tree()

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

prod(axis=None, dtype=None, out=None, keepdims=False, initial=1, where=True)

Return the product of the array elements over the given axis

Refer to `numpy.prod` for full documentation.

See also

numpy.prod
equivalent function

ptp(axis=None, out=None, keepdims=False)

Peak to peak (maximum - minimum) value along a given axis.

Refer to `numpy.ptp` for full documentation.

See also

numpy.ptp
equivalent function

put (*indices, values, mode='raise'*)
Set `a.flat[n] = values[n]` for all n in *indices*.
Refer to `numpy.put` for full documentation.

↳ See also

`numpy.put`
equivalent function

ravel ([*order*])
Return a flattened array.
Refer to `numpy.ravel` for full documentation.

↳ See also

`numpy.ravel`
equivalent function
`ndarray.flat`
a flat iterator on the array.

real

The real part of the array.

ⓘ Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1.        ,  0.70710678])
>>> x.real.dtype
dtype('float64')
```

↳ See also

`numpy.real`
equivalent function

repeat (*repeats, axis=None*)
Repeat elements of an array.
Refer to `numpy.repeat` for full documentation.

 See also

`numpy.repeat`
equivalent function

`reshape(shape, order='C')`

Returns an array containing the same data with a new shape.

Refer to `numpy.reshape` for full documentation.

 See also

`numpy.reshape`
equivalent function

 Notes

Unlike the free function `numpy.reshape`, this method on `ndarray` allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11))`.

`resize(new_shape, refcheck=True)`

Change shape and size of array in-place.

Parameters

- `new_shape` (tuple of ints, or `n` ints) – Shape of resized array.
- `refcheck` (`bool`, optional) – If False, reference count will not be checked. Default is True.

Returns

`None`

Raises

- `ValueError` – If `a` does not own its own data or references or views to it exist, and the data memory must be changed. PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.
- `SystemError` – If the `order` keyword argument is specified. This behaviour is a bug in NumPy.

 See also

`resize`

Return a new array with the specified shape.

Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set `refcheck` to False.

Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])

>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that references or is referenced ...
```

Unless `refcheck` is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

round (*decimals=0, out=None*)

Return *a* with each element rounded to the given number of decimals.

Refer to `numpy.around` for full documentation.

↳ See also

`numpy.around`
equivalent function

`searchsorted(v, side='left', sorter=None)`

Find indices where elements of `v` should be inserted in `a` to maintain order.

For full documentation, see `numpy.searchsorted`

↳ See also

`numpy.searchsorted`
equivalent function

`setfield(val, dtype, offset=0)`

Put a value into a specified place in a field defined by a data-type.

Place `val` into `a`'s field defined by `dtype` and beginning `offset` bytes into the field.

Parameters

- `val` (`object`) – Value to be placed in field.
- `dtype` (`dtype object`) – Data-type of the field in which to place `val`.
- `offset` (`int, optional`) – The number of bytes into the field at which to place `val`.

Returns

`None`

↳ See also

`getfield`

ⓘ Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]], dtype=int32)
>>> x
array([[1.0e+000, 1.5e-323, 1.5e-323],
       [1.5e-323, 1.0e+000, 1.5e-323],
```

```
[1.5e-323, 1.5e-323, 1.0e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

setflags (*write=None*, *align=None*, *uic=None*)

Set array flags WRITEABLE, ALIGNED, WRITEBACKIFCOPY, respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The WRITEBACKIFCOPY flag can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

Parameters

- ***write*** (*bool*, *optional*) – Describes whether or not *a* can be written to.
- ***align*** (*bool*, *optional*) – Describes whether or not *a* is aligned properly for its type.
- ***uic*** (*bool*, *optional*) – Describes whether or not *a* is a copy of another “base” array.

Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 7 Boolean flags in use, only four of which can be changed by the user: WRITEBACKIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

WRITEBACKIFCOPY (X) this array is a copy of some other array (referenced by .base). When the C-API function PyArray_ResolveWritebackIfCopy is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

Examples

```
>>> y = np.array([[3, 1, 7],
...                 [2, 0, 0],
...                 [8, 5, 9]])
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
```

```

ALIGNED : True
WRITEBACKIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : False
ALIGNED : False
WRITEBACKIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True

```

shape

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with `numpy.reshape`, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

⚠ Warning

Setting `arr.shape` is discouraged and may be deprecated in the future. Using `ndarray.reshape` is the preferred approach.

➊ Examples

```

>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
>>> np.zeros((4,2))[:,2].shape = (-1,)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Incompatible shape for in-place modification. Use
` .reshape()` to make a copy with the desired shape.

```

See also

numpy.shape

Equivalent getter function.

numpy.reshape

Function similar to setting shape.

ndarray.reshape

Method similar to setting shape.

size

Number of elements in the array.

Equal to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

Notes

`a.size` returns a standard arbitrary precision Python integer. This may not be the case with other methods of obtaining the same value (like the suggested `np.prod(a.shape)`, which returns an instance of `np.int_`), and may be relevant if the value is used further in calculations that may overflow a fixed size integer type.

Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

sort (axis=-1, kind=None, order=None)

Sort an array in-place. Refer to `numpy.sort` for full documentation.

Parameters

- **axis** (`int, optional`) – Axis along which to sort. Default is -1, which means sort along the last axis.
 - **kind** (`{'quicksort', 'mergesort', 'heapsort', 'stable'}`, `optional`) – Sorting algorithm. The default is ‘quicksort’. Note that both ‘stable’ and ‘mergesort’ use timsort under the covers and, in general, the actual implementation will vary with datatype. The ‘mergesort’ option is retained for backwards compatibility.
- Changed in version 1.15.0: The ‘stable’ option was added.
- **order** (`str or list of str, optional`) – When `a` is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the `dtype`, to break ties.

↳ See also

numpy.sort
Return a sorted copy of an array.

numpy.argsort
Indirect sort.

numpy.lexsort
Indirect stable sort on multiple keys.

numpy.searchsorted
Find elements in sorted array.

numpy.partition
Partial sort.

ⓘ Notes

See `numpy.sort` for notes on the different sorting algorithms.

ⓘ Examples

```
>>> a = np.array([[1, 4], [3, 1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the `order` keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([(b'c', 1), (b'a', 2)],
      dtype=[('x', 'S1'), ('y', '<i8')])
```

squeeze (axis=None)

Remove axes of length one from *a*.

Refer to `numpy.squeeze` for full documentation.

↳ See also

numpy.squeeze
equivalent function

`std(axis=None, dtype=None, out=None, ddof=0, keepdims=False, *, where=True)`

Returns the standard deviation of the array elements along given axis.

Refer to `numpy.std` for full documentation.

See also

`numpy.std`
equivalent function

strides

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element ($i[0], i[1], \dots, i[n]$) in an array a is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the “ndarray.rst” file in the NumPy reference guide.

Warning

Setting `arr.strides` is discouraged and may be deprecated in the future. `numpy.lib.stride_tricks.as_strided` should be preferred to create a new view of the same data in a safer way.

Notes

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array x will be $(20, 4)$.

See also

`numpy.lib.stride_tricks.as_strided`

Examples

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]],
```

```

[[12, 13, 14, 15],
 [16, 17, 18, 19],
 [20, 21, 22, 23]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813

```

sum (*axis=None, dtype=None, out=None, keepdims=False, initial=0, where=True*)

Return the sum of the array elements over the given axis.

Refer to *numpy.sum* for full documentation.

↳ See also

numpy.sum
equivalent function

swapaxes (*axis1, axis2*)

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to *numpy.swapaxes* for full documentation.

↳ See also

numpy.swapaxes
equivalent function

take (*indices, axis=None, out=None, mode='raise'*)

Return an array formed from the elements of *a* at the given indices.

Refer to *numpy.take* for full documentation.

↳ See also

numpy.take

equivalent function

tobytes (*order='C'*)

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object is produced in C-order by default. This behavior is controlled by the *order* parameter.

Added in version 1.9.0.

Parameters

- **order** ({'C', 'F', 'A'}, *optional*) – Controls the memory layout of the bytes object. ‘C’ means C-order, ‘F’ means F-order, ‘A’ (short for *Any*) means ‘F’ if *a* is Fortran contiguous, ‘C’ otherwise. Default is ‘C’.

Returns

s (*bytes*) – Python bytes exhibiting a copy of *a*’s raw data.

See also

frombuffer

Inverse of this operation, construct a 1-dimensional array from Python bytes.

Examples

```
>>> x = np.array([[0, 1], [2, 3]], dtype='<u2')
>>> x.tobytes()
b'\x00\x00\x01\x00\x02\x00\x03\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x02\x00\x01\x00\x03\x00'
```

tofile (*fid*, *sep=*"", *format=%s*)

Write array to a file as text or binary (default).

Data is always written in ‘C’ order, independent of the order of *a*. The data produced by this method can be recovered using the function `fromfile()`.

Parameters

- **fid** (*file or str or Path*) – An open file object, or a string containing a filename.
Changed in version 1.17.0: *pathlib.Path* objects are now accepted.
- **sep** (*str*) – Separator between array items for text output. If “” (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.
- **format** (*str*) – Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using “format” % item.

Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

When `fid` is a file object, array contents are directly written to the file, bypassing the file object's `write` method. As a result, `tofile` cannot be used with files objects supporting compression (e.g., `GzipFile`) or file-like objects that do not support `fileno()` (e.g., `BytesIO`).

`tolist()`

Return the array as an `a.ndim`-levels deep nested list of Python scalars.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible builtin Python type, via the `~numpy.ndarray.item` function.

If `a.ndim` is 0, then since the depth of the nested list is 0, it will not be a list at all, but a simple Python scalar.

Parameters

`None`

Returns

`y (object, or list of object, or list of list of object, or ...)` – The possibly nested list of array elements.

Notes

The array may be recreated via `a = np.array(a.tolist())`, although this may sometimes lose precision.

Examples

For a 1D array, `a.tolist()` is almost the same as `list(a)`, except that `tolist` changes numpy scalars to Python scalars:

```
>>> a = np.uint32([1, 2])
>>> a_list = list(a)
>>> a_list
[1, 2]
>>> type(a_list[0])
<class 'numpy.uint32'>
>>> a_to_list = a.tolist()
>>> a_to_list
[1, 2]
>>> type(a_to_list[0])
<class 'int'>
```

Additionally, for a 2D array, `tolist` applies recursively:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

The base case for this recursion is a 0D array:

```
>>> a = np.array(1)
>>> list(a)
Traceback (most recent call last):
...
TypeError: iteration over a 0-d array
>>> a.tolist()
1
```

`tostring(order='C')`

A compatibility alias for `tobytes`, with exactly the same behavior.

Despite its name, it returns `bytes` not `strs`.

Deprecated since version 1.19.0.

`trace(offset=0, axis1=0, axis2=1, dtype=None, out=None)`

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

See also

`numpy.trace`

equivalent function

`transpose(*axes)`

Returns a view of the array with axes transposed.

Refer to `numpy.transpose` for full documentation.

Parameters

`axes` (None, tuple of ints, or n ints) –

- None or no argument: reverses the order of the axes.
- tuple of ints: i in the j -th place in the tuple means that the array's i -th axis becomes the transposed array's j -th axis.
- n ints: same as an n -tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form).

Returns

`p (ndarray)` – View of the array with its axes suitably permuted.

See also

`transpose`

Equivalent function.

`ndarray.T`

Array property returning the array transposed.

ndarray.reshape

Give a new shape to an array without changing its data.

i Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])

>>> a = np.array([1, 2, 3, 4])
>>> a
array([1, 2, 3, 4])
>>> a.transpose()
array([1, 2, 3, 4])
```

var (*axis=None, dtype=None, out=None, ddof=0, keepdims=False, *, where=True*)

Returns the variance of the array elements, along given axis.

Refer to *numpy.var* for full documentation.

See also**numpy.var**

equivalent function

view (*[dtype][, type]*)

New view of array with the same data.

i Note

Passing `None` for `dtype` is different from omitting the parameter, since the former invokes `dtype(None)` which is an alias for `dtype('float_')`.

Parameters

- **dtype** (*data-type or ndarray sub-class, optional*) – Data-type descriptor of the returned view, e.g., `float32` or `int16`. Omitting it results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

- **type** (*Python type, optional*) – Type of the returned view, e.g., ndarray or matrix.
Again, omission of the parameter results in type preservation.

❶ Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the last axis of `a` must be contiguous. This axis will be resized in the result.

Changed in version 1.23.0: Only the last axis needs to be contiguous. Previously, the entire array had to be C-contiguous.

❶ Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2), (3, 4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2., 3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0,1] = 20
>>> x
array([(1, 20), (3, 4)], dtype=[('a', 'i1'), ('b', 'i1')])
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1, 3], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int16)
>>> y = x[:, ::2]
>>> y
array([[1, 3],
       [4, 6]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
...
ValueError: To change to a dtype of a different size, the last axis must be contiguous
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[[(1, 3),
          (4, 6)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

However, views that change dtype are totally fine for arrays with a contiguous last axis, even if the rest of the axes are not C-contiguous:

```
>>> x = np.arange(2 * 3 * 4, dtype=np.int8).reshape(2, 3, 4)
>>> x.transpose(1, 0, 2).view(np.int16)
array([[ [ 256,  770],
         [3340, 3854]],
       [[1284, 1798],
        [4368, 4882]],
       [[2312, 2826],
        [5396, 5910]]], dtype=int16)
```

`walk(depth=0)`

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

`depth (int, default: 0)` – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

`class ComputableNode`

Bases: `Evaluatable[EvaluatedType]`

Base class representing a computable node in an expression tree.

`add_label(label, label_children=False)`

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label` (`str`) – Label string to be assigned to node. Overwrites any existing label.
- `label_children` (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

`children()`

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

`default_evaluate(parameters, protocol=None)`

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- `parameters` (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- `protocol` (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

`evaluate(evaluator=None)`

Evaluate this object using the provided evaluator function.

Parameters

`evaluator` (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.

Returns

`Union[TypeVar(EvaluatableType, bound=Evaluatable), TypeVar(EvaluatedType)]` – The computed result.

`free_symbols()`

Returns the union of free symbols from all children.

Returns

`Set[Symbol]` – A set containing the free symbols from all children.

`free_symbols_ordered()`

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

`is_leaf()`

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

label: `str = None`

print_tree()

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk (`depth=0`)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

`depth` (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class ComputableSingleChild

Bases: `ComputableNode[EvaluatedType]`

Computable class which has a single child.

add_label (`label, label_children=False`)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label` (`str`) – Label string to be assigned to node. Overwrites any existing label.
- `label_children` (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

children()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

default_evaluate (`parameters, protocol=None`)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- `parameters` (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- `protocol` (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

evaluate (evaluator=None)

Evaluate this object using the provided evaluator function.

Parameters

`evaluator` (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.

Returns

`Union[TypeVar(TQCone, bound= ComputableSingleChild), TypeVar(EvaluatedType)]` – The computed result.

free_symbols()

Returns the union of free symbols from all children.

Returns

`Set[Symbol]` – A set containing the free symbols from all children.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

label: str = None**print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk (depth=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

`depth` (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class ComputableTuple (*items: Any)

Bases: `tuple, ComputableNode[Tuple]`

Class representing a tuple of items of any types in the computable expression tree.

Parameters

`*items` – One or more items, which can be of any type.

Example

```
>>> from inquanto.computables.primitive import ComputableInt
>>> k_tuple = ComputableTuple(ComputableInt(1), ComputableInt(2), 3, "foo")
>>> k_tuple
(ComputableInt(value=1), ComputableInt(value=2), 3, 'foo')
>>> len(k_tuple)
4
>>> k_tuple.evaluate()
(1, 2, 3, 'foo')
```

`children()`

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator`

`evaluate(evaluator=None)`

Evaluates each item in the tuple and returns the computed results as a tuple.

If an item is a computable, its `evaluate()` method is called. Otherwise, the item itself is returned.

Parameters

`evaluator` (`Optional[Callable[[Evaluatable], Any]]`, default: `None`) – Callable passed to each item's `evaluate()` method.

Returns

`Union[ComputableTuple, Tuple]` – The computed results of the items.

`class Evaluatable`

Bases: `Generic[EvaluatedType]`

Base class for classes that have `evaluate()` methods.

`default_evaluate(parameters, protocol=None)`

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- `parameters` (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- `protocol` (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

`evaluate(evaluator=None)`

Evaluate this object using the provided evaluator function.

Parameters

`evaluator` (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.

Returns

`Union[TypeVar(EvaluatableType, bound= Evaluatable), TypeVar(EvaluatedType)]` –
The computed result.

27.3.3 `inquanto.computables.composite`

Submodule for composite quantum computable expressions. Composed of atomic computables.

`class CommutatorComputable(state, operator_left, operator_right)`

Bases: `ComputableSingleChild[complex]`

Computable expression to calculate the expectation value of commutator between two qubit operators.

Represents the expression $\langle \Psi | [A, B] | \Psi \rangle$.

Parameters

- `state` (`GeneralAnsatz`) – Trial state $|\Psi\rangle$.
- `operator_left` (`QubitOperator`) – Left-hand operator A .
- `operator_right` (`QubitOperator`) – Right-hand operator B .

`add_label(label, label_children=False)`

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label` (`str`) – Label string to be assigned to node. Overwrites any existing label.
- `label_children` (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

`children()`

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

`default_evaluate(parameters, protocol=None)`

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- `parameters` (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- `protocol` (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

evaluate (*evaluator=None*)

Evaluate this object using the provided evaluator function.

Parameters

evaluator (`Optional[Callable[[Evaluable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.

Returns

`Union[TypeVar(TQCOne, bound= ComputableSingleChild), TypeVar(EvaluatedType)]` – The computed result.

free_symbols()

Returns the union of free symbols from all children.

Returns

`Set[Symbol]` – A set containing the free symbols from all children.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

label: str = None**print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk (*depth=0*)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

depth (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class ExpectationValueSumComputable (*states, kernels*)

Bases: `ComputableSingleChild[float]`

Computable expression to calculate the sum of expectation values with different states and hermitian kernels.

Represents the expression $\sum_i \langle \psi_i | A | \psi_i \rangle$ or $\sum_i \langle \psi_i | A_i | \psi_i \rangle$.

Parameters

- **states** (`Sequence[GeneralAnsatz]`) – List of states.
- **kernels** (`Union[QubitOperator, Sequence[QubitOperator]]`) – Hermitian kernel, or list of hermitian kernels. If a list is provided, `kernel[i]` corresponds to `states[i]`.

Raises

ValueError – If a list of `kernels` is provided which is not the same length as `states`.

add_label(*label*, *label_children=False*)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- **label** (`str`) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

children()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

default_evaluate(*parameters*, *protocol=None*)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

evaluate(*evaluator=None*)

Evaluate this object using the provided evaluator function.

Parameters

evaluator (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.

Returns

`Union[TypeVar(TQCOne, bound= ComputableSingleChild), TypeVar(EvaluatedType)]` – The computed result.

free_symbols()

Returns the union of free symbols from all children.

Returns

`Set[Symbol]` – A set containing the free symbols from all children.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

label: str = None**print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk(depth=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

`depth (int, default: 0)` – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class HoleGFComputable(state, kernel, krylov, left=QubitOperator.identity(), right=QubitOperator.identity())

Bases: `ComputableSingleChild[Expr]`

Computable expression for the Many-body hole Green's function.

Internally it measures the moments to compute the Lanczos coefficients, and it will be used to evaluate the Green's function matrix elements.

Parameters

- `state (GeneralAnsatz)` – Initial ansatz state.
- `kernel (QubitOperator)` – Hermitian operator kernel.
- `krylov (int)` – Dimension of the Krylov space.
- `left (QubitOperator, default: QubitOperator.identity())` – Optional operator to transform the moments from the left.
- `right (QubitOperator, default: QubitOperator.identity())` – Optional operator to transform the moments from the right, currently assumed to be the hermitian conjugate of the left.

add_label(label, label_children=False)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label (str)` – Label string to be assigned to node. Overwrites any existing label.
- `label_children (bool, default: False)` – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns`ComputableNode` – Self.**children()**

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type`Iterator[ComputableNode]`**default_evaluate(parameters, protocol=None)**

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.**Parameters**

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns`Union[Evaluatable, Any]` – Final value of the evaluable object.**evaluate(evaluator=None)**

Evaluate this object using the provided evaluator function.

Parameters`evaluator` (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.**Returns**`Expr` – The computed result.**free_symbols()**

Returns the union of free symbols from all children.

Returns`Set[Symbol]` – A set containing the free symbols from all children.**free_symbols_ordered()**Returns the free symbols in increasing lexicographic order as `SymbolSet`.**Returns**`SymbolSet` – Ordered free symbols in object.**is_leaf()**

Check if the current computable node is a leaf (i.e., it has no children).

Returns`bool` – True if the computable node is a leaf, `False` otherwise.**label: str = None****print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type`None`

walk (*depth=0*)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

depth (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class KrylovSubspace (moments)

Bases: `object`

Helper class for Lanczos coefficients.

Calculates α_n and β_n coefficients for the Krylov-space from a list of moments. If the number of moments is $2*n - 1$ then the Krylov-space has a dimension n .

Parameters

moments (`Union[List[float], List[Expr]]`) – List of moments as [$<>$, $<H>$, $<H^2>$, \dots , $<H^{(2*n-1)}>$].

alpha_f (n)

Recursively computes n-th alpha Lanczos coefficient.

Parameters

n (`int`) – The n-th Lanczos coefficient.

Returns

`float` – The value of alpha.

beta_f (n)

Recursively computes n-th beta Lanczos coefficient.

Parameters

n (`int`) – The n-th Lanczos coefficient.

Returns

`float` – The value of beta.

construct_symbolic_recursive_gf (z=Symbol('z'), factor=1.0, shift=0.0, tolerance=1e-8)

Generate an expression for the Green's function with the recursive formulae.

Parameters

- **z** (`Union[float, Expr]`, default: `Symbol("z")`) – Symbol z is the complex energy.
- **factor** (`float`, default: 1.0) – This is +1 or -1, for particle or hole, respectively.
- **shift** (`Union[float, Expr]`, default: 0.0) – The ground energy shift.
- **tolerance** (`float`, default: `1e-8`) – Stop the recursion if bn is smaller than the tolerance.

Returns

`Union[float, complex, Expr]` – Symbolic expression for G(z) approximation.

construct_symbolic_recursive_gf_h (z=Symbol('z'), e0=Symbol('e0'), eta=Symbol('eta'))

Expression for the hole Green's function with the recursive formulae.

Note: Equivalent to `construct_symbolic_recursive_gf(z, -1, e0 + I * eta)`.

Parameters

- **`z`** (`Expr`, default: `Symbol("z")`) – Symbol z is the complex energy.
- **`e0`** (`Expr`, default: `Symbol("e0")`) – Ground state energy.
- **`eta`** (`Symbol`, default: `Symbol("eta")`) – Magnitude of the small imaginary shift.

Returns`Union[float, complex, Expr]` – Symbolic expression.**`construct_symbolic_recursive_gf_p(z=Symbol('z'), e0=Symbol('e0'), eta=Symbol('eta'))`**

Expression for the particle Green's function with the recursive formulae.

Note: Equivalent to `construct_symbolic_recursive_gf(z, +1, e0 + I * eta)`.**Parameters**

- **`z`** (`Expr`, default: `Symbol("z")`) – Symbol z is the complex energy.
- **`e0`** (`Expr`, default: `Symbol("e0")`) – Ground state energy.
- **`eta`** (`Symbol`, default: `Symbol("eta")`) – Magnitude of the small imaginary shift.

Returns`Union[float, complex, Expr]` – Symbolic expression.**`construct_symbolic_recursive_lanczos_gf00(z=Symbol('z'))`**

Expression for the Greens function with the recursive formulae.

Note: Equivalent to `construct_symbolic_recursive_gf(z, +1, 0)`.**Parameters**`z` (`Expr`, default: `Symbol("z")`) – Symbol z is the complex energy.**Returns**`Union[float, complex, Expr]` – Symbolic expression.**`construct_tridiagonal_representation()`**

Constructs the tridiagonal representation of the Hamiltonian.

Returns`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – Tridiagonalized matrix of the Hamiltonian in the Krylov basis.**`eigenvalues()`**

Eigenvalues of the Lanczos matrix.

Returns`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – Array of eigenvalues.**`factors()`**

Calculates the Lanczos coefficients.

Returns`Tuple[List[float], List[float]]` – A tuple of lists, the first list has n alpha values, the second list has n-1 beta values.**`lowest_eigenvalue()`**

The lowest eigenvalues of the Lanczos matrix.

Returns`float` – The lowest eigenvalue.

moments()

Moments internally stored.

Return type

`List[Union[float, Expr]]`

```
class KrylovSubspaceComputable(state, kernel, krylov, left=QubitOperator.identity(),
                                right=QubitOperator.identity())
```

Bases: `ComputableSingleChild[KrylovSubspace]`

Computable expression for `KrylovSubspace` moments.

Based on *arXiv:2009.13140* <<https://arxiv.org/pdf/2009.13140.pdf>>.

Indirectly measures the moments to compute the Lanczos coefficients α_n and β_n of the Krylov-space.

More precisely, internally it will measure $\langle \Psi(\theta) | L * H^n * R | \Psi(\theta) \rangle$ where H is a hermitian operator, $L, R = L^\dagger$ are left and right operators, usually identities.

Parameters

- **state** (`GeneralAnsatz`) – Initial ansatz state.
- **kernel** (`QubitOperator`) – Hermitian operator kernel.
- **krylov** (`int`) – Dimension of the Krylov space.
- **left** (`QubitOperator`, default: `QubitOperator.identity()`) – Optional operator to transform the moments from the left.
- **right** (`QubitOperator`, default: `QubitOperator.identity()`) – Optional operator to transform the moments from the right, currently assumed to be the hermitian conjugate of the left.

add_label(label, label_children=False)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- **label** (`str`) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

children()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

default_evaluate(parameters, protocol=None)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- **parameters** (*SymbolDict*) – *SymbolDict* or dict to map symbols to values.
- **protocol** (*Optional[Any]*, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns`Union[Evaluatable, Any]` – Final value of the evaluable object.**evaluate** (*evaluator=None*)

Evaluate this object using the provided evaluator function.

Parameters`evaluator` (*Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]*, default: `None`) – A callable evaluator that is called on the instance.**Returns**`KrylovSubspace` – The computed result.**free_symbols()**

Returns the union of free symbols from all children.

Returns`Set[Symbol]` – A set containing the free symbols from all children.**free_symbols_ordered()**Returns the free symbols in increasing lexicographic order as `SymbolSet`.**Returns**`SymbolSet` – Ordered free symbols in object.**is_leaf()**

Check if the current computable node is a leaf (i.e., it has no children).

Returns`bool` – True if the computable node is a leaf, `False` otherwise.**label: str = None****print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type`None`**walk** (*depth=0*)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters`depth` (*int*, default: 0) – The initial depth of the tree. Default is 0.**Yields**

A tuple containing the current computable node and its depth in the tree.

Return type`Iterator[Tuple[ComputableNode, int]]`

```
class LanczosCoefficientsComputable(state, kernel, krylov, left=QubitOperator.identity(),
                                     right=QubitOperator.identity())
```

Bases: `ComputableSingleChild[Tuple[List[float], List[float]]]`

Computable expression for the Lanczos coefficients.

Internally it measures the moments to compute the Lanczos coefficients α_n and β_n of the Krylov-space.

Parameters

- **state** (*GeneralAnsatz*) – Initial ansatz state.
- **kernel** (*QubitOperator*) – Hermitian operator kernel.
- **krylov** (*int*) – Dimension of the Krylov space.
- **left** (*QubitOperator*, default: `QubitOperator.identity()`) – Optional operator to transform the moments from the left.
- **right** (*QubitOperator*, default: `QubitOperator.identity()`) – Optional operator to transform the moments from the right, currently assumed to be the hermitian conjugate of the left.

add_label (*label*, *label_children=False*)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.**Parameters**

- **label** (*str*) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (*bool*, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns*ComputableNode* – Self.**children** ()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type*Iterator[ComputableNode]***default_evaluate** (*parameters*, *protocol=None*)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.**Parameters**

- **parameters** (*SymbolDict*) – `SymbolDict` or dict to map symbols to values.
- **protocol** (*Optional[Any]*, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns*Union[Evaluatable, Any]* – Final value of the evaluable object.**evaluate** (*evaluator=None*)

Evaluate this object using the provided evaluator function.

Parameters

evaluator (*Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]*), default: `None`) – A callable evaluator that is called on the instance.

Returns*Tuple[List[float], List[float]]* – The computed result.

free_symbols()

Returns the union of free symbols from all children.

Returns

`Set[Symbol]` – A set containing the free symbols from all children.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

label: str = None**print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk(depth=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

`depth (int, default: 0)` – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class LanczosMatrixComputable(state, kernel, krylov, left=QubitOperator.identity(), right=QubitOperator.identity())

Bases: `ComputableSingleChild[ndarray[Any, dtype[_ScalarType_co]]]`

Computable expression for the tridiagonal Lanczos matrix.

Internally it measures the moments to compute the tridiagonal Lanczos matrix with coefficients α_n and β_n of the Krylov-space.

Parameters

- `state (GeneralAnsatz)` – Initial ansatz state.
- `kernel (QubitOperator)` – Hermitian operator kernel.
- `krylov (int)` – Dimension of the Krylov space.
- `left (QubitOperator, default: QubitOperator.identity())` – Optional operator to transform the moments from the left.
- `right (QubitOperator, default: QubitOperator.identity())` – Optional operator to transform the moments from the right, currently assumed to be the hermitian conjugate of the left.

add_label(label, label_children=False)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label` (`str`) – Label string to be assigned to node. Overwrites any existing label.
- `label_children` (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

children()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

default_evaluate(parameters, protocol=None)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- `parameters` (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- `protocol` (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

evaluate(evaluator=None)

Evaluate this object using the provided evaluator function.

Parameters

`evaluator` (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.

Returns

`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – The computed result.

free_symbols()

Returns the union of free symbols from all children.

Returns

`Set[Symbol]` – A set containing the free symbols from all children.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

label: str = None**print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk(depth=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

`depth (int, default: 0)` – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

```
class ManyBodyGFComputable(fermion_space, ground_state, hamiltonian, krylov,
                           encoding=QubitMappingJordanWigner(), ground_state_energy=None, eta=0)
```

Bases: `ComputableSingleChild[MutableDenseMatrix]`

Computable expression for the 1-particle Many-Body Green's function matrix, in a basis of spin orbitals.

Parameters

- `fermion_space (FermionSpace)` – Fermion occupation space described by this Green's function.
- `ground_state (GeneralAnsatz)` – Ground state.
- `hamiltonian (Union[QubitOperator, FermionOperator])` – Hamiltonian of the system.
- `krylov (int)` – Dimension of the Krylov space.
- `encoding (QubitMapping, default: QubitMappingJordanWigner())` – Fermion to qubit mapping.
- `ground_state_energy (Optional[float], default: None)` – Ground state energy.
- `eta (Union[Symbol, float], default: 0)` – Infinitesimal number for the many-body Green's function.

add_label(label, label_children=False)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label (str)` – Label string to be assigned to node. Overwrites any existing label.
- `label_children (bool, default: False)` – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns`ComputableNode` – Self.**children()**

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type`Iterator[ComputableNode]`**default_evaluate(parameters, protocol=None)**

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.**Parameters**

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns`Union[Evaluatable, Any]` – Final value of the evaluatable object.**default_evaluate_as_function(parameters=None)**

Evaluates to a function with the default protocol.

Evaluates the Green's function matrix with the default protocol and simulator and constructs a function from complex energy to a numpy matrix.

Returns`Callable[[Union[float, complex]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A function $G(z)$ that returns a numpy matrix.**Parameters**`parameters` (`Union[SymbolDict, Dict]`, default: `None`)**evaluate(evaluator=None)**

Evaluate this object using the provided evaluator function.

Parameters`evaluator` (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.**Returns**`MutableDenseMatrix` – The computed result.**evaluate_as_function(evaluator=None)**

Evaluates to a function.

Evaluates the GF matrix and constructs a function from complex energy to a numpy matrix.

Returns`Callable[[Union[float, complex]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A function $G(z)$ that returns a numpy matrix.**Parameters**`evaluator` (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`)

free_symbols()

Returns the union of free symbols from all children.

Returns

`Set[Symbol]` – A set containing the free symbols from all children.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

label: str = None**print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk(depth=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

`depth` (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class NonOrthogonalMatricesComputable(hermitian_operator, states)

Bases: `ComputableNode`

Computable expression representing NO (Non Orthogonal) 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 a list of states, and S is the overlap matrix of the states.

Both, H and S matrices are represented with `OverlapMatrixComputable` internally.

Based on arxiv.org/abs/2205.09039.

Parameters

- `hermitian_operator` (`QubitOperator`) – A Hermitian operator, typically a Hamiltonian, to be expanded in the subspace.
- `states` (`List[GeneralAnsatz]`) – Ansatz states used to span the subspace.

add_label(label, label_children=False)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- **label** (`str`) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns`ComputableNode` – Self.**children()**

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type`Iterator[ComputableNode]`**default_evaluate(parameters, protocol=None)**

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.**Parameters**

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns`Union[Evaluatable, Any]` – Final value of the evaluable object.**evaluate(evaluator=None)**

Evaluate this object using the provided evaluator function.

Parameters`evaluator` (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.**Returns**`Union[NonOrthogonalMatricesComputable, Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]]` – The computed result.**free_symbols()**

Returns the union of free symbols from all children.

Returns`Set[Symbol]` – A set containing the free symbols from all children.**free_symbols_ordered()**Returns the free symbols in increasing lexicographic order as `SymbolSet`.**Returns**`SymbolSet` – Ordered free symbols in object.**is_leaf()**

Check if the current computable node is a leaf (i.e., it has no children).

Returns`bool` – True if the computable node is a leaf, `False` otherwise.

`label: str = None`

`print_tree()`

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

`walk(depth=0)`

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

`depth (int, default: 0)` – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

`class OverlapMatrixComputable(states, kernel=QubitOperator.identity())`

Bases: `ComputableNode`

Computable expression representing an overlap matrix.

The general overlap matrix defined as $S_{ij} = \langle \Psi_i | \hat{O} | \Psi_j \rangle$ where \hat{O} kernel is a Hermitian qubit operator and $|\Psi_i\rangle$ are normalised states as ansatzes.

The diagonal elements of the overlap matrix are represented with `ExpectationValue` computables if the kernel is not identity, while the off-diagonal elements are represented with `Overlap` computables. Since the overlap matrix is a Hermitian matrix, the lower triangular part is excluded from any subsequent measurements or simulation workflow and on evaluation it is computed from the upper triangular part. If the kernel is identity, the diagonal elements are set to 1 and the expectation values are not included into any measurements or simulation workflow.

Parameters

- `states (List[GeneralAnsatz])` – Ansatz states used to span the subspace.
- `kernel (QubitOperator, default: QubitOperator.identity())` – An optional Hermitian operator, by default it is identity.

`add_label(label, label_children=False)`

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label (str)` – Label string to be assigned to node. Overwrites any existing label.
- `label_children (bool, default: False)` – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

`children()`

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

default_evaluate(*parameters*, *protocol*=None)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from [pytket-extensions](#). First, it will try the `AerStateBackend` from [pytket-qiskit](#), and then the `QulacsBackend` from [pytket-qulacs](#).

Parameters

- **parameters** (*SymbolDict*) – *SymbolDict* or dict to map symbols to values.
- **protocol** (*Optional[Any]*, default: None) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

Union[Evaluatable, Any] – Final value of the evaluable object.

evaluate(*evaluator*=None)

Evaluate this object using the provided evaluator function.

Parameters

evaluator (*Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]*, default: None) – A callable evaluator that is called on the instance.

Returns

Union[OverlapMatrixComputable, ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]] – The computed result.

free_symbols()

Returns the union of free symbols from all children.

Returns

Set[Symbol] – A set containing the free symbols from all children.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as *SymbolSet*.

Returns

SymbolSet – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

bool – True if the computable node is a leaf, False otherwise.

label: str = None

print_tree()

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

None

walk(*depth*=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

depth (*int*, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type`Iterator[Tuple[ComputableNode, int]]`

```
class PDM1234RealComputable(space, ansatz, encoding, symmetry_operators, cas_elec, cas_orbs, cu4=True,
taperer=None)
```

Bases: `ComputableSingleChild[Tuple[ndarray[Any, dtype[_ScalarType_co]], ...]]`

Computable expressions for 1,2,3-PDMs of a given state, defined by ansatz and parameters.

Represents the Pre-Density Matrix (PDM) according to doi.org/10.1063/5.0051211.

Parameters

- **space** (`FermionSpace`) – Fermion occupation space spanned by this RDM.
- **ansatz** (`GeneralAnsatz`) – Ansatz state with respect to which expectation values are computed.
- **encoding** (`QubitMapping`) – Fermion to qubit mapping.
- **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-PDM 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.

Note

1,2,3,4-PDMs are returned as a list of arrays, in PySCF-style ordering (<p^r^sq>)

add_label (`label, label_children=False`)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- **label** (`str`) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

children()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

default_evaluate(*parameters*, *protocol*=None)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from [pytket-extensions](#). First, it will try the `AerStateBackend` from [pytket-qiskit](#), and then the `QulacsBackend` from [pytket-qulacs](#).

Parameters

- **parameters** (*SymbolDict*) – *SymbolDict* or dict to map symbols to values.
- **protocol** (*Optional[Any]*, default: None) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

evaluate(**args*, ***kwargs*)

Evaluate this object using the provided evaluator function.

Parameters

- **evaluator** – A callable evaluator that is called on the instance.
- **args** (*Any*)
- **kwargs** (*Any*)

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]], ...]` – The computed result.

free_symbols()

Returns the union of free symbols from all children.

Returns

`Set[Symbol]` – A set containing the free symbols from all children.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as *SymbolSet*.

Returns

SymbolSet – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, False otherwise.

label: str = None

print_tree()

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk(*depth*=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

- **depth** (*int*, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

```
class ParticleGFComputable(state, kernel, krylov, left=QubitOperator.identity(), right=QubitOperator.identity())
```

Bases: `ComputableSingleChild[Expr]`

Computable expression for the Many-body particle Green's function.

Internally it measures the moments to compute the Lanczos coefficients, and it will be used to evaluate the Green's function matrix elements.

Parameters

- **state** (`GeneralAnsatz`) – Initial ansatz state.
- **kernel** (`QubitOperator`) – Hermitian operator kernel.
- **krylov** (`int`) – Dimension of the Krylov space.
- **left** (`QubitOperator`, default: `QubitOperator.identity()`) – Optional operator to transform the moments from the left.
- **right** (`QubitOperator`, default: `QubitOperator.identity()`) – Optional operator to transform the moments from the right, currently assumed to be the hermitian conjugate of the left.

`add_label(label, label_children=False)`

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- **label** (`str`) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

`children()`

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

`default_evaluate(parameters, protocol=None)`

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluatable object.

evaluate (evaluator=None)

Evaluate this object using the provided evaluator function.

Parameters

`evaluator` (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`), default: `None`) – A callable evaluator that is called on the instance.

Returns

`Expr` – The computed result.

free_symbols ()

Returns the union of free symbols from all children.

Returns

`Set[Symbol]` – A set containing the free symbols from all children.

free_symbols_ordered ()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf ()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

label: str = None**print_tree ()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk (depth=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

`depth` (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class QCM4Computable (state, kernel)

Bases: `ComputableSingleChild[KrylovSubspace]`

Computable expression for infimum approximation quantum computed moments up to order of 4.

Based on *arXiv:2311.02533* <<https://arxiv.org/pdf/2311.02533.pdf>>.

Parameters

- `state` (`GeneralAnsatz`)
- `kernel` (`QubitOperator`)

`__init__(state, kernel)`

Quantum computed moment computable to approximate the lowest eigenvalue of the kernel.

Parameters

- **state** (`GeneralAnsatz`) – Initial ansatz state.
- **kernel** (`QubitOperator`) – Hermitian operator kernel.

`add_label(label, label_children=False)`

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- **label** (`str`) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

`children()`

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

`default_evaluate(parameters, protocol=None)`

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

`evaluate(evaluator=None)`

Evaluate this object using the provided evaluator function.

Parameters

- **evaluator** (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.

Returns

`Union[TypeVar(TQCOne, bound= ComputableSingleChild), TypeVar(EvaluatedType)]` – The computed result.

`free_symbols()`

Returns the union of free symbols from all children.

Returns

`Set[Symbol]` – A set containing the free symbols from all children.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

label: str = None**print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk(depth=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

`depth (int, default: 0)` – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class QSEMatricesComputable(state, hermitian_operator, expansion_operators)

Bases: `ComputableNode`

Computable expression representing 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. QSE 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.

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.

add_label(label, label_children=False)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- **label** (`str`) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns`ComputableNode` – Self.**children()**

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type`Iterator[ComputableNode]`**default_evaluate(parameters, protocol=None)**

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.**Parameters**

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns`Union[Evaluatable, Any]` – Final value of the evaluable object.**evaluate(*args, **kwargs)**

Evaluate this object using the provided evaluator function.

Parameters`evaluator` – A callable evaluator that is called on the instance.**Returns**`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – The computed result.**free_symbols()**

Returns the union of free symbols from all children.

Returns`Set[Symbol]` – A set containing the free symbols from all children.**free_symbols_ordered()**Returns the free symbols in increasing lexicographic order as `SymbolSet`.**Returns**`SymbolSet` – Ordered free symbols in object.**is_leaf()**

Check if the current computable node is a leaf (i.e., it has no children).

Returns`bool` – True if the computable node is a leaf, `False` otherwise.

```
label: str = None
```

```
print_tree()
```

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

None

```
walk(depth=0)
```

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

- **depth** (int, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

Iterator[Tuple[ComputableNode, int]]

```
class RDM1234RealComputable(space, ansatz, encoding, symmetry_operators, cas_elec, cas_orbs, cu4=True, taperer=None)
```

Bases: ComputableSingleChild[List[ndarray[Any, dtype[_ScalarType_co]]]]

Computable expression for 1,2,3-RDMs of a given state, defined by ansatz and parameters.

Parameters

- **space** (*FermionSpace*) – Fermion occupation space spanned by this RDM.
- **ansatz** (*GeneralAnsatz*) – Ansatz state with respect to which expectation values are computed.
- **encoding** (*QubitMapping*) – Fermion to qubit mapping.
- **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.

Note

1,2,3,4-RDMs are returned as a list of arrays, in PySCF-style ordering (<p^r^sq>)

```
add_label(label, label_children=False)
```

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- **label** (str) – Label string to be assigned to node. Overwrites any existing label.

- **label_children** (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns`ComputableNode` – Self.**children()**

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type`Iterator[ComputableNode]`**default_evaluate(parameters, protocol=None)**

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.**Parameters**

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns`Union[Evaluatable, Any]` – Final value of the evaluable object.**evaluate(*args, **kwargs)**

Evaluate this object using the provided evaluator function.

Parameters

- **evaluator** – A callable evaluator that is called on the instance.
- **args** (`Any`)
- **kwargs** (`Any`)

Returns`List[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – The computed result.**free_symbols()**

Returns the union of free symbols from all children.

Returns`Set[Symbol]` – A set containing the free symbols from all children.**free_symbols_ordered()**Returns the free symbols in increasing lexicographic order as `SymbolSet`.**Returns**`SymbolSet` – Ordered free symbols in object.**is_leaf()**

Check if the current computable node is a leaf (i.e., it has no children).

Returns`bool` – True if the computable node is a leaf, `False` otherwise.

```
label: str = None
```

```
print_tree()
```

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

```
walk(depth=0)
```

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

- `depth (int, default: 0)` – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

```
class RestrictedOneBodyRDMComputable(fermion_space, ansatz, encoding)
```

Bases: `_BaseCollinearOneBodyRDMComputable`

Computable expression for a `RestrictedOneBodyRDM`.

Parameters

- `fermion_space (FermionSpace)` – Fermion occupation space spanned by this RDM.
- `ansatz (GeneralAnsatz)` – Ansatz state with respect to which expectation values are computed.
- `encoding (QubitMapping)` – Fermion to qubit mapping.

```
add_label(label, label_children=False)
```

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label (str)` – Label string to be assigned to node. Overwrites any existing label.
- `label_children (bool, default: False)` – If True, all child nodes of this computable are labeled with `label`. If False, children remain unlabeled.

Returns

`ComputableNode` – Self.

```
children()
```

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

```
default_evaluate(parameters, protocol=None)
```

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- **parameters** (*SymbolDict*) – *SymbolDict* or dict to map symbols to values.
- **protocol** (*Optional[Any]*, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluatable object.

evaluate (*args, **kwargs)

Evaluate this object using the provided evaluator function.

Parameters

evaluator – A callable evaluator that is called on the instance.

Returns

`RestrictedOneBodyRDM` – The computed result.

free_symbols()

Returns the union of free symbols from all children.

Returns

`Set[Symbol]` – A set containing the free symbols from all children.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

label: str = None**print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk (depth=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

depth (*int*, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class RestrictedOneBodyRDMRealComputable (fermion_space, ansatz, encoding)

Bases: `_BaseCollinearOneBodyRDMComputable`

Computable expression for the real part of a `RestrictedOneBodyRDM`.

Parameters

- **fermion_space** (*FermionSpace*) – Fermion occupation space spanned by this RDM.
- **ansatz** (*GeneralAnsatz*) – Ansatz state with respect to which expectation values are computed.
- **encoding** (*QubitMapping*) – Fermion to qubit mapping.

add_label (*label*, *label_children=False*)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with *label*.

Parameters

- **label** (*str*) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (*bool*, default: *False*) – If *True*, all child nodes of this computable are labeled with *label*. If *False*, children remain unlabeled.

Returns

ComputableNode – Self.

children()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

Iterator[ComputableNode]

default_evaluate (*parameters*, *protocol=None*)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from *pytket-extensions*. First, it will try the *AerStateBackend* from *pytket-qiskit*, and then the *QulacsBackend* from *pytket-qulacs*.

Parameters

- **parameters** (*SymbolDict*) – *SymbolDict* or dict to map symbols to values.
- **protocol** (*Optional[Any]*, default: *None*) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

Union[Evaluatable, Any] – Final value of the evaluable object.

evaluate (**args*, ***kwargs*)

Evaluate this object using the provided evaluator function.

Parameters

evaluator – A callable evaluator that is called on the instance.

Returns

RestrictedOneBodyRDM – The computed result.

free_symbols()

Returns the union of free symbols from all children.

Returns

Set[Symbol] – A set containing the free symbols from all children.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

label: str = None**print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk(depth=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

`depth` (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class SCEOMMatrixComputable(*space, fermion_state, ground_state, mapping, hermitian_operator, expansion_operators, pointgroup=None*)

Bases: `ComputableSingleChild`

Computable Quantum Self Consistent Equation of Motion (SCEOM) matrix.

Implementation based on doi.org/10.1039/D2SC05371C.

Parameters

- `space` (`FermionSpace`) – Fermion Space object.
- `fermion_state` (`FermionState`) – The inquanto datastructure for a parametric-state, which contains a symbolic circuit and a symbol register.
- `ground_state` (`GeneralAnsatz`) – Ground state as ansatz to be used as starting state for the SCEOM matrix.
- `mapping` (`QubitMapping`) – Qubit mappings to be used.
- `hermitian_operator` (`QubitOperator`) – A dictionary mapping QubitPauliStrings to coefficients. A QubitPauliString maps UnitIDs of qubits to Paulis.
- `expansion_operators` (`Iterable[QubitOperator]`) – A list of expansion operators to build a subspace around a reference-state.
- `pointgroup` (`str`, default: `None`) – Point Group symmetry used for reducing matrix elements to be calculated.

`add_label (label, label_children=False)`

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label` (`str`) – Label string to be assigned to node. Overwrites any existing label.
- `label_children` (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

`check_energies (eigvs)`

Returns a computable list of the energies of the SCEOM eigenstates.

Note

This should be the same as the eigenvalues of the SCEOM matrix.

Parameters

`eigvs` (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Eigenvector of the SCEOM matrix.

Returns

`ComputableList` – Computable list of expectation values of the Hamiltonian.

`children ()`

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

`default_evaluate (parameters, protocol=None)`

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.

Parameters

- `parameters` (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- `protocol` (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

`evaluate (evaluator=None)`

Evaluate this object using the provided evaluator function.

Parameters

`evaluator` (`Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]`, default: `None`) – A callable evaluator that is called on the instance.

Returns

`Union[SCEOMMatrixComputable, ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – The computed result.

free_symbols()

Returns the union of free symbols from all children.

Returns

`Set[Symbol]` – A set containing the free symbols from all children.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

get_overlap_computables(eigvs)

Returns a computable list of overlaps of ground state with the SCEOM eigenstates.

Parameters

`eigvs` (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Eigenvector of the SCEOM matrix.

Returns

`ComputableList` – Computable list of overlaps.

get_s2_computables(eigvs)

Returns a computable list of expectation values of S^2 with SCEOM eigenstates.

Parameters

`eigvs` (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Eigenvector of the SCEOM matrix.

Returns

`ComputableList` – Computable list of expectation values.

get_sz_computables(eigvs)

Returns a computable list of expectation values of Sz with SCEOM eigenstates.

Parameters

`eigvs` (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Eigenvector of the SCEOM matrix.

Returns

`ComputableList` – Computable list of expectation values.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

label: str = None**print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk (depth=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

- depth** (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

```
class SpinlessNBodyPDMArrayRealComputable(n, fermion_space, ansatz, encoding, symmetry_operators,
taperer=None)
```

Bases: `ComputableSingleChild[ndarray[Any, dtype[_ScalarType_co]]]`

Computable expression for spinless, n-body PDM (Pre-Density Matrix).

Calculates a general n-body PDM $\gamma_{m \dots n}^{i \dots j} = \langle \Psi_0 | \hat{E}_n^i \hat{E}_n^j \dots | \Psi_0 \rangle$ where \hat{E}_n^i is a spin-traced excitation operator, defined as: $\hat{E}_q^p = \hat{a}_{p\uparrow}^\dagger \hat{a}_{q\uparrow} + \hat{a}_{p\downarrow}^\dagger \hat{a}_{q\downarrow}$. For example, in the four-body case, PDM is:

$$\gamma_{qsuw}^{ptrv} = \langle \Psi_0 | \hat{E}_q^p \hat{E}_s^r \hat{E}_u^t \hat{E}_w^v | \Psi_0 \rangle.$$

Parameters

- **n** (`int`) – Order of n-body PDM.
- **fermion_space** (`FermionSpace`) – Fermion occupation space spanned by this PDM.
- **ansatz** (`GeneralAnsatz`) – Ansatz state with respect to which expectation values are computed.
- **encoding** (`QubitMapping`) – Fermion to qubit mapping.
- **symmetry_operators** (`List[SymmetryOperatorPauli]`) – List of Z2 symmetries of the Hamiltonian.
- **taperer** (`Optional[TapererZ2]`, default: `None`) – Optional taperer object.

add_label (label, label_children=False)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- **label** (`str`) – Label string to be assigned to node. Overwrites any existing label.
- **label_children** (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

children()

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

default_evaluate (*parameters*, *protocol*=*None*)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from [pytket-extensions](#). First, it will try the `AerStateBackend` from [pytket-qiskit](#), and then the `QulacsBackend` from [pytket-qulacs](#).

Parameters

- **parameters** (*SymbolDict*) – *SymbolDict* or dict to map symbols to values.
- **protocol** (*Optional[Any]*, default: *None*) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

Union[Evaluatable, Any] – Final value of the evaluable object.

evaluate (*evaluator*=*None*)

Evaluate this object using the provided evaluator function.

Parameters

evaluator (*Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]*, default: *None*) – A callable evaluator that is called on the instance.

Returns

Union[TypeVar(TQCOne, bound= ComputableSingleChild), TypeVar(EvaluatedType)] – The computed result.

free_symbols ()

Returns the union of free symbols from all children.

Returns

Set[Symbol] – A set containing the free symbols from all children.

free_symbols_ordered ()

Returns the free symbols in increasing lexicographic order as *SymbolSet*.

Returns

SymbolSet – Ordered free symbols in object.

is_leaf ()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

bool – True if the computable node is a leaf, *False* otherwise.

label: str = None

print_tree ()

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

None

walk (*depth*=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

depth (*int*, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type`Iterator[Tuple[ComputableNode, int]]`

```
class SpinlessNBodyRDMArrayRealComputable(n, fermion_space, ansatz, encoding, symmetry_operators,
taperer=None, ordering='p^r^sq')
```

Bases: `ComputableSingleChild[ndarray[Any, dtype[_ScalarType_co]]]`

Computable expression for spinless, n-body RDM matrices.

Calculates a general n-body RDM $\Gamma_{n\dots m}^{i\dots j} = \langle \Psi_0 | \hat{E}_{n\dots m}^{i\dots j} | \Psi_0 \rangle$ where $\hat{E}_{n\dots m}^{i\dots j}$ is a spin-traced excitation operator. For example, in the one-body case this is:

$$\hat{E}_q^p = \hat{a}_{p\uparrow}^\dagger \hat{a}_{q\uparrow} + \hat{a}_{p\downarrow}^\dagger \hat{a}_{q\downarrow}.$$

And in the two-body case:

$$\hat{E}_{qs}^{pr} = \hat{a}_{p\uparrow}^\dagger \hat{a}_{r\uparrow}^\dagger \hat{a}_{s\uparrow} \hat{a}_{q\uparrow} + \hat{a}_{p\uparrow}^\dagger \hat{a}_{r\downarrow}^\dagger \hat{a}_{s\downarrow} \hat{a}_{q\uparrow} + \hat{a}_{p\downarrow}^\dagger \hat{a}_{r\uparrow}^\dagger \hat{a}_{s\uparrow} \hat{a}_{q\downarrow} + \hat{a}_{p\downarrow}^\dagger \hat{a}_{r\downarrow}^\dagger \hat{a}_{s\downarrow} \hat{a}_{q\downarrow}.$$

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[TapererZZ]`, default: `None`) – Optional taperer object.
- `ordering` (`str`, default: "p^r^sq") – RDM index convention. The default corresponds to PySCF.

`add_label(label, label_children=False)`

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label` (`str`) – Label string to be assigned to node. Overwrites any existing label.
- `label_children` (`bool`, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns

`ComputableNode` – Self.

`children()`

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type

`Iterator[ComputableNode]`

default_evaluate(*parameters*, *protocol*=None)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from [pytket-extensions](#). First, it will try the `AerStateBackend` from [pytket-qiskit](#), and then the `QulacsBackend` from [pytket-qulacs](#).

Parameters

- **parameters** (*SymbolDict*) – *SymbolDict* or dict to map symbols to values.
- **protocol** (*Optional[Any]*, default: None) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns

`Union[Evaluatable, Any]` – Final value of the evaluable object.

evaluate(*evaluator*=None)

Evaluate this object using the provided evaluator function.

Parameters

evaluator (*Optional[Callable[[Evaluatable], Union[Evaluatable, Any]]]*, default: None) – A callable evaluator that is called on the instance.

Returns

`Union[TypeVar(TQCOne, bound= ComputableSingleChild), TypeVar(EvaluatedType)]` – The computed result.

free_symbols()

Returns the union of free symbols from all children.

Returns

`Set[Symbol]` – A set containing the free symbols from all children.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

label: str = None

print_tree()

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk(*depth*=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

depth (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type`Iterator[Tuple[ComputableNode, int]]`**class UnrestrictedOneBodyRDMComputable** (*fermion_space*, *ansatz*, *encoding*)Bases: `_BaseCollinearOneBodyRDMComputable`Computable expression for an `UnrestrictedOneBodyRDM`.**Parameters**

- **`fermion_space`** (*FermionSpace*) – Fermion occupation space spanned by this RDM.
- **`ansatz`** (*GeneralAnsatz*) – Ansatz state with respect to which expectation values are computed.
- **`encoding`** (*QubitMapping*) – Fermion to qubit mapping.

add_label (*label*, *label_children=False*)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.**Parameters**

- **`label`** (*str*) – Label string to be assigned to node. Overwrites any existing label.
- **`label_children`** (*bool*, default: `False`) – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns`ComputableNode` – Self.**children()**

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type`Iterator[ComputableNode]`**default_evaluate** (*parameters*, *protocol=None*)

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.**Parameters**

- **`parameters`** (*SymbolDict*) – `SymbolDict` or dict to map symbols to values.
- **`protocol`** (*Optional[Any]*, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns`Union[Evaluatable, Any]` – Final value of the evaluable object.**evaluate** (**args*, ***kwargs*)

Evaluate this object using the provided evaluator function.

Parameters`evaluator` – A callable evaluator that is called on the instance.**Returns**`UnrestrictedOneBodyRDM` – The computed result.

free_symbols()

Returns the union of free symbols from all children.

Returns

`Set[Symbol]` – A set containing the free symbols from all children.

free_symbols_ordered()

Returns the free symbols in increasing lexicographic order as `SymbolSet`.

Returns

`SymbolSet` – Ordered free symbols in object.

is_leaf()

Check if the current computable node is a leaf (i.e., it has no children).

Returns

`bool` – True if the computable node is a leaf, `False` otherwise.

label: str = None**print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type

`None`

walk(depth=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

- `depth (int, default: 0)` – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

class UnrestrictedOneBodyRDMRealComputable(fermion_space, ansatz, encoding)

Bases: `_BaseCollinearOneBodyRDMComputable`

Computable expression for the real part of an `UnrestrictedOneBodyRDM`.

Parameters

- `fermion_space (FermionSpace)` – Fermion occupation space spanned by this RDM.
- `ansatz (GeneralAnsatz)` – Ansatz state with respect to which expectation values are computed.
- `encoding (QubitMapping)` – Fermion to qubit mapping.

add_label(label, label_children=False)

Assign a label to the current computable.

Overwrites any existing label. Access a computable node's label with `label`.

Parameters

- `label (str)` – Label string to be assigned to node. Overwrites any existing label.
- `label_children (bool, default: False)` – If `True`, all child nodes of this computable are labeled with `label`. If `False`, children remain unlabeled.

Returns`ComputableNode` – Self.**children()**

Generator method that yields the child computable nodes of the current computable node.

Yields

An iterator over the child computable nodes of the current computable node.

Return type`Iterator[ComputableNode]`**default_evaluate(parameters, protocol=None)**

Evaluate the final results immediately for return.

If a protocol is not given it will attempt to use statevector backends from `pytket-extensions`. First, it will try the `AerStateBackend` from `pytket-qiskit`, and then the `QulacsBackend` from `pytket-qulacs`.**Parameters**

- **parameters** (`SymbolDict`) – `SymbolDict` or dict to map symbols to values.
- **protocol** (`Optional[Any]`, default: `None`) – Optional protocol is assumed to be capable of measuring and evaluating the internal quantities to calculate the final value.

Returns`Union[Evaluatable, Any]` – Final value of the evaluable object.**evaluate(*args, **kwargs)**

Evaluate this object using the provided evaluator function.

Parameters`evaluator` – A callable evaluator that is called on the instance.**Returns**`UnrestrictedOneBodyRDM` – The computed result.**free_symbols()**

Returns the union of free symbols from all children.

Returns`Set[Symbol]` – A set containing the free symbols from all children.**free_symbols_ordered()**Returns the free symbols in increasing lexicographic order as `SymbolSet`.**Returns**`SymbolSet` – Ordered free symbols in object.**is_leaf()**

Check if the current computable node is a leaf (i.e., it has no children).

Returns`bool` – True if the computable node is a leaf, `False` otherwise.**label: str = None****print_tree()**

Prints the nodes of the computable expression tree, with an indentation level proportional to their depth.

Return type`None`

walk (depth=0)

Generator method to traverse the computable expression tree in a depth-first manner.

Parameters

depth (`int`, default: 0) – The initial depth of the tree. Default is 0.

Yields

A tuple containing the current computable node and its depth in the tree.

Return type

`Iterator[Tuple[ComputableNode, int]]`

27.4 inquanto.core

27.4.1 Parameters Classes

`class SymbolDict (initializer=None, **kwargs)`

Bases: `SymbolTypeKeyDictWrapper`

Holds an ordered map of `sympy.Symbols` to values.

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`) – Input data for building symbol-to-value 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})
>>> SymbolDict({"c": 0.1, "d": 0.2})
SymbolDict({c: 0.1, d: 0.2})
>>> SymbolDict([("sym0", 10), ("sym1", 20)])
SymbolDict({sym0: 10, sym1: 20})
```

`clear()`

Remove all symbols.

Return type

`None`

`copy()`

Performs a deep copy.

Returns

`Union[SymbolTypeKeyDictWrapper, SymbolDict, SymbolSet]` – A deep copy of this object.

❶ 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 `SymbolDict` data.

Return type

`DataFrame`

`discard(*symbols)`

Discard symbols in-place.

Parameters

`symbols` (`Union[str, Symbol]`) – Symbols to be discarded.

Returns

`SymbolTypeKeyDictWrapper` – Updated instance.

`from_array(array)`

Sets symbol values from an input `numpy.ndarray`.

The array must have at least as many elements as there are symbols in the `SymbolDict`.

Parameters

`array` (`Union[ndarray, list]`) – Array of numeric values.

Returns

`SymbolDict` – Updated instance of `SymbolDict`.

`classmethod from_circuit(circuit)`

Instantiate from a pyket circuit.

Parameters

`circuit` (`Circuit`) – Input circuit from which symbols are extracted.

Return type

`Union[SymbolTypeKeyDictWrapper, SymbolDict, SymbolSet]`

`generate_report()`

Generates a report on the symbol order and values in `.json` format.

Return type

`list[dict]`

`items()`

Returns an iterator over symbol-value pairs.

Return type

`ItemsView[Symbol, Union[float, complex, Expr, None]]`

keys ()

Returns an iterator over the symbols.

Return type

`KeysView[Symbol]`

make_hashable ()

Returns a hashable object corresponding to the `SymbolDict`.

Return type

`str`

print_report ()

Prints the `SymbolDict`.

Return type

`None`

set (symbol, value=None)

Adds or updates a symbol-value pair.

Parameters

- **symbol** (`Union[str, Symbol]`) – A symbol to add/update.
- **value** (`Union[float, complex, Expr, None]`, default: `None`) – Value assigned to symbol.

Return type

`None`

property symbols: list[Symbol]

Returns a list of all symbols.

to_array ()

Returns `SymbolDict` values as a `numpy.ndarray`.

Return type

`ndarray`

to_dict ()

Returns the underlying Python `dict`.

Return type

`dict[Symbol, Union[float, complex, Expr, None]]`

update (other)

Update the symbol values based on another `SymbolDict`.

The order is defined by the symbols first appearance.

Parameters

- **other** (`Union[SymbolDict, dict[Symbol, Union[float, complex, Expr, None]]]`) – Symbol-to-value map from which to update.

Returns

`SymbolDict` – Updated instance of `SymbolDict`.

values ()

Returns an iterator over the values.

Return type

`ValuesView[Union[float, complex, Expr, None]]`

```
class SymbolSet(iterable=None)
```

Bases: `SymbolTypeKeyDictWrapper`

Holds an ordered set of `sympy.Symbols`.

It can be converted to a `SymbolDict` by assigned values to the symbols, using provided class methods.

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.

Parameters

`symbol` (`Union[str, Symbol]`) – The symbol to be added. If it already exists in the `SymbolSet`, the set remains unchanged.

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 given an array of values.

The new `SymbolDict` object is constructed from the original `SymbolSet` instance and the input array. Values from the input array are assigned to symbols according to the order of the `SymbolSet`.

Parameters

`array` (`Union[numpy.ndarray, list]`) – Array containing values to be assigned.

Returns

SymbolDict – New *SymbolDict* instance.

Raises

AssertionError – When the input array and *SymbolSet* have different lengths.

construct_from_dict (other)

Constructs a *SymbolDict* object given a symbol-to-value map.

The new *SymbolDict* contains all the symbols from the original *SymbolSet* instance, with values assigned according to other.

Parameters

other (`Union[SymbolDict, dict]`) – A symbol-to-value map.

Returns

SymbolDict – A new *SymbolDict* instance.

Raises

RuntimeError – When other 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 with random values.

The new *SymbolDict* object contains all symbols from the original *SymbolSet*, with values drawn randomly from a Gaussian distribution.

Parameters

- **seed** (default: 0) – Seed for random number generation.
- **mu** (default: 0.0) – Mean of the distribution.
- **sigma** (default: 1.0) – Standard deviation of the distribution.

Returns

SymbolDict – New *SymbolDict* instance.

construct_zeros ()

Constructs a *SymbolDict* object with all values set to zero.

The new *SymbolDict* object contains all symbols from the original *SymbolSet*, with values set to zero.

Returns

SymbolDict – New *SymbolDict* instance.

copy ()

Performs a deep copy.

Returns

`Union[SymbolTypeKeyDictWrapper, SymbolDict, SymbolSet]` – A deep copy of this object.

i 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 `SymbolSet` data.

Return type

`DataFrame`

discard(*symbols)

Discard symbols in-place.

Parameters

`symbols` (`Union[str, Symbol]`) – Symbols to be discarded.

Returns

`SymbolTypeKeyDictWrapper` – Updated instance.

classmethod from_circuit(circuit)

Instantiate from a pytket circuit.

Parameters

`circuit` (`Circuit`) – Input circuit from which symbols are extracted.

Return type

`Union[SymbolTypeKeyDictWrapper, SymbolDict, SymbolSet]`

make_hashable()

Returns a hashable object corresponding to the `SymbolSet`.

Return type

`str`

pop()

Removes the first element of the set and returns it.

Returns

`Union[str, Symbol]` – The removed element.

renamed(name_function=None, prefix='new_')

Returns a new set with the same order but renamed.

Parameters

- **name_function** (`Optional[Callable[[str], str]]`, default: `None`) – Function to rename symbols.
- **prefix** (`str`, default: `"new_"`) – If `name_function` is not provided, `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.

update (other)

Update the `SymbolSet` based on another iterable of symbols.

The order is defined by a symbol's first appearance.

Parameters

`other` (`Iterable[Union[str, Symbol]]`) – Symbols from which to update.

Returns

`SymbolSet` – Updated instance of `SymbolSet`.

27.4.2 Cache

class Cache (level=CacheLevels.LIFETIME, max_mem_size=None, size_unit=CacheSizeUnit.MB, key_generator=None)

Bases: `object`

Implements general cache handling.

Parameters

- **level** (`CacheLevels`, default: `CacheLevels.LIFETIME`) – Cache level (`CacheLevels.NONE`: nothing is cached, or `CacheLevels.LIFETIME`: cache is not cleared during lifetime of the object, using it (unless memory used exceeds `max_mem_size` parameter value).
- **max_mem_size** (`Optional[int]`, default: `None`) – Maximum memory size that cache is allowed to occupy. Not checked if set to `None`.
- **size_unit** (`CacheSizeUnit`, default: `CacheSizeUnit.MB`) – Memory size unit (used to process input and output).
- **key_generator** (`Optional[Callable[..., Hashable]]`, default: `None`) – Function to generate a key from the input objects.

Raises

`ValueError` – When `max_mem_size` parameter has negative value.

property cache: Dict[Any, Any]

Returns the internal dict, storing cache.

property check_mem: bool

Returns whether the size of memory occupied by cache is to be checked at runtime.

Notes

Currently the `Cache` class does not check for memory size itself (except for the report). Such checks are supposed to be done by a handler (e.g. a caching decorator).

clear()

Clears cache.

Return type

`None`

hashkey(*args, **kwargs)

Creates a hashable key from input arguments.

Optionally uses a key generator function if passed at the class instance construction.

Parameters

- `args` (`Any`) – Arguments to be made hashable.
- `kwargs` (`Any`) – Keyword arguments, which values are to be made hashable.

Returns

`Tuple[Any, ...]` – A list of hashable objects, to be used as a cache dict key.

Raises

`RuntimeError` – If created key is not hashable.

property level: CacheLevels

Returns the cache level.

property max_mem_size: int | None

Returns the maximum allowed memory size for cache.

property mem_size: float

Calculates size of memory occupied by cache.

Currently uses implementation given `here` and converts result (in bytes) to the set memory size units.

property num_entries: int

Returns the number of entries currently stored in cache.

report()

Returns a pandas `DataFrame` object containing a report of the current state of cache.

Return type

`DataFrame`

property size_unit: CacheSizeUnit

Returns the memory size unit as used by the class instance.

class CacheLevels(value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)

Bases: `IntEnum`

Enum class containing possible cache levels.

LIFETIME = 3**NONE** = 1**RUNTIME** = 2**as_integer_ratio()**

Return integer ratio.

Return a pair of integers, whose ratio is exactly equal to the original int and with a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

conjugate()

Returns self, the complex conjugate of any int.

denominator

the denominator of a rational number in lowest terms

from_bytes(*byteorder='big'*, *, *signed=False*)

Return the integer represented by the given array of bytes.

bytes

Holds the array of bytes to convert. The argument must either support the buffer protocol or be an iterable object producing bytes. Bytes and bytearray are examples of built-in objects that support the buffer protocol.

byteorder

The byte order used to represent the integer. If byteorder is ‘big’, the most significant byte is at the beginning of the byte array. If byteorder is ‘little’, the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value. Default is to use ‘big’.

signed

Indicates whether two’s complement is used to represent the integer.

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

real

the real part of a complex number

to_bytes (*length=1*, *byteorder='big'*, *, *signed=False*)

Return an array of bytes representing an integer.

length

Length of bytes object to use. An OverflowError is raised if the integer is not representable with the given number of bytes. Default is length 1.

byteorder

The byte order used to represent the integer. If byteorder is ‘big’, the most significant byte is at the beginning of the byte array. If byteorder is ‘little’, the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value. Default is to use ‘big’.

signed

Determines whether two’s complement is used to represent the integer. If signed is False and a negative integer is given, an OverflowError is raised.

```
class CacheSizeUnit (value, names=None, *, module=None, qualname=None, type=None, start=1,  
                  boundary=None)
```

Bases: `IntEnum`

Enum class containing possible memory units.

BYTES = 1

GB = 4

KB = 2

MB = 3

as_integer_ratio()

Return integer ratio.

Return a pair of integers, whose ratio is exactly equal to the original int and with a positive denominator.

```
>>> (10).as_integer_ratio()  
(10, 1)  
>>> (-10).as_integer_ratio()  
(-10, 1)  
>>> (0).as_integer_ratio()  
(0, 1)
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

conjugate()

Returns self, the complex conjugate of any int.

denominator

the denominator of a rational number in lowest terms

from_bytes (byteorder='big', *, signed=False)

Return the integer represented by the given array of bytes.

bytes

Holds the array of bytes to convert. The argument must either support the buffer protocol or be an iterable object producing bytes. Bytes and bytearray are examples of built-in objects that support the buffer protocol.

byteorder

The byte order used to represent the integer. If byteorder is ‘big’, the most significant byte is at the beginning of the byte array. If byteorder is ‘little’, the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value. Default is to use ‘big’.

signed

Indicates whether two’s complement is used to represent the integer.

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

real

the real part of a complex number

to_bytes (length=1, byteorder='big', *, signed=False)

Return an array of bytes representing an integer.

length

Length of bytes object to use. An OverflowError is raised if the integer is not representable with the given number of bytes. Default is length 1.

byteorder

The byte order used to represent the integer. If byteorder is ‘big’, the most significant byte is at the beginning of the byte array. If byteorder is ‘little’, the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value. Default is to use ‘big’.

signed

Determines whether two's complement is used to represent the integer. If signed is False and a negative integer is given, an OverflowError is raised.

27.4.3 Logging and Timing

`class Timer`

Bases: `object`

Simple Timer.

`start()`

Starts timer.

Return type

`None`

`stop()`

Stops timer.

Return type

`None`

`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: str`

Filename base.

`property prefix: str`

Filename prefix.

27.4.4 Methods

InQuanto's core module containing utility methods and classes useful in multiple submodules.

`dict_to_matrix(ordered_symbols, symbol_pair_to_value)`

Return a matrix of values with specified order.

Parameters

- `ordered_symbols` (`Sequence[Symbol]`) – Sequence of symbols defining output order.
- `symbol_pair_to_value` (`dict[tuple[Symbol, Symbol], float]`) – Map of symbol-pairs to values.

Returns

`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – Matrix of values with axes ordered according to `ordered_symbols`.

`dict_to_vector(ordered_symbols, symbol_to_value)`

Return an array of values with specified order.

Parameters

- `ordered_symbols` (`Sequence[Symbol]`) – Sequence of symbols defining output order.
- `symbol_to_value` (`dict[Symbol, float]`) – Map of symbols to values.

Returns

`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – Values ordered according to `ordered_symbols`.

`matrix_to_dict(ordered_symbols, matrix)`

Converts a matrix of values to a map of symbol-pairs to values.

Parameters

- `ordered_symbols` (`Sequence[Symbol]`) – Sequence of symbols defining output order.
- `matrix` (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Matrix of values to be assigned to symbol-pairs.

Returns

`dict[tuple[Symbol, Symbol], float]` – Map of symbol-pairs to values.

pd_safe_eigh(*h, s, lindep=1e-14*)

Solves the generalized eigenvalue problem $HC = SCE$.

H and *S* are square matrices, and *C* is a matrix of eigenvectors. The matrices *H* and *S* are usually Hermitian (or symmetric if real) and *S* is positive definite.

Parameters

- **h** (ndarray[*Any*, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]) – Hermitian matrix.
- **s** (ndarray[*Any*, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]) – Hermitian matrix.
- **lindep** (*float*, default: $1e-14$) – Tolerance to determine linear dependency. Diagonalizing *S* identifies eigenvectors as linearly dependent subsets when their corresponding eigenvalues fall below the specified threshold.

Returns

tuple[ndarray[*Any*, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]], ndarray[*Any*, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]], ndarray[*Any*, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]] – Eigenvalues *E* in the linearly independent subspace, matrix of eigenvectors *C*, and eigenvalues of *S*.

vector_to_dict(*ordered_symbols, vector*)

Converts a vector of values to a symbol-value map.

Parameters

- **ordered_symbols** (Sequence[Symbol]) – Sequence of symbols defining output order.
- **vector** (ndarray[*Any*, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]) – Values to be assigned to symbols.

Returns

dict[Symbol, float] – Map of symbols to values.

27.5 inquanto.embeddings

inquanto.embeddings module provides classes to decompose a system to smaller fragments for which more accurate theories can be applied.

27.5.1 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,
               check_unitary=False, check_unitary_atol=1e-10)
```

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** (`Union[ChemistryRestrictedIntegralOperator, TypeVar(PySCFRestrictedIntegralOperatorT, bound= PySCFChemistryRestrictedIntegralOperator)]`) – 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 $|u - u'| < \text{scf_tolerance}$.
- **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, $|N - N'| < \text{newton_tol}$.
- **occupation_rtol** (`float`, default: `1e-5`) – Relative tolerance (`numpy.isclose`) to check orbital occupation.
- **occupation_atol** (`float`, default: `1e-8`) – Absolute tolerance (`numpy.isclose`) to check orbital occupation.
- **check_unitary** (`bool`, default: `False`) – Whether to perform the check for unitarity of the rotation matrix before fragment rotation basis change.
- **check_unitary_atol** (`float`, default: `1e-10`) – Absolute tolerance of unitarity check.

static construct_random_parameters (`pattern, seed=0, mu=0.0, sigma=0.01`)

Generates an array of initial parameters randomized according to a 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 (μ) for the distribution.
- **sigma** (`float`, default: 0.01) – Standard deviation (σ) 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

`ndarray` – 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 from 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 stores 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, Optional[float], Optional[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 details, please see the InQuanto user guide.

Parameters

- **dmet** (`DMETRHF`) – Instance of `DMETRHF` 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** (Optional[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 and bath) that includes the energy and one- and two-particle reduced density matrices (1-RDM and 2-RDM).

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, 1-RDM, 2-RDM 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 details, please see the InQuanto user guide.

Parameters

- **dmet** (*DMETRHF*) – Instance of *DMETRHF* object 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** (Optional[str], default: None) – Name of the fragment.
- **frozen** (Optional[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 democratic 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, 1-RDM 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, 1-RDM 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 details, please see the InQuanto user guide.

Parameters

- `dmet` (`DMETRHF`) – Instance of `DMETRHF` object 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` (Optional[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 democratic 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 and 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, 1-RDM 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 corresponding `solve_active()` method with the custom quantum algorithm.

Parameters

- `dmet` (`DMETRHF`) – Instance of `DMETRHF` object 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` (Optional[str], default: None) – Name of the fragment.
- `frozen` (Optional[list[int]], default: None) – List of indices of frozen spatial orbitals.
- `backend` (Optional[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 democratic 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, 1-RDM 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, check_unitary=False, check_unitary_atol=1e-10)
```

Bases: `object`

Basic Impurity ROHF-DMET class to drive the Impurity DMET 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** (`float`, default: `1e-5`) – Relative tolerance (`isclose`) to check orbital occupation.
- **occupation_atol** (`float`, default: `1e-8`) – Absolute tolerance (`isclose`) to check orbital occupation.
- **check_unitary** (`bool`, default: `False`) – Whether to perform the check for unitarity of the rotation matrix before fragment rotation basis change.
- **check_unitary_atol** (`float`, default: `1e-10`) – Absolute tolerance of unitarity check.

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 `ImpurityDMETROHF`.

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 details, please see the InQuanto user guide.

Parameters

- `dmet` (`ImpurityDMETROHF`) – Instance of `ImpurityDMETROHF` object 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` (`Optional[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 and 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, 1-RDM 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 details, please see the InQuanto user guide.

Parameters

- `dmet` (`ImpurityDMETROHF`) – Instance of `ImpurityDMETROHF` object 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` (`Optional[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 and 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`) – Instance of `ImpurityDMETROHF` object 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** (`Optional[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 and 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 is that this fragment solver does not require the computation of a 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 details, please see the InQuanto user guide.

Parameters

- **dmet** (*ImpurityDMETROHF*) – Instance of *ImpurityDMETROHF* object 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** (*Optional[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 and bath).

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).

27.6 inquanto.express

27.6.1 Express drivers

```
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** (`Union[float, list[float]]`, default: 0.0) – on-site energies. If a float is given, applied to all sites. If list is given, its len must equal n, and index corresponds to site.
- **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** (`Union[float, list[float]]`) – Value(s) substituted to all 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** (`Union[float, list[float]]`) – Value(s) 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.

```
property n_orb: int
```

Returns the number of orbitals in the total system.

```
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.

property n_orb: int
    Returns the number of orbitals in the total system.

print_json_report(*args, **kwargs)
    Prints report in json format.

class DriverIsingCustomConnectivity(j, h, connectivity_matrix)
    Bases: GeneralDriver
    Driver for a transverse-field Ising hamiltonian with general lattice topology from a qubit connectivity matrix.

    Builds the hamiltonian:  $H = \sum_{i < j}^{N_q} J_{ij} Z_i Z_j + h \sum_i^{N_q} X_i$ , where  $N_q$  is the number of qubits, and  $J_{ij}$  is the product of the interaction strength  $j$  and the connectivity matrix.

    Parameters
        • j (float) – Interaction strength (negative for ferromagnetic, positive for antiferromagnetic).
        • h (float) – Transverse field.
        • connectivity_matrix (ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]) – Lattice connectivity matrix. Only the lower triangular elements are used.

generate_report(*args, **kwargs)
    Generates report in a hierarchical dictionary format.

get_system()
    Return the qubit hamiltonian, space and state.

    Returns
        Tuple[QubitOperator, QubitSpace, QubitState] – TFIM qubit hamiltonian, corresponding qubit space, qubit state of zeroes (uniform ferromagnetic state).
```

```
print_json_report(*args, **kwargs)
    Prints report in json format.
```

```
class DriverIsing1D(j, h, n)
    Bases: DriverIsingCustomConnectivity
```

Driver for a 1D, nearest-neighbour transverse-field Ising hamiltonian with open boundary conditions.

Builds the hamiltonian: $H = J \sum_i^{N_q} Z_i Z_{i+1} + h \sum_i^{N_q} X_i$, where N_q is the number of qubits.

Parameters

- **j** (`float`) – Interaction strength (negative for ferromagnetic, positive for antiferromagnetic).
- **h** (`float`) – Transverse field.
- **n** (`int`) – Number of sites (qubits).

```
generate_report(*args, **kwargs)
```

Generates report in a hierarchical dictionary format.

```
get_system()
```

Return the qubit hamiltonian, space and state.

Returns

`Tuple[QubitOperator, QubitSpace, QubitState]` – TFIM qubit hamiltonian, corresponding qubit space, qubit state of zeroes (uniform ferromagnetic state).

```
print_json_report(*args, **kwargs)
```

Prints report in json format.

```
class DriverIsing1DRing(j, h, n)
```

```
    Bases: DriverIsingCustomConnectivity
```

Driver for a 1D, nearest-neighbour transverse-field Ising hamiltonian with periodic boundary conditions.

Builds the hamiltonian: $H = J \sum_i^{N_q} Z_i Z_{i+1} + h \sum_i^{N_q} X_i$, where N_q is the number of qubits. The first and last qubits are connected to form a ring geometry.

Parameters

- **j** (`float`) – Interaction strength (negative for ferromagnetic, positive for antiferromagnetic).
- **h** (`float`) – Transverse field.
- **n** (`int`) – Number of sites (qubits).

```
generate_report(*args, **kwargs)
```

Generates report in a hierarchical dictionary format.

```
get_system()
```

Return the qubit hamiltonian, space and state.

Returns

`Tuple[QubitOperator, QubitSpace, QubitState]` – TFIM qubit hamiltonian, corresponding qubit space, qubit state of zeroes (uniform ferromagnetic state).

```
print_json_report(*args, **kwargs)
```

Prints report in json format.

27.6.2 Express functions

Data for example systems for testing.

`get_noisy_backend(n_qubits, cx_err=0.008, ro_err=0.01)`

Makes a noisy Aer backend with specified error rates.

This function uses the AerBackend from pytket.extensions.qiskit and sets up a custom noise model with specified depolarizing error rates for CX gates and readout errors for qubits.

Parameters

- `n_qubits` (`int`) – The number of qubits for the noisy backend.
- `cx_err` (`float`, default: 0.008) – The depolarizing error rate for CX gates. Defaults to 0.008.
- `ro_err` (`float`, default: 0.01) – The readout error rate for the qubits. Defaults to 0.01.

Returns

`Backend` – A noisy Aer backend instance constructed with the specified noise model.

Examples

```
>>> backend = get_noisy_backend(5)
>>> type(backend)
<class 'pytket.extensions.qiskit.backends.aer.AerBackend'>
```

`get_system(data_in)`

Return the fermionic Hamiltonian operator, Fock space, and Hartree Fock state from an express dataset.

Parameters

`data_in` (`Union[str, dict[str, Any], tuple]`) – An express dataset file name, or previously loaded express dataset.

Returns

`Tuple[Union[ChemistryRestrictedIntegralOperator, ChemistryUnrestrictedIntegralOperator], FermionSpace, FermionState]` – Fermion Hamiltonian, Fock space, Fock state.

`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.

propagate (*qubit_hamiltonian*, *qubit_state*, *t*)

Propagate a qubit state with a Hamiltonian to a point of time.

Parameters

- **qubit_hamiltonian** (*QubitOperator*) – A Hamiltonian qubit operator.
- **qubit_state** (*QubitState*) – The initial qubit state.
- **t** (*Union[float, complex]*) – The propagation time.

Returns

The evolved qubit state, $|\Psi(t)\rangle = e^{-itH}|\Psi(0)\rangle$

random_circuit_ansatz (*n_qubit*=3, *seed*=0, *return_sv*=False)

Constructs a random circuit ansatz.

A random normalised complex statevector will be generated with a seed, and by using *StatePreparationBox* a circuit will be built.

Parameters

- **n_qubit** (*int*, default: 3) – Number of qubits.
- **seed** (*int*, default: 0) – Random seed.
- **return_sv** (*bool*, default: False) – If True, the function also returns the statevector.

Returns

Union[CircuitAnsatz, Tuple[CircuitAnsatz, ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]] – A circuit ansatz built with *StatePreparationBox* from pytket, and optionally the statevector.

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 hamiltonian 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_time_evolution (*initial_state*, *time_span*, *qubit_hamiltonian*, **operators*, *n_qubits=None*, *real=True*)

Simulating exact time evolution.

Parameters

- **initial_state** (*QubitState*) – The initial qubit state for the time evolution.
- **time_span** (*Sequence[Union[float, complex]]*) – The time span, which is a list of points of time.
- **qubit_hamiltonian** (*QubitOperator*) – The Hamiltonian to drive the time evolution.
- ***operators** (*QubitOperator*) – Optional operators for which the expectation value should be calculated during the evolution.
- **real** (*bool*, default: *True*) – Optional to switch to imaginary time evolution. Default value is real.
- **n_qubits** (*int*, default: *None*)

Returns

Tuple[List[QubitState], ...] – The qubit states for each time in the time span and the expectation values.

run_vqe (*ansatz*, *hamiltonian*, *backend*, *with_gradient=True*,
minimizer=MinimizerScipy(method=OptimizationMethod.L_BFGS_B_smooth), *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** (*GeneralAnsatz*) – 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=OptimizationMethod.L_BFGS_B_smooth)`) – 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
```

save_h5_system (`fname`, `ham`, `fermion_space`, `hf_state`, `**kwargs`)

Save essential system information to the *.h5 file.

This function can be used both for molecular and periodic systems.

Parameters

- **fname** (`str`) – h5 filename to save the system to.
- **ham** (`FermionOperator`) – fermionic hamiltonian for the system.
- **fermion_space** (`FermionSpace`) – fermion space object for the system.
- **hf_state** (`FermionState`) – fermionic reference state for the system.
- ****kwargs** – Additional data to be saved.

27.7 inquanto.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, DataFrame, list, None]`, default: `None`) – Geometrical information - can be a list of lists (e.g `[['H', [0, 0, 0]], ['H', [0, 0, 1]]]`), a zmatrix string, xyz coordinates as a string or `pandas.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.

Parameters

- **element** (`str`) – Element symbol.
- **position** (`Optional[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]]`, 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 to align the bond defined by the elements of the `atom_ids` list to a coordinate axis.

Align

Parameters

- **atom_ids** (`list`) – Two atom indices which define the bond to align to the axis.
- **axis** (`str`) – A cartesian co-ordinate axis, options are "X", "Y", "Z", "x",
:param "y": :param "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 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_co, bound= generic, covariant=True)]]`) – The vector of length three that we use to align the bond.

Returns

`DataFrame` – The updated `dataframe` object.

align_to_plane (`atom_ids, vectors`)

Rotate and translate the geometry such that three atoms lie in the plane defined by the vectors provided.

Parameters

- **atom_ids** (`list[int]`) – The indices of three atoms in the geometry that we use to align to the plane.

- **vectors** (`list[ndarray[Any, dtype[_ScalarType_co], bound= generic, covariant=True]]`) – An iterable containing three vectors that define the plane.

Returns

`DataFrame` – The updated `dataframe` object.

align_to_xy_plane (`atom_ids=None`)

Align any three atoms to the xy plane.

Rotate and translate the geometry such that the plane defined by the three atom indices in `atom_ids` becomes the xy plane.

Parameters

- `atom_ids` (`Optional[list[int]]`, default: `None`) – The indices of three atoms in the geometry that we align to the xy 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 three atom indices in the `atom_ids` variable becomes the xz plane.

Parameters

- `atom_ids` (`Optional[list[int]]`, default: `None`) – The indices of three atoms in the geometry that we align to the xz plane.

Returns

`DataFrame` – The updated `dataframe` object.

align_to_yz_plane (`atom_ids=None`)

Align any three atoms to the yz plane.

Rotate and translate the geometry such that the plane defined by the three atom indices in `atom_ids` becomes the yz plane.

Parameters

- `atom_ids` (`Optional[list[int]]`, default: `None`) – The indices of three atoms in the geometry that we align to the yz plane.

Returns

`DataFrame` – The updated `dataframe` object.

property atomic_coordinates: ndarray[Any, dtype[_ScalarType_co]]

Get an array of position vectors of each atom in the geometry.

Each row corresponds to one position vector.

Returns

An array of floats where each row at index i contains the x, y, z coordinates of atom i .

bond_angle (`atom_ids, units='deg'`)

Compute the bond angle defined by the three atoms indexed by the elements of `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]`) – An iterable containing the indices of three atoms to use in calculating the angle.

- **units** (`str`, default: "deg") – Units of the reported bond angle, options are "deg" or "rad".

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]`) – An 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 of element A.
- **a** (`float`) – Distance between A and B in the AB pairs.
- **bb** (`str`) – Atomic symbol of element B.
- **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 of the element.
- **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.

Each *i, j* th element is the internuclear separation between atom *i* and *j*.**Returns**

`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` –
The distance matrix.

property dataframe: DataFrameReturn the geometry in a `pandas.DataFrame` object.Each row corresponds to an atom in the geometry and each column holds the atomic symbol of an element and their *x, y, z* coordinates.**Returns**An `pandas.DataFrame` containing a geometry.**delete_atom**(*atom_index*)

Delete an atom from the geometry.

Parameters

- **atom_index** (`Union[int, list]`) – Index of the atom to delete.

Returns`DataFrame` – The updated `dataframe` object.**df_to_xyz**()Returns the xyz geometry based on the `dataframe` attribute.**Returns**

`list[tuple[str, tuple[float, float, float]]]` – The xyz geometry.

dihedral_angle(*atom_ids*, *units='deg'*)Compute the dihedral angle defined by the four atoms specified in `atom_ids`.**Parameters**

- **atom_ids** (`list[int]`) – The four atom indices.
- **units** (`str`, default: "deg") – The units the angle is reported in, options are "deg" or "rad".

Returns

`float` – The dihedral angle defined by the four 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.

static from_xyz_string(*xyz_string*, *distance_units*='angstrom')

Create a `Geometry` object from a string.

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 the `dataframe` attribute.

Parameters

- **fn** (`str`) – The input filename.

Return type

`None`

load_json(*fn*)

Load a json file into the `dataframe` attribute of the `Geometry` object.

Parameters

- **fn** (`str`) – The input filename.

Return type

`None`

classmethod 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. Supported units are Angstrom ("angstrom") and Bohrs ("bohr").

Returns

`Geometry` – The loaded geometry.

load_zmatrix(*filename*, *distance_units*='angstrom')

Load a z-matrix from file into the `zmatrix` attribute.

Parameters

- **filename** (`str`) – Filename.
- **distance_units** (`str`, default: "angstrom") – Distance units; supported units are Angstrom ("angstrom") and Bohrs ("bohr").

Return type

`None`

`modify_bond_angle`(*atom_ids*, *theta*, *units*='deg')

Change the bond angle at the atom with the index that matches the second element of *atom_ids*.

The atom indexed by the third element of *atom_ids* has its position updated.

Parameters

- **`atom_ids`** (`list[int]`) – The indices of three atoms.
- **`theta`** (`float`) – New bond angle.
- **`units`** (`str`, default: "deg") – Units of the bond angle, options are "deg" or "rad".

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]`) – The indices of three atoms.
- **`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]`) – The indices of two atoms.
- **`new_bond_length`** (`float`) – The new 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` attribute.

`modify_dihedral_angle(atom_ids, theta, units='deg')`

Modify a dihedral angle.

Change the dihedral angle defined by the four atoms indexed by the elements of `atom_ids`. The atom indexed by the last element has its position updated.

Parameters

- `atom_ids` (`list[int]`) – The indices of the four atoms.
- `theta` (`float`) – The dihedral angle.
- `units` (`str`, default: "deg") – Units of the dihedral angle, supported units are "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]`) – The indices of four atoms.
- `theta` (`float`) – The dihedral angle.
- `group` (`str`) – Grouping label.
- `units` (`str`, default: "deg") – Units of the dihedral angle; supported units are "rad" or "deg".

Returns

`DataFrame` – The updated `dataframe` object.

`randomize_xyz(sigma=0.05, seed=0, freeze_atoms=None)`

Add random numbers to the xyz geometry.

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) – The 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 around one of the coordinate axes.

Parameters

- **theta** (`float`) – The angle through which to rotate.
- **axis** (`str`) – the coordinate axis to rotate around (upper or lower case).
- **units** (`str`, default: "deg") – units of the angle specified, options are "deg" or :code:`"rad".

Returns

`DataFrame` – The updated `dataframe` object.

`rotate_around_vector (vector, theta, units='deg')`

Rotate the geometry around a vector defined by input `theta`.

Parameters

- **vector** (`ndarray[Any, dtype[TypeVar(_ScalarType_co, 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" or "rad".

Returns

`DataFrame` – The updated `dataframe` object.

`save_csv (fn)`

Write the current `dataframe` object of the `Geometry` object to a csv file.

Parameters

- **fn** (`str`) – The output filename.

Return type

`None`

`save_json (fn)`

Write the current `self.df` property of the `Geometry` object to a json file.

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 output file containing the geometry.

Return type

`None`

`save_zmatrix (filename)`

Write the `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 that correspond 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 indices of the atoms between which the bond angle is changed.
- **`bond_angles`** (`list[float]`) – A list of bond angles.
- **`units`** (`str`, default: "deg")

Returns`list[Geometry]` – The `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]`) – The indices of three atoms.
- **`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 :code:bond_angles` in "deg" or "rad".

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 bond lengths.Create `Geometry` objects 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]` – The `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]`) – The indices of three atoms.
- **`bond_lengths`** (`list[float]`) – Bond lengths to create `Geometry` objects for.

- **group** (`str`) – Group heading.

Returns

`list[Geometry]` – The 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 dihedral angles.

Create `Geometry` objects 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** (`list[int]`) – The four atoms between which the dihedral is changed.
- **dihedral_angles** (`list[float]`) – The dihedral angles.
- **units** (`str`, default: "deg")

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** (`str`, default: "deg") – Units of the 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` attribute of the geometry. The source argument can be either a `dict`, 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 `dict`, the mapping argument does nothing. If source is a `string`, it must be an existing column heading in `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 `dataframe` attribute from Bohrs to Angstroms.

Returns

`DataFrame` – The updated `dataframe` object.

to_bohr()

Convert the geometry in the `dataframe` attribute from Angstroms to Bohrs.

Returns

`DataFrame` – The updated `dataframe` object.

to_zmatrix()

Map the geometry held in the `dataframe` attribute to a z-matrix.

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.

Returns

`str` – The updated `zmatrix` attribute.

translate_by_vector (*v*)

Translate all atoms in the geometry by a vector, *v*.

Parameters

- **v** (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Vector defining the translation.

Returns

`DataFrame` – The updated `dataframe` object.

property xyz: list[tuple[str, tuple[float, float, float]]]

Return the geometry in xyz format.

Returns

The geometry as a list of lists of atom symbols and positions

xyz_to_df (*xyz*)

Convert the xyz geometry to a `pandas.DataFrame`.

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.

property zmatrix: str

Return the geometry in z-matrix format.

Returns

The geometry as a z-matrix string.

zmatrix_to_df()

Convert the `zmatrix` attribute to a `pandas.DataFrame`.

Returns

`DataFrame` – The updated `dataframe` object geometry.

class GeometryPeriodic(geometric=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, DataFrame, list, None]`, default: `None`) – Geometrical information - can be a list of lists (e.g `[['H', [0, 0, 0]], ['H', [0, 0, 1]]]`), xyz coordinates as a string or `pandas.DataFrame`.
- **unit_cell** (`Optional[ndarray[Any, dtype[TypeVar(_ScalarType_co, 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.

Parameters

- **element** (`str`) – Element symbol.
- **position** (`Optional[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]]`, 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 to align the bond defined by the elements of the `atom_ids` list to a coordinate axis.

Align

Parameters

- **atom_ids** (`list`) – Two atom indices which define the bond to align to the axis.
- **axis** (`str`) – A cartesian co-ordinate axis, options are "X", "Y", "Z", "x",
:param "y": :param "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 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_co, bound= generic, covariant=True)]]`) – The vector of length three that we use to align the bond.

Returns

DataFrame – The updated `dataframe` object.

`align_to_plane`(*atom_ids*, *vectors*)

Rotate and translate the geometry such that three atoms lie in the plane defined by the vectors provided.

Parameters

- **`atom_ids`** (`list[int]`) – The indices of three atoms in the geometry that we use to align to the plane.
- **`vectors`** (`list[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]`) – An iterable containing three vectors that define the plane.

Returns

DataFrame – The updated `dataframe` object.

`align_to_xy_plane`(*atom_ids=None*)

Align any three atoms to the *xy* plane.

Rotate and translate the geometry such that the plane defined by the three atom indices in `atom_ids` becomes the *xy* plane.

Parameters

`atom_ids` (`Optional[list[int]]`, default: `None`) – The indices of three atoms in the geometry that we align to the *xy* 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 three atom indices in the `atom_ids` variable becomes the *xz* plane.

Parameters

`atom_ids` (`Optional[list[int]]`, default: `None`) – The indices of three atoms in the geometry that we align to the *xz* plane.

Returns

DataFrame – The updated `dataframe` object.

`align_to_yz_plane`(*atom_ids=None*)

Align any three atoms to the *yz* plane.

Rotate and translate the geometry such that the plane defined by the three atom indices in `atom_ids` becomes the *yz* plane.

Parameters

atom_ids (`Optional[list[int]]`, default: `None`) – The indices of three atoms in the geometry that we align to the yz plane.

Returns

`DataFrame` – The updated `dataframe` object.

property atomic_coordinates: ndarray[Any, dtype[_ScalarType_co]]

Get an array of position vectors of each atom in the geometry.

Each row corresponds to one position vector.

Returns

An array of floats where each row at index i contains the x, y, z coordinates of atom i .

bond_angle (`atom_ids, units='deg'`)

Compute the bond angle defined by the three atoms indexed by the elements of `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]`) – An iterable containing the indices of three atoms to use in calculating the angle.
- **units** (`str`, default: "deg") – Units of the reported bond angle, options are "deg" or "rad".

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]`) – An 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 of element A.
- **a** (`float`) – Distance between A and B in the AB pairs.

- **bb** (`str`) – Atomic symbol of element B.
- **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 of the element.
- **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.

Each *i, j* th element is the internuclear separation between atom *i* and *j*.**Returns**`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` –
The distance matrix.**`property dataframe: DataFrame`**Return the geometry in a `pandas.DataFrame` object.Each row corresponds to an atom in the geometry and each column holds the atomic symbol of an element and their *x, y, z* coordinates.

Returns

An pandas.DataFrame containing a geometry.

delete_atom(atom_index)

Delete an atom from the geometry.

Parameters

atom_index (Union[int, list]) – Index of the atom to delete.

Returns

DataFrame – The updated `dataframe` object.

df_to_xyz()

Returns the xyz geometry based on the `dataframe` attribute.

Returns

`list[tuple[str, tuple[float, float, float]]]` – The xyz geometry.

dihedral_angle(atom_ids, units='deg')

Compute the dihedral angle defined by the four atoms specified in atom_ids.

Parameters

- **atom_ids** (list[int]) – The four atom indices.
- **units** (str, default: "deg") – The units the angle is reported in, options are "deg" or "rad".

Returns

`float` – The dihedral angle defined by the four 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.

static from_xyz_string(xyz_string, distance_units='angstrom')

Create a Geometry object from a string.

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 the `dataframe` attribute.

Parameters

fn (str) – The input filename.

Return type

`None`

load_json(fn)

Load a json file into the `dataframe` attribute of the Geometry object.

Parameters

- fn** (`str`) – The input filename.

Return type

`None`

classmethod `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. Supported units are Angstrom ("angstrom") and Bohrs ("bohr").

Returns

`Geometry` – The loaded geometry.

modify_bond_angle (*atom_ids*, *theta*, *units='deg'*)

Change the bond angle at the atom with the index that matches the second element of *atom_ids*.

The atom indexed by the third element of *atom_ids* has its position updated.

Parameters

- **atom_ids** (`list[int]`) – The indices of three atoms.
- **theta** (`float`) – New bond angle.
- **units** (`str`, default: "deg") – Units of the bond angle, options are "deg" or "rad".

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]`) – The indices of three atoms.
- **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]`) – The indices of two atoms.

- **new_bond_length** (`float`) – The new 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` attribute.

modify_dihedral_angle (`atom_ids, theta, units='deg'`)

Modify a dihedral angle.

Change the dihedral angle defined by the four atoms indexed by the elements of `atom_ids`. The atom indexed by the last element has its position updated.

Parameters

- **atom_ids** (`list[int]`) – The indices of the four atoms.
- **theta** (`float`) – The dihedral angle.
- **units** (`str`, default: "deg") – Units of the dihedral angle, supported units are "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]`) – The indices of four atoms.
- **theta** (`float`) – The dihedral angle.
- **group** (`str`) – Grouping label.
- **units** (`str`, default: "deg") – Units of the dihedral angle; supported units are "rad" or "deg".

Returns

`DataFrame` – The updated `dataframe` object.

randomize_xyz(*sigma*=0.05, *seed*=0, *freeze_atoms*=None)

Add random numbers to the xyz geometry.

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`) – The 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 around one of the coordinate axes.

Parameters

- ***theta*** (`float`) – The angle through which to rotate.
- ***axis*** (`str`) – the coordinate axis to rotate around (upper or lower case).
- ***units*** (`str`, default: "deg") – units of the angle specified, options are "deg" or :code:`"rad".

Returns

`DataFrame` – The updated `dataframe` object.

rotate_around_vector(*vector*, *theta*, *units*='deg')

Rotate the geometry around a vector defined by input *theta*.

Parameters

- ***vector*** (`ndarray[Any, dtype[TypeVar(_ScalarType_co, 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" or "rad".

Returns

`DataFrame` – The updated `dataframe` object.

save_csv(*fn*)

Write the current `dataframe` object of the `Geometry` object to a csv file.

Parameters

- ***fn*** (`str`) – The output filename.

Return type

`None`

save_json (fn)

Write the current `self.df` property of the `Geometry` object to a json file.

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 output file containing the geometry.

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 that correspond 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 indices of the atoms between which the bond angle is changed.
- `bond_angles (list[float])` – A list of bond angles.
- `units (str, default: "deg")`

Returns

`list[Geometry]` – The `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])` – The indices of three atoms.
- `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 :code:bond_angles` in "deg" or "rad".

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 bond lengths.

Create `Geometry` objects 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]` – The 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]`) – The indices of three atoms.
- **bond_lengths** (`list[float]`) – Bond lengths to create Geometry objects for.
- **group** (`str`) – Group heading.

Returns

`list[Geometry]` – The 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 dihedral angles.

Create Geometry objects 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** (`list[int]`) – The four atoms between which the dihedral is changed.
- **dihedral_angles** (`list[float]`) – The dihedral angles.
- **units** (`str`, default: "deg")

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** (`str`, default: "deg") – Units of the 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` attribute of the geometry. The source argument can be either a `dict`, 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 `dict`, the mapping argument does nothing. If source is a `string`, it must be an existing column heading in `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 `dataframe` attribute from Bohrs to Angstroms.

Returns

`DataFrame` – The updated `dataframe` object.

`to_bohr()`

Convert the geometry in the `dataframe` attribute from Angstroms to Bohrs.

Returns

`DataFrame` – The updated `dataframe` object.

`translate_by_vector(v)`

Translate all atoms in the geometry by a vector, v.

Parameters

- `v` (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Vector defining the translation.

Returns

`DataFrame` – The updated `dataframe` object.

`property xyz: list[tuple[str, tuple[float, float, float]]]`

Return the geometry in xyz format.

Returns

The geometry as a list of lists of atom symbols and positions

`xyz_to_df(xyz)`

Convert the xyz geometry to a `pandas.DataFrame`.

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.

27.8 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 detailed 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"]`.

```
abstract classmethod flip_set(cls, qubit, qubits)
```

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]`) – A list of qubits containing the register to be mapped to or an integer 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

`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.
- Integral operator classes, such as `ChemistryRestrictedIntegralOperator` 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], SymmetryOperatorFermionicFactorised]`) – 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 integer 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, QubitOperatorString, QubitOperatorList, list, ndarray, SymmetryOperatorPauliFactorised]` – The mapped operator or set of operators in the output format described above.

abstract 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`) – A list of qubits containing the register to be mapped to or an integer 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

`list[Qubit]` – List of Qubits comprising the parity set for orbital i.

abstract classmethod rho_set(`cls, qubit, qubits`)

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]`) – A list of qubits containing the register to be mapped to or an integer 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

`list[Qubit]` – List of Qubits comprising the rho set for orbital i.

classmethod state_map(`state, qubits=None`)

Maps a fermionic state to a qubit state.

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

Generally, implementations of this method use a state mapping scheme given by literature convention – 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 integer 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 is an alias of `state_map()` - see the documentation of this method for further details.

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 integer 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, list, ndarray, spmatrix]` – The mapped state.

abstract 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` – A matrix mapping the binary representation of a fermionic basis state index to the binary representation of the corresponding qubit basis state index.

abstract classmethod `update_set`(*cls*, *qubit*, *qubits*)

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]`) – A list of qubits containing the register to be mapped to or an integer 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

`list[Qubit]` – List of Qubits comprising the update set for orbital i.

class `QubitMappingJordanWigner`

Bases: `QubitMapping`

Maps 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`) – A list of qubits containing the register to be mapped to or an integer 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

`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.
- Integral operator classes, such as `ChemistryRestrictedIntegralOperator` 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], SymmetryOperatorFermionicFactorised]`) – 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 integer 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, QubitOperatorString, QubitOperatorList, list, ndarray, SymmetryOperatorPauliFactorised]` – 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`) – A list of qubits containing the register to be mapped to or an integer 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

`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`) – A list of qubits containing the register to be mapped to or an integer 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

`list[Qubit]` – List of Qubits comprising the rho set for orbital i.

classmethod state_map (`state, qubits=None`)

Maps a fermionic state to a qubit state.

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

Generally, implementations of this method use a state mapping scheme given by literature convention – 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 integer 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 is an alias of `state_map()` - see the documentation of this method for further details.

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 integer 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, list, ndarray, spmatrix]` – The mapped state.

`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[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – A matrix mapping the binary representation of a fermionic basis state index to the binary representation of the corresponding qubit basis state index.

```
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`) – A list of qubits containing the register to be mapped to or an integer 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

`list[Qubit]` – List of Qubits comprising the update set for orbital i.

```
class QubitMappingBravyiKitaev
```

Bases: `QubitMapping`

Maps 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`) – A list of qubits containing the register to be mapped to or an integer 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

`list[Qubit]` – List of Qubits comprising the flip set for orbital i.

```
classmethod operator_map(cls, operator, qubits, 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.
- Integral operator classes, such as *ChemistryRestrictedIntegralOperator* 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, BaseChemistryIntegralOperator, list, ndarray]`) – An object representing a fermionic operator or set of operators, with type behaviour specified above.
- **qubits** (`Union[list[Qubit], int]`) – A list of qubits containing the register to be mapped to or an integer 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, QubitOperatorString, 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`) – A list of qubits containing the register to be mapped to or an integer 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

`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`) – A list of qubits containing the register to be mapped to or an integer 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

`list[Qubit]` – List of Qubits comprising the rho set for orbital i.

classmethod state_map (`state, qubits=None`)

Maps a fermionic state to a qubit state.

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

Generally, implementations of this method use a state mapping scheme given by literature convention – 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 integer 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 is an alias of `state_map()` - see the documentation of this method for further details.

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 integer 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, list, ndarray, spmatrix]` – The mapped state.

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` – A matrix mapping the binary representation of a fermionic basis state index to the binary representation of the corresponding qubit basis state index.

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`) – A list of qubits containing the register to be mapped to or an integer 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

`list[Qubit]` – List of Qubits comprising the update set for orbital i.

class `QubitMappingParity`

Bases: `QubitMapping`

Maps states and operators in a Fermionic space to states and operators in a qubit space using the parity mapping.

The `parity mapping` uses qubits to store the parity of sums of fermionic occupation numbers, thereby reducing the number of operations required to enforce anticommutation relations (but commensurately increasing the number of operations required to update occupation numbers).

`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`) – A list of qubits containing the register to be mapped to or an integer 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

`list[Qubit]` – List of Qubits comprising the flip set for orbital i.

`classmethod operator_map(cls, operator, qubits, 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.
- Integral operator classes, such as `ChemistryRestrictedIntegralOperator` 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, BaseChemistryIntegralOperator, list, ndarray]`) – An object representing a fermionic operator or set of operators, with type behaviour specified above.

- **qubits** (`Union[list[Qubit], int]`) – A list of qubits containing the register to be mapped to or an integer 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, QubitOperatorString, 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`) – A list of qubits containing the register to be mapped to or an integer 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

`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`) – A list of qubits containing the register to be mapped to or an integer 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

`list[Qubit]` – List of Qubits comprising the rho set for orbital i.

classmethod state_map (`state, qubits=None`)

Maps a fermionic state to a qubit state.

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**

Generally, implementations of this method use a state mapping scheme given by literature convention – 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 integer 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 is an alias of `state_map()` - see the documentation of this method for further details.

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 integer 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, list, ndarray, spmatrix]` – The mapped state.

`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` – A matrix mapping the binary representation of a fermionic basis state index to the binary representation of the corresponding qubit basis state index.

```
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`) – A list of qubits containing the register to be mapped to or an integer 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

`list[Qubit]` – List of Qubits comprising the update set for orbital i.

```
class QubitMappingParaparticular
```

Bases: `QubitMapping`

Maps fermions to qubits a paraparticle 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 anti-commutation 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](https://arxiv.org/abs/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`) – A list of qubits containing the register to be mapped to or an integer 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

`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.
- Integral operator classes, such as `ChemistryRestrictedIntegralOperator` 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], SymmetryOperatorFermionicFactorised]`) – 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 integer 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, QubitOperatorString, QubitOperatorList, list, ndarray, SymmetryOperatorPauliFactorised]` – 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`) – A list of qubits containing the register to be mapped to or an integer 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

`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`) – A list of qubits containing the register to be mapped to or an integer 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

`list[Qubit]` – List of Qubits comprising the rho set for orbital i.

classmethod state_map(state, qubits=None)

Maps a fermionic state to a qubit state.

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

Generally, implementations of this method use a state mapping scheme given by literature convention – 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 integer 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 is an alias of `state_map()` - see the documentation of this method for further details.

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 integer 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, list, ndarray, spmatrix]` – The mapped state.

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[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – A matrix mapping the binary representation of a fermionic basis state index to the binary representation of the corresponding qubit basis state index.

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`) – A list of qubits containing the register to be mapped to or an integer 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

`list[Qubit]` – List of Qubits comprising the update set for orbital i.

27.9 inquanto.minimizers

Module provides access to minimizers compatible with variational experiments.

```
class MinimizerRotosolve(max_iterations=20, tolerance=1e-4, disp=False, order_independence=True)
```

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.
- `order_independence` (`bool`, default: `True`) – If `False`, the minimizer depends on the order of parameters. If `True`, the minimizer operates independently of parameter order.

```
generate_report()
```

Generates a summary of the minimization.

Returns

`dict` – A dictionary 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 parameters provided by the initial argument.

Parameters

- `function` (`Callable[[Union[ndarray, list[float]]], float]`) – Sinusoidal objective function to minimize.
- `initial` (`Union[ndarray, list[float]]`) – Initial parameters to optimize.

Returns

`tuple[float, ndarray]` – A tuple containing the final value and parameters obtained by the minimization.

Raises

`RuntimeError` – If the optimizer does not converge within the maximum number of iterations.

```
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, displays minimization history.
- **callback** (`Optional[Callable]`, default: None) – Custom callback for minimizer.

generate_report ()

Generates a report summarizing the minimization.

Includes the final value, the final parameters and the number of iterations performed.

Returns

`dict` – 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 MinimizerSPSA (max_iterations=int(1e5), tolerance=1e-4, disp=False)

Bases: GeneralMinimizer

The Simultaneous Perturbation Stochastic Approximation (SPSA) minimizer.

Implementation details are based on https://www.jhuapl.edu/spsa/PDF-SPSA/Spall_Implementation_of_the_Simultaneous.PDF

Parameters

- **max_iterations** (`int`, default: `int(1e5)`) – Maximum number of iterations allowed before the minimization is terminated.
- **disp** (`bool`, default: False) – If True, print information to the screen throughout minimization.
- **tolerance** (`float`, default: `1e-4`) – Tolerance for convergence.
- **disp** – If True, print information to the screen throughout minimization.

generate_report ()

Generate a report summarizing the minimization.

Includes the final value and the final parameters performed.

Returns

`dict` – A dictionary containing the final value and parameters obtained by the minimizer.

```
minimize(function, initial, alpha=0.602, gamma=0.101, a=0.5, c=0.2, stability_constant=None,
perturbation_samples=1, perturbation_samples_init=None, gradient_smoothing=False)
```

Minimize the function provided.

Minimization starts at the parameters provided by the initial argument.

Parameters

- **function** (`Callable[[Union[ndarray, list[float]]], float]`) – Objective function to minimize.
- **initial** (`Union[ndarray, list[float]]`) – Initial parameters to minimize.
- **alpha** (`float`, default: 0.602) – The exponent of the learning rate powerseries.
- **gamma** (`float`, default: 0.101) – The exponent of the perturbation powerseries.
- **a** (`float`, default: 0.5) – The numerator of the initial learning rate magnitude.
- **c** (`float`, default: 0.2) – The initial perturbation magnitude.
- **stability_constant** (`Optional[float]`, default: None) – The denominator of the initial learning rate magnitude.
- **perturbation_samples** (`int`, default: 1) – The number of perturbation samples used for gradient approximation.
- **perturbation_samples_init** (`Optional[int]`, default: None) – The number of perturbation samples used for the initial gradient approximation. If `None`, this will be the same as `perturbation_samples`.
- **gradient_smoothing** (`bool`, default: `False`) – If `True`, the gradient approximation is based on previous approximations.

Returns

`tuple[float, Union[ndarray, list[float]]]` – The final value and parameters obtained by the minimization.

```
class MinimizerScipy(method=OptimizationMethod.L_BFGS_B_smooth, options=None, disp=False,
callback=None)
```

Bases: `GeneralMinimizer`

A simple wrapper for SciPy minimization routines.

More minimizer details can be found in the [SciPy documentation](#).

Parameters

- **method** (`OptimizationMethod` | `str`, default: `OptimizationMethod.L_BFGS_B_smooth`) – The method to use. Popular methods to choose from include "CG", "BFGS", "SLSQP", and "COBYLA". For the L-BFGS-B method, the "OptimizationMethod" enum is used to conveniently specify the optimization method along with its associated default parameters.
- **options** (`Optional[dict]`, default: `None`) – Options used for calibration which are passed through to the SciPy minimization.
- **argument.** (*This overrides any default settings provided by the method*)
- **disp** (`bool`, default: `False`) – If `True`, prints minimization history.
- **callback** (`Optional[Callable]`, default: `None`) – Custom callback function.

generate_report ()

Generates 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.

Raises

`ValueError` – If no result is available.

property method: str

Get the method being used by the optimizer as a string.

Returns

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 minimize.
- `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.

Raises

`ValueError` – If optimization process fails.

property options: dict

Get the options passed to SciPy by the minimizer internally.

Returns

A dictionary 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_co, bound= generic, covariant=True)]]`) – A monotonically increasing or decreasing sequence of time points at which derivatives are evaluated.
- `disp` (`bool`, default: `False`) – If `True`, print information to the screen throughout minimization.
- `callback` (`Optional[Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], float, ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], Any]`, default:

`None`) – An optional function $f(p, t, x)$, where p are the parameters of the differential equation, t is the time at which the derivatives are evaluated and x are the derivatives.

- **linear_solver** (`Optional[Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]`, default: `GeneralIntegrator.linear_solver_scipy_linalg`) – An optional solver for the derivative at time t .

```
static linear_solver_scipy_linalg(a, b)
```

A wrapper for the `scipy.linalg.solve()` method.

Solves the linear equation $a @ x == b$ for the unknown x for the square a matrix. More information can be found in the [SciPy documentation](#).

Parameters

- **a** (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Square shaped matrix.
- **b** (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Input for the right hand side of the equation.

Returns

`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – An ndarray of the solution.

```
static linear_solver_scipy_pinvh(a, b)
```

Linear equation solver using `scipy.linalg.pinvh()`.

Solves the linear equation $a @ x == b$ for the unknown x using the (Moore-Penrose) pseudo-inverse of a Hermitian matrix, a . More information on `scipy.linalg.pinvh()` can be found in the [SciPy documentation](#).

Parameters

- **a** (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Hermitian matrix to be inverted.
- **b** (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Input for the right hand side of the equation.

Returns

`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – An ndarray of the solution.

```
solve(linear_problem, initial, *args, **kwargs)
```

Solve the differential equation.

Parameters

- **linear_problem** (`Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]], float], tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]]`) – A function $f(p, t) \mapsto A, b$ which takes the parameters and time and returns the linear problem (matrix $A(t)$, vector $b(t)$ of $A * x = b$) at a time t .
- **initial** (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Initial parameters.

- **args** ([Any](#))
- **kwargs** ([Any](#))

Returns

```
ndarray[Any, dtype[TypeVar\(\_ScalarType\_co, bound= generic, covariant=True\)\]\] :-
```

Array containing the solution of the differential equation for each time in `time_eval`, with the initial value in the first row.

```
class OptimizationMethod(value, names=None, *, module=None, qualname=None, type=None, start=1,
                        boundary=None)
```

Bases: [Enum](#)

Enumeration of optimization methods with associated parameters.

Each enum member represents an optimization method along with a dictionary of parameters used in the optimization process. These settings are recommended based on empirical testing for optimal performance.

L_BFGS_B_smooth

method: "L-BFGS-B". Applicability: This method is more suitable for smoother and more fine-grained optimization. ftol: Tolerance for termination. The iteration stops when

$$\frac{f^k - f^{k+1}}{\max(|f^k|, |f^{k+1}|)} \leq \text{ftol}$$

$$\text{eps}$$
: Absolute step size used for numerical approximation of the Jacobian via forward differences.

L_BFGS_B_coarse

method: "L-BFGS-B". Applicability: This method is generally more suitable for coarser and more rapid optimization and may be preferable for optimizing noisy objective functions. ftol: Tolerance for termination. The iteration stops when

$$\frac{f^k - f^{k+1}}{\max(|f^k|, |f^{k+1}|)} \leq \text{ftol}$$

$$\text{eps}$$
: Absolute step size used for numerical approximation of the Jacobian via forward differences.

```
L_BFGS_B_coarse = ('L-BFGS-B', {'eps': 0.1, 'ftol': 0.0001})
```

```
L_BFGS_B_smooth = ('L-BFGS-B', {'eps': 1e-08, 'ftol': 2.2e-09})
```

```
class ScipyIVPIntegrator(time_eval, disp=False, callback=None,
                           linear_solver=GeneralIntegrator.linear_solver_scipy_linalg)
```

Bases: [GeneralIntegrator](#)

A simple wrapper for SciPy `solve_ivp()` for linear problems.

More details about `solve_ivp()` can be found in the [SciPy documentation](#).

Parameters

- **time_eval** (ndarray[[Any](#), dtype[[TypeVar\(_ScalarType_co, bound= generic, covariant=True\)\]\]\]\) – A monotonically increasing or decreasing sequence of time points at which derivatives are evaluated.](#)
- **disp** ([bool](#), default: `False`) – If `True`, print information to the screen throughout minimization.
- **callback** ([Optional\[Callable\[\[ndarray\[Any\], float, ndarray\[Any\]\], Any\]\]](#), default: `None`) – An optional function $f(p, t, x)$, where p are the parameters of the differential equation, t is the time at which the derivatives are evaluated and x are the derivatives.

- **linear_solver** (`Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]`) – default: `GeneralIntegrator`.
`linear_solver_scipy_linalg`) – An optional solver for the derivative at time t .

static linear_solver_scipy_linalg(a, b)

A wrapper for the `scipy.linalg.solve()` method.

Solves the linear equation $a @ x == b$ for the unknown x for the square a matrix. More information can be found in the [SciPy documentation](#).

Parameters

- **a** (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Square shaped matrix.
- **b** (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Input for the right hand side of the equation.

Returns

`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – An `ndarray` of the solution.

static linear_solver_scipy_pinvh(a, b)

Linear equation solver using `scipy.linalg.pinvh()`.

Solves the linear equation $a @ x == b$ for the unknown x using the (Moore-Penrose) pseudo-inverse of a Hermitian matrix, a . More information on `scipy.linalg.pinvh()` can be found in the [SciPy documentation](#).

Parameters

- **a** (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Hermitian matrix to be inverted.
- **b** (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Input for the right hand side of the equation.

Returns

`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – An `ndarray` of the solution.

solve(linear_problem, initial, *args, **kwargs)

Solve the differential equation.

Parameters

- **linear_problem** (`Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]], float], tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]]`) – A function $f(p, t) \mapsto A, b$ which takes the parameters and time and returns the linear problem (matrix $A(t)$, vector $b(t)$) of $A * x = b$ at a time t .
- **initial** (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Initial parameters.
- **args** (`Any`)
- **kwargs** (`Any`)

Returns

`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` –
Array containing the solution of the differential equation for each time in `time_eval`, with the initial value in the first row.

```
class ScipyODEIntegrator(time_eval, disp=False, callback=None,
                           linear_solver=GeneralIntegrator.linear_solver_scipy_linalg)
```

Bases: GeneralIntegrator

A simple wrapper for SciPy `odeint()` for linear problems.

More details about `odeint()` can be found in the [SciPy documentation](#).

Parameters

- **time_eval** (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – A monotonically increasing or decreasing sequence of time points at which derivatives are evaluated.
- **disp** (`bool`, default: `False`) – If True, print information to the screen throughout minimization.
- **callback** (`Optional[Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], float, ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], Any]]`, default: `None`) – An optional function $f(p, t, x)$, where p are the parameters of the differential equation, t is the time at which the derivatives are evaluated and x are the derivatives.
- **linear_solver** (`Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]`, default: `GeneralIntegrator.linear_solver_scipy_linalg`) – An optional solver for the derivative at time t .

```
static linear_solver_scipy_linalg(a, b)
```

A wrapper for the `scipy.linalg.solve()` method.

Solves the linear equation $a @ x == b$ for the unknown x for the square a matrix. More information can be found in the [SciPy documentation](#).

Parameters

- **a** (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Square shaped matrix.
- **b** (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Input for the right hand side of the equation.

Returns

`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` –
An `ndarray` of the solution.

```
static linear_solver_scipy_pinvh(a, b)
```

Linear equation solver using `scipy.linalg.pinvh()`.

Solves the linear equation $a @ x == b$ for the unknown x using the (Moore-Penrose) pseudo-inverse of a Hermitian matrix, a . More information on `scipy.linalg.pinvh()` can be found in the [SciPy documentation](#).

Parameters

- **a** (ndarray[`Any`, dtype[`TypeVar(_ScalarType_co`, bound=`generic`, covariant=`True`)]]) – Hermitian matrix to be inverted.
- **b** (ndarray[`Any`, dtype[`TypeVar(_ScalarType_co`, bound=`generic`, covariant=`True`)]]) – Input for the right hand side of the equation.

Returns

ndarray[`Any`, dtype[`TypeVar(_ScalarType_co`, bound=`generic`, covariant=`True`)]] – An ndarray of the solution.

solve (*linear_problem*, *initial*, **args*, ***kwargs*)

Solve the differential equation.

Parameters

- **linear_problem** (`Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]], float], tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]]]) – A function $f(p, t) \mapsto A, b$ which takes the parameters and time and returns the`
- **problem** (*linear*)
- **initial** (ndarray[`Any`, dtype[`TypeVar(_ScalarType_co`, bound=`generic`, covariant=`True`)]]) – Initial parameters.
- **args** (`Any`)
- **kwargs** (`Any`)

Returns

ndarray[`Any`, dtype[`TypeVar(_ScalarType_co`, bound=`generic`, covariant=`True`)]] – Array containing the solution to the problem for each time in `time_eval`, with the initial value in the first row.

27.10 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 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** (`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 one, 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*= -1.0 , *tol2*=*None*, *method*=`DecompositionMethod.EIG`, *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.

The Hamiltonian can be written as $\hat{H} = H_0 + \hat{H}_1 + \hat{S} + \hat{V}$ where $\hat{S} + \hat{V}$ is the Coulomb interaction. $V = \frac{1}{2} \sum_{ijkl} (ij|kl) a_i^\dagger a_j a_k^\dagger a_l$ is a reordered two-body operator and \hat{S} is a one-body offset term given by $\hat{S} = \sum_{ij} s_{ij} a_i^\dagger a_j$ where $s_{ij} = -\frac{1}{2} \sum_k (ik|kj)$. H_0 and \hat{H}_1 are the constant and one-electron terms respectively.

The first level of factorization decomposes the electron repulsion integrals tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or a pivoted, incomplete Cholesky decomposition (`method="cho"`) (see [J. Chem. Phys. 118, 9481–9484 \(2003\)](#) and [J. Chem. Phys. 139, 134105 \(2013\)](#)). The second factorization is diagonalization of the V_{pq}^t matrix for each t: $V_{pq}^t = \sum_u^{N_\lambda} U_{pu}^t \lambda_u^t U_{qu}^t$.

At the first factorization stage, truncation depends on the decomposition method. With "`eig`", truncation is performed by discarding eigenvalues, starting from the smallest in magnitude, until the sum of those discarded

magnitudes exceeds the threshold `tol1`. With "cho", the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding low-magnitude eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr}\omega_r W_{qr}$. One-body diagonalization is not truncated.

Warning

This is 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: `-1.0`) – 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.
- `method` (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`)
 - Decomposition method used for the first level of factorization. "eig" for an eigenvalue decomposition, "cho" for a pivoted, incomplete Cholesky decomposition.
- `diagonalize_one_body` (`bool`, default: `True`) – Whether to diagonalize the physical one-body integrals h_{pq} .
- `diagonalize_one_body_offset` (`bool`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq} .
- `combine_one_body_terms` (`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, diagonalized one-body integrals.

`property dtype: dtype`

Returns numpy data type of the two-body integral terms.

`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.

This is defined as the contribution to the effective potential matrix from the difference of the alpha and beta 1-RDMs .

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]` – The mean-field electronic energy (one-body + two-body), and the mean-field Coulomb contribution (two-body only).

classmethod from_FermionOperator(*operator*, *dtype=float*)

Convert a two-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 fermionic operator.
- `dtype` (default: `float`) – Desired term value `dtype`.

Returns

`ChemistryRestrictedIntegralOperator` – The converted operator.

⚠ Warning

This converter assumes that `operator` represents a spin-restricted operator.

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

`Tuple[ChemistryRestrictedIntegralOperator, int, int, int]` – A tuple containing the operator generated from the FCIDUMP file, the number of orbitals in the system, the number of electrons in the system, and the spin multiplicity of the system.

property imag: ChemistryRestrictedIntegralOperator

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 (*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 this instance.

print_table ()

Prints operator terms in a table format.

Return type

`None`

qubit_encode (mapping=None, qubits=None)

Performs qubit encoding (mapping), using provided mapping class, of this instance.

This proceeds by creation of a `FermionOperator` class, via the `to_FermionOperator()` method, before mapping to a `QubitOperator` using the provided mapping scheme. Please see the documentation for `to_FermionOperator()` for finer grained control over operator generation.

Parameters

- **mapping** (`Optional[QubitMapping]`, default: `None`) – `QubitMapping`-derived instance. Default mapping procedure is `QubitMappingJordanWigner`.

- **qubits** (`Optional[List[Qubit]]`, default: `None`) – The qubit register. If left as `None`, a default register will be assumed if possible. See `QubitMapping` documentation for further details.

Returns

`QubitOperator` – Mapped `QubitOperator` object.

property real: ChemistryRestrictedIntegralOperator

Extract the real part of the integral operator.

Returns

New integral operator object with only the real parts of all elements remaining.

rotate (rotation, check_unitary=True, check_unitary_atol=1e-15)

Performs an in-place unitary rotation of the chemistry integrals.

Parameters

- **rotation** (`ndarray`) – Unitary rotation matrix.
- **check_unitary** (`bool`, default: `True`) – Whether to perform the check for unitarity of the rotation matrix.
- **check_unitary_atol** (`float`, default: `1e-15`) – Absolute tolerance of unitarity check.

Returns

`ChemistryRestrictedIntegralOperator` – This instance 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`.

Parameters

- **yield_constant** (`bool`, default: `True`) – Whether to include a constant term.
- **yield_one_body** (`bool`, default: `True`) – Whether to include one-body terms.
- **yield_two_body** (`bool`, default: `True`) – Whether to include two-body terms.

Returns

`FermionOperator` – Integral operator in general fermionic operator form.

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

`ChemistryRestrictedIntegralOperatorCompact` – Equivalent integral operator with two-body integrals stored in compact form.

`two_body_iijj()`

Returns pair-diagonal elements of two-body integrals, ($ii|jj$) in chemist 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] = (pq|rs)`. Two-body integrals are stored as a `CompactTwoBodyIntegralsS4` or `CompactTwoBodyIntegralsS8` 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=-1.0, tol2=None, method=DecompositionMethod.EIG, 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.

The Hamiltonian can be written as $\hat{H} = H_0 + \hat{H}_1 + \hat{S} + \hat{V}$ where $\hat{S} + \hat{V}$ is the Coulomb interaction. $V = \frac{1}{2} \sum_{ijkl} (ij|kl) a_i^\dagger a_j a_k^\dagger a_l$ is a reordered two-body operator and \hat{S} is a one-body offset term given by $\hat{S} = \sum_{ij} s_{ij} a_i^\dagger a_j$ where $s_{ij} = -\frac{1}{2} \sum_k (ik|kj)$. H_0 and \hat{H}_1 are the constant and one-electron terms respectively.

The first level of factorization decomposes the electron repulsion integrals tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or a pivoted, incomplete Cholesky decomposition (`method="cho"`) (see [J. Chem. Phys. 118, 9481–9484 \(2003\)](#) and [J. Chem. Phys. 139, 134105 \(2013\)](#)). The second factorization is diagonalization of the V_{pq}^t matrix for each t: $V_{pq}^t = \sum_u^{N_\lambda} U_{pu}^t \lambda_u^t U_{qu}^t$.

At the first factorization stage, truncation depends on the decomposition method. With "eig", truncation is performed by discarding eigenvalues, starting from the smallest in magnitude, until the sum of those discarded magnitudes exceeds the threshold `tol1`. With "cho", the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding low-magnitude eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. One-body diagonalization is not truncated.

 **Warning**

This is 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: `-1.0`) – 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.
- `method` (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`)
 - Decomposition method used for the first level of factorization. "eig" for an eigenvalue decomposition, "cho" for a pivoted, incomplete Cholesky decomposition.
- `diagonalize_one_body` (`bool`, default: `True`) – Whether to diagonalize the physical one-body integrals h_{pq} .
- `diagonalize_one_body_offset` (`bool`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq}

- `combine_one_body_terms` (`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, diagonalized one-body integrals.

property dtype: dtype

Returns numpy data type of the two-body integral terms.

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.

This is defined as the contribution to the effective potential matrix from the difference of the alpha and beta 1-RDMs.

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]` – The mean-field electronic energy (one-body + two-body), and the mean-field Coulomb contribution (two-body only).

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** (default: `float`) – Desired term value `dtype`.
- **symmetry** (`Union[str, int]`, default: "s4") – Target symmetry. Four-fold symmetry (4, "4" or "s4") or eight-fold symmetry (8, "8" or "s8") are supported.

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.

`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, qubits=None*)

Performs qubit encoding (mapping), using provided mapping class, of this instance.

This proceeds by creation of a `FermionOperator` class, via the `to_FermionOperator()` method, before mapping to a `QubitOperator` using the provided mapping scheme. Please see the documentation for `to_FermionOperator()` for finer grained control over operator generation.

Parameters

- `mapping` (`Optional[QubitMapping]`, default: `None`) – `QubitMapping`-derived instance. Default mapping procedure is `QubitMappingJordanWigner`.
- `qubits` (`Optional[List[Qubit]]`, default: `None`) – The qubit register. If left as `None`, a default register will be assumed if possible. See `QubitMapping` documentation for further details.

Returns`QubitOperator` – 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, check_unitary=True, check_unitary_atol=1e-15*)

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.
- `check_unitary` (`bool`, default: `True`) – Whether to perform the check for unitarity of the rotation matrix.
- `check_unitary_atol` (`float`, default: `1e-15`) – Absolute tolerance of unitarity check.

Returns`ChemistryRestrictedIntegralOperatorCompact` – This instance 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`.

Parameters

- `yield_constant` (`bool`, default: `True`) – Whether to include a constant term.

- **`yield_one_body`** (`bool`, default: `True`) – Whether to include one-body terms.
- **`yield_two_body`** (`bool`, default: `True`) – Whether to include two-body terms.

Returns

`FermionOperator` – Integral operator in general fermionic operator form.

`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 chemist 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 integrals for the alpha (a) spin channel.
- **`one_body_bb`** (`ndarray`) – One-body 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 one, 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=-1.0, tol2=None, method=DecompositionMethod.EIG, 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.

The Hamiltonian can be written as $\hat{H} = H_0 + \hat{H}_1 + \hat{S} + \hat{V}$ where $\hat{S} + \hat{V}$ is the Coulomb interaction. $V = \frac{1}{2} \sum_{ijkl} (ij|kl) a_i^\dagger a_j a_k^\dagger a_l$ is a reordered two-body operator and \hat{S} is a one-body offset term given by $\hat{S} = \sum_{ij} s_{ij} a_i^\dagger a_j$ where $s_{ij} = -\frac{1}{2} \sum_k (ik|kj)$. H_0 and \hat{H}_1 are the constant and one-electron terms respectively.

The first level of factorization decomposes the electron repulsion integrals tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or a pivoted, incomplete Cholesky decomposition (`method='cho'`) (see [J. Chem. Phys. 118, 9481–9484 \(2003\)](#) and [J. Chem. Phys. 139, 134105 \(2013\)](#)). 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 the first factorization stage, truncation depends on the decomposition method. With "`eig`", truncation is performed by discarding eigenvalues, starting from the smallest in magnitude, until the sum of those discarded magnitudes exceeds the threshold `tol1`. With "`cho`", the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding low-magnitude eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. One-body diagonalization is not truncated.

 **Warning**

This is 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: `-1.0`) – 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.

- **method** (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`) – Decomposition method used for the first level of factorization. "eig" for an eigenvalue decomposition, "cho" for a pivoted, incomplete Cholesky decomposition.
- **diagonalize_one_body** (`bool`, default: `True`) – Whether to diagonalize the physical one-body integrals h_{pq} .
- **diagonalize_one_body_offset** (`bool`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq} .
- **combine_one_body_terms** (`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, diagonalized 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]` – The mean-field electronic energy (one-body + two-body), and the mean-field Coulomb contribution (two-body only).

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 – The converted operator.

⚠ Warning

The same-spin (aaaa and bbbb) blocks returned are contracted - this is fine for use in quantum chemistry calculations with density matrices having unit elements (e.g. 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*, *qubits=None*)

Performs qubit encoding (mapping), using provided mapping class, of this instance.

This proceeds by creation of a *FermionOperator* class, via the `to_FermionOperator()` method, before mapping to a *QubitOperator* using the provided mapping scheme. Please see the documentation for `to_FermionOperator()` for finer grained control over operator generation.

Parameters

- **mapping** (`Optional[QubitMapping]`, default: `None`) – *QubitMapping*-derived instance. Default mapping procedure is *QubitMappingJordanWigner*.
- **qubits** (`Optional[List[Qubit]]`, default: `None`) – The qubit register. If left as `None`, a default register will be assumed if possible. See *QubitMapping* documentation for further details.

Returns

QubitOperator – Mapped *QubitOperator* object.

rotate (*rotation_aa*, *rotation_bb*, *check_unitary=True*, *check_unitary_atol=1e-15*)

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.
- **check_unitary** (bool, default: True) – Whether to perform the check for unitarity of the rotation matrices.
- **check_unitary_atol** (float, default: 1e-15) – Absolute tolerance of unitarity checks.

Returns

ChemistryUnrestrictedIntegralOperator – This instance 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*.

Parameters

- **yield_constant** (bool, default: True) – Whether to include a constant term.
- **yield_one_body** (bool, default: True) – Whether to include one-body terms.
- **yield_two_body** (bool, default: True) – Whether to include two-body terms.

Returns

FermionOperator – Integral operator in general fermionic operator form.

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] = (pq|rs)`. Two-body integrals are stored as a `CompactTwoBodyIntegralsS4` or `CompactTwoBodyIntegralsS8` 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 `bbba` 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_bbba` (`CompactTwoBodyIntegralsS4`) – ERI with spatial orbitals in the `bbba` 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=-1.0, tol2=None, method=DecompositionMethod.EIG, 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.

The Hamiltonian can be written as $\hat{H} = H_0 + \hat{H}_1 + \hat{S} + \hat{V}$ where $\hat{S} + \hat{V}$ is the Coulomb interaction. $V = \frac{1}{2} \sum_{ijkl} (ij|kl) a_i^\dagger a_j a_k^\dagger a_l$ is a reordered two-body operator and \hat{S} is a one-body offset term given by $\hat{S} = \sum_{ij} s_{ij} a_i^\dagger a_j$ where $s_{ij} = -\frac{1}{2} \sum_k (ik|kj)$. H_0 and \hat{H}_1 are the constant and one-electron terms respectively.

The first level of factorization decomposes the electron repulsion integrals tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or a pivoted, incomplete Cholesky decomposition (`method="cho"`) (see [J. Chem. Phys. 118, 9481–9484 \(2003\)](#) and [J. Chem. Phys. 139, 134105 \(2013\)](#)). The second factorization is diagonalization of the V_{pq}^t matrix for each t: $V_{pq}^t = \sum_u^{N_\lambda} U_{pu}^t \lambda_u^t U_{qu}^t$.

At the first factorization stage, truncation depends on the decomposition method. With "eig", truncation is performed by discarding eigenvalues, starting from the smallest in magnitude, until the sum of those discarded magnitudes exceeds the threshold `tol1`. With "cho", the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding low-magnitude eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. One-body diagonalization is not truncated.

 **Warning**

This is 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: `-1.0`) – 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.
- `method` (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`)
 - Decomposition method used for the first level of factorization. "eig" for an eigenvalue decomposition, "cho" for a pivoted, incomplete Cholesky decomposition.
- `diagonalize_one_body` (`bool`, default: `True`) – Whether to diagonalize the physical one-body integrals h_{pq} .
- `diagonalize_one_body_offset` (`bool`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq}

- `combine_one_body_terms` (`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, diagonalized 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]` – The mean-field electronic energy (one-body + two-body), and the mean-field Coulomb contribution (two-body only).

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.
- `symmetry` (`Union[str, int]`, default: `"s4"`) – Target symmetry. Four-fold symmetry (4, "4" or "s4") or eight-fold symmetry (8, "8" or "s8") are supported.

Returns

`ChemistryUnrestrictedIntegralOperatorCompact` – Output integral operator.

Warning

The same-spin (aaaa and bbbb) blocks returned are contracted - this is fine for use in quantum chemistry calculations with density matrices having unit elements (e.g. 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, qubits=None)

Performs qubit encoding (mapping), using provided mapping class, of this instance.

This proceeds by creation of a `FermionOperator` class, via the `to_FermionOperator()` method, before mapping to a `QubitOperator` using the provided mapping scheme. Please see the documentation for `to_FermionOperator()` for finer grained control over operator generation.

Parameters

- **mapping** (`Optional[QubitMapping]`, default: `None`) – `QubitMapping`-derived instance. Default mapping procedure is `QubitMappingJordanWigner`.
- **qubits** (`Optional[List[Qubit]]`, default: `None`) – The qubit register. If left as `None`, a default register will be assumed if possible. See `QubitMapping` documentation for further details.

Returns

`QubitOperator` – Mapped `QubitOperator` object.

rotate(rotation_aa, rotation_bb, check_unitary=True, check_unitary_atol=1e-15)

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.
- **check_unitary** (bool, default: True) – Whether to perform the check for unitarity of the rotation matrices.
- **check_unitary_atol** (float, default: $1e-15$) – Absolute tolerance of unitarity checks.

Returns

ChemistryUnrestrictedIntegralOperatorCompact – This instance 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*.

Parameters

- **yield_constant** (bool, default: True) – Whether to include a constant term.
- **yield_one_body** (bool, default: True) – Whether to include one-body terms.
- **yield_two_body** (bool, default: True) – Whether to include two-body terms.

Returns

FermionOperator – Integral operator in general fermionic operator form.

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 due to 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 numpy dtype to cast into.

Returns

`CompactTwoBodyIntegralsS4` – New compact integrals object with compact array cast to the given `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 $i \leftrightarrow j$ and $k \leftrightarrow l$.

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.

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 array 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 values 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

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 multiple 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

The next pair index, first orbital index, and second orbital index.

Return type`Iterator[Tuple[int, int, int]]`**property real: `CompactTwoBodyIntegralsS4`**

Extract the real part of the two-body integrals.

Returns

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 $(n_{\text{orb}_{ij}}, n_{\text{orb}_{ij}})$. For rotating the first two indices of the two-body integrals tensor.
- **`u_kl`** (Optional[ndarray], default: None) – A real, orthogonal matrix of dimensions $(n_{\text{orb}_{kl}}, n_{\text{orb}_{kl}})$. 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

A tuple containing 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 that block does not respect $ij \leftrightarrow kl$ swap symmetry.

Parameters

`compact_array` (ndarray) – Two-body integrals in an eight-fold symmetry-reduced form. A `numpy.ndarray` with shape (n_{pp}) where $n_{pp} = n_{\text{pair}}(n_{\text{pair}} + 1)/2$ and $n_{\text{pair}} = n_{\text{orb}}(n_{\text{orb}} + 1)/2$.

`astype(dtype)`

Returns a copy of the current compact integrals, with the compact array cast to a new type.

Parameters

`dtype` (Any) – The `numpy.dtype` to cast into.

Returns

`CompactTwoBodyIntegralsS8` – New compact integrals object with compact array cast to the given `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 $i \leftrightarrow j$ and $k \leftrightarrow l$, and pair index swaps $ij \leftrightarrow kl$.

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.

```
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 uncompactd array 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 values 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

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 multiple 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

The next pair index, first orbital index, and second orbital index.

Return type

`Iterator[Tuple[int, int, int]]`

property real: `CompactTwoBodyIntegralsS8`

Extract the real part of the two-body integrals.

Returns

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 ($n_{\text{orb}}, n_{\text{orb}}$).

Return type

None

property shape: Tuple[int, int, int, int]

The shape of the corresponding rank-4 tensor.

Returns

A tuple containing the shape (n_p, n_q, n_r, n_s) of the two body tensor ($pq|rs$).

```
class DecompositionMethod(value, names=None, *, module=None, qualname=None, type=None, start=1,
                           boundary=None)
```

Bases: `Enum`

Methods for factorization of two electron integrals tensor.

`CHOLESKY = 'cho'`

`EIG = 'eig'`

```
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 $\hat{H} = H_0 + \hat{H}_1 + \hat{S} + \hat{V}$ where $\hat{S} + \hat{V}$ is the coulomb interaction. $\hat{V} = \frac{1}{2} \sum_{ijkl} (ij|kl) a_i^\dagger a_j a_k^\dagger a_l$ is a reordered two-body operator and \hat{S} is a one-body offset to the two-body operator given by $\hat{S} = \sum_{ij} s_{ij} a_i^\dagger a_j$ where $s_{ij} = -\frac{1}{2} \sum_k (ik|kj)$. H_0 and \hat{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 $A_t = \gamma^t \left(\sum_u^{N_\lambda} \lambda_u^t a_u^\dagger a_u \right)^2$ and $R(U_t)$ is a basis rotation operator.

One-body-like terms may be consolidated into a single effective-one-body term: $\hat{H}'_1 = \hat{H}_1 + \hat{S}$. One-body term(s) may be stored as arrays, or diagonalized. When diagonalized, one-body terms are given by: $\hat{H}_1 = R(W) \left(\sum_i^N \omega_i a_i^\dagger a_i \right) R(W)^\dagger$.

Parameters

- **constant** (`float`) – Constant (electron-independent) energy.
- **one_body** (`Union[DiagonalizedOneBodyIntegrals, ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]`) – One-body integrals. These are the physical one-body integrals \hat{H}_1 if `one_body_offset != None`, or effective one-body integrals \hat{H}'_1 if `one_body_offset == None`.
- **one_body_offset** (`Union[DiagonalizedOneBodyIntegrals, ndarray[Any, dtype[TypeVar(_ScalarType_co, 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.

`items` (`include_constant=True`, `include_one_body=True`, `include_one_body_offset=True`,
`include_two_body=True`, `square_two_body_sum=True`)

Iterate through 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.
- `include_one_body_offset` (`bool`, default: `True`) – Whether to include the one-body offset terms. If `one_body_offset == None`, this term is omitted anyway.
- `include_two_body` (`bool`, default: `True`) – Whether to include the two-body terms.
- `square_two_body_sum` (`bool`, default: `True`) – Whether to square the linear combination of number operators in the two-body term.

Yields

Fermion operators terms, corresponding rotation matrices.

Return type

`Iterator[Tuple[FermionOperator, ndarray[Any, dtype[float]]]]`

`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`) – The filename of the file to be read.

Return type`None`**`one_body_to_array()`**

Unpack the one-body data into a $N \times N$ 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_co, 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 $N \times N$ one-body matrix and $N \times N \times N \times N$ two-body tensor, where N is the number of spatial orbitals.

Parameters

`filename` – The name of the FCIDUMP file.

Returns

`ChemistryRestrictedIntegralOperator` – A `ChemistryRestrictedIntegralOperator` object corresponding to the integrals contained in the FCIDUMP file represented by this instance.

`to_arrays()`

Transform the one- and two-body data into their unpacked array forms.

Chemist notation is followed, and permutationally equivalent elements to those in the FCIDUMP file are filled.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – The one- and two-body integral arrays as a tuple.

`two_body_to_tensor()`

Unpack the one-body data into an $N \times N \times N \times N$ 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_co, 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_co, bound= generic, covariant=True)]])` – The spatial orbital, one-body integrals as a numpy array.

- **two_body** (ndarray[*Any*, dtype[TypeVar(_ScalarType_co, 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 spin 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*

Represents an operator on a Fermionic Hilbert space.

The operator is stored as a *dict*, with the keys being *FermionOperatorString* objects (terms) and values being the corresponding term coefficients. This can be directly instantiated with a *tuple*, a *list* of *tuples*, or a *dict*.

See examples below, and the documentation for the *from_tuple()* and *from_list()* methods for further detail.

Parameters

- **data** (*Union[Tuple, FermionOperatorString, List[Tuple[Union[int, float, complex, Expr], Union[FermionOperatorString, str]]], Dict[FermionOperatorString, Union[int, float, complex, Expr]]]*, 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, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None*)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.

capitalize()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()

Return a version of the string suitable for caseless comparisons.

center(width, fillchar=' ', /)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count(sub[, start[, end]]) → int

Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode(encoding='utf-8', errors='strict')

Encode the string using the codec registered for encoding.

encoding

The encoding in which to encode the string.

errors

The error handling scheme to use for encoding errors. The default is ‘strict’ meaning that encoding errors raise a UnicodeEncodeError. Other possible values are ‘ignore’, ‘replace’ and ‘xmlcharrefreplace’ as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(suffix[, start[, end]]) → bool

Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings 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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(*args, **kwargs) → str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(mapping) → str

Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index(*sub*[, *start*[, *end*]]) → **int**

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

`istitle()`

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

`isupper()`

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

`join(iterable, /)`

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

`ljust(width, fillchar=' ', /)`

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

`lower()`

Return a copy of the string converted to lowercase.

`lstrip(chars=None, /)`

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

`static maketrans()`

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

`partition(sep, /)`

Partition the string into three parts using the given separator.

This will search for the separator in the string. 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 string and two empty strings.

`removeprefix(prefix, /)`

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return `string[len(prefix):]`. Otherwise, return a copy of the original string.

`removesuffix(suffix, /)`

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return `string[:-len(suffix)]`. Otherwise, return a copy of the original string.

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 S where substring sub is found, such that sub is contained within S[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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust(width, fillchar=' ', /)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition(sep, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string, 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 strings and the original string.

rsplit(sep=None, maxsplit=-1)

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

rstrip(chars=None, /)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split(sep=None, maxsplit=-1)

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines (keepends=False)
 Return a list of the lines in the string, 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 S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip (chars=None, /)
 Return a copy of the string with leading and trailing whitespace removed.
 If chars is given and not None, remove characters in chars instead.

swapcase ()
 Convert uppercase characters to lowercase and lowercase characters to uppercase.

title ()
 Return a version of the string where each word is titlecased.
 More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate (table, /)
 Replace each character in the string using the given translation table.
table
 Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.
 The table must implement lookup/indexing via `__getitem__`, for instance a dictionary or list. If this operation raises `LookupError`, the character is left untouched. Characters mapped to None are deleted.

upper ()
 Return a copy of the string converted to uppercase.

zfill (width, /)
 Pad a numeric string with zeros on the left, to fill a field of the given width.
 The string is never truncated.

apply_bra (fock_state)
 Performs an operation on a `inquanto.states.FermionState` representing a bra.
 This transforms the provided `inquanto.states.FermionState` with the operator on the right. Conjugation on the `inquanto.states.FermionState` object is not performed.

Parameters
`fock_state (FermionState)` – Object representing a bra.

Returns
`FermionState` – A representation of the post-operation bra.

apply_ket (fock_state)
 Performs an operation on a ket `inquanto.states.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 this instance and another *FermionOperator*.

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 against 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 of terms and will return zero only if all coefficients are zero.

Parameters

- **abs_tol** (*float*, default: *None*) – Tolerance for checking if coefficients are zero. Set to *None* to test using a standard

:param 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 its *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 its `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 its `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.

commutator(other_operator)

Returns the commutator of this instance 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 this instance 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 `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 ansatzes and digital quantum simulation. Otherwise, it may remove terms relevant to computation. Each expression is of the form $f(a_1, a_2, \dots, a_n)$ for some symbols a_i . $|f(a_1, a_2, \dots, a_n)|$ 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 its `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]]`) – 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 annihilation 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 its `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 have complex values.

Returns

`bool` – False if a non-complex value occurs before any free symbols in the `dict` values, or True if no non-complex values occur.

Warning

Returns `None` if a free symbol occurs before any non-complex values in the coefficients.

is_all_coeff_imag()

Check if all coefficients have purely imaginary values.

Returns

`bool` – False if a non-complex value occurs before any free symbols in the `dict` values, or True if no non-complex values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any non-imaginary values in the coefficients.

`is_all_coeff_real()`

Check if all coefficients have real values.

Returns

`bool` – `False` if a non-real value occurs before any free symbols in the `dict` values, or `True` if no non-real values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any non-real values in the `dict` coefficients.

`is_all_coeff_symbolic()`

Check if all coefficients contain free symbols.

Returns

`bool` – Whether all coefficients contain free symbols.

`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 anti-Hermitian, otherwise `False`.

☢ Danger

This method uses the `normal_ordered()` method internally, which may be exponentially costly.

`is_any_coeff_complex()`

Check if any coefficients have complex values.

Returns

`bool` – `True` if a complex value occurs before any free symbols in the `dict` values, or `False` if no complex values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any complex values in the coefficients.

`is_any_coeff_imag()`

Check if any coefficients have imaginary values.

Returns

`bool` – True if an imaginary value occurs before any free symbols in the `dict` values, or `False` if no imaginary values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any imaginary values in the coefficients.

`is_any_coeff_real()`

Check if any coefficients have real values.

Returns

`bool` – True if a real value occurs before any free symbols in the `dict` values, or `False` if no real values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any real values in the coefficients.

`is_any_coeff_symbolic()`

Check if any coefficients contain free symbols.

Returns

`bool` – Whether any coefficients contain free symbols.

`is_commuting_operator()`

Returns `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()`

Returns whether or not term is in normal-ordered form.

The assumed convention for normal ordering is for all creation operators to be 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_normalized(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_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 a molecular Hamiltonian.

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`, default: `False`) – 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 -

:param see `FermionOperatorString.from_string()` for more detail.:

Returns

`FermionOperatorString` – A generated `FermionOperatorString` instance.

list_class

alias of `FermionOperatorList`

make_hashable ()

Return a hashable representation of the object.

Returns

`str` – A string representation of this instance.

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` – This instance.

property n_symbols: int
 Returns the number of free symbols in the object.

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).

⚠ Danger

This may be exponential in runtime.

Returns

FermionOperator – The operator in normal-ordered form.

➊ 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 )
```

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.

permuted_operator(permuation)

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

- permuation** (`Union[Dict, List[int]]`) – Mapping from the old operator terms indices to the new ones. In case if a `list` is given, its indices acts as keys (old indices) and its values

correspond to the new indices 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 class, of this instance.

Parameters

- `mapping` (`QubitMapping`, default: `None`) – `QubitMapping`-derived instance. Default mapping procedure is `QubitMappingJordanWigner`.
- `qubits` (`Optional[List[Qubit]]`, default: `None`) – The qubit register. If left as `None`, a default register will be assumed if possible. See `QubitMapping` documentation for further details.

Returns

`QubitOperator` – Mapped `QubitOperator` object.

`remove_global_phase()`

Returns a copy with a global phase applied such that the first element has a real coefficient.

This is an alias for `set_global_phase()` - see this method for greater control over the phase to be applied.

Returns

`LinearDictCombiner` – A copy of the object with a global phase applied such that the first element has a real coefficient.

`reversed_order()`

Reverses the order of terms and returns it as a new object.

Return type

`LinearDictCombiner`

`set_global_phase(phase=0.0)`

Returns a copy with a global phase applied such that the first element has the desired phase.

Parameters

- `phase` (`Union[int, float, complex, Expr]`, default: `0.0`) – The phase to yield on the first element, in half-turns (i.e. multiples of pi).

- **phase**. (A symbolic expression can be assigned to)

Returns

LinearDictCombiner – A copy of the object with the desired global phase applied.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

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` – Sympification fails.

property terms: List[Any]

Returns dictionary keys.

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} a_{3}^{\dagger} + c a_{0} a_{5} a_{3}^{\dagger} \rightarrow \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}
>>>
>>> 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} \rightarrow Z_{2}
>>> qo = QubitOperator({qos3: 3j})
>>> print(qo.to_latex(imaginary_unit="j", show_labels=True))
```

```
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 instance with each element 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 supports values 1 (i.e. AB) and 2 (i.e. $A/2 B/2 B/2 A/2$)
- **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 returned instance, with the coefficient of each inner `Operator` set to 1. This behaviour can be controlled with this argument - set to "outer" for default behaviour. On the other hand, setting this parameter to "inner" will store all scalars in the inner `Operator` coefficient, with the outer coefficients of the returned instance set to 1. Setting this parameter to "mixed" will store the Trotter factor in the outer coefficients of the returned instance, 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.

❶ 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** (default: `1e-8`) – Threshold below which terms are removed.
- **normal_ordered** (default: `True`) – Should the operator be returned in normal-ordered form.

Returns

FermionOperator – New, modified operator.

i 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(*precision*=*15*, *partial*=*False*)

Unsympifies dictionary values.

Replaces symbolic expressions with their corresponding numeric values.

Parameters

- **precision** (`int`, default: 15) – The number of decimal digits of precision used for evaluation.
- **partial** (`bool`, default: `False`) – Set to `True` to allow partial unsympification where terms containing free symbols are present. By default, free symbols in any coefficient will cause an exception.

Returns

`LinearDictCombiner` – Updated instance of `LinearDictCombiner`.

Raises

`TypeError` – 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 represent 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` containing two-element `tuples` of scalar or symbolic coefficients and `FermionOperator` objects. In contrast to `FermionOperator`, in this class the ordering of terms is well-defined and consistent.

Parameters

- `data` (`Union[FermionOperator, FermionOperatorString, List[Tuple[Union[int, float, complex, Expr], FermionOperator]]]`, 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, names=None, *, module=None, qualname=None, type=None,
                                start=1, boundary=None)
```

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.

capitalize()
    Return a capitalized version of the string.

    More specifically, make the first character have upper case and the rest lower case.

casefold()
    Return a version of the string suitable for caseless comparisons.

center(width, fillchar=' ', /)
    Return a centered string of length width.

    Padding is done using the specified fill character (default is a space).

count(sub[, start[, end]]) → int
    Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode(encoding='utf-8', errors='strict')
    Encode the string using the codec registered for encoding.

encoding
    The encoding in which to encode the string.

errors
    The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(suffix[, start[, end]]) → bool
    Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings 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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

    Return -1 on failure.

format(*args, **kwargs) → str
    Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(mapping) → str
    Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

```

index(*sub*[, *start*[, *end*]]) → **int**

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

`istitle()`

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

`isupper()`

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

`join(iterable, /)`

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

`ljust(width, fillchar=' ', /)`

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

`lower()`

Return a copy of the string converted to lowercase.

`lstrip(chars=None, /)`

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

`static maketrans()`

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

`partition(sep, /)`

Partition the string into three parts using the given separator.

This will search for the separator in the string. 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 string and two empty strings.

`removeprefix(prefix, /)`

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return `string[len(prefix):]`. Otherwise, return a copy of the original string.

`removesuffix(suffix, /)`

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return `string[:-len(suffix)]`. Otherwise, return a copy of the original string.

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 S where substring sub is found, such that sub is contained within S[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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust(width, fillchar=' ', /)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition(sep, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string, 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 strings and the original string.

rsplit(sep=None, maxsplit=-1)

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

rstrip(chars=None, /)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split(sep=None, maxsplit=-1)

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines (keepends=False)
 Return a list of the lines in the string, 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 S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip (chars=None, /)
 Return a copy of the string with leading and trailing whitespace removed.
 If chars is given and not None, remove characters in chars instead.

swapcase ()
 Convert uppercase characters to lowercase and lowercase characters to uppercase.

title ()
 Return a version of the string where each word is titlecased.
 More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate (table, /)
 Replace each character in the string using the given translation table.
table
 Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.
 The table must implement lookup/indexing via `__getitem__`, for instance a dictionary or list. If this operation raises `LookupError`, the character is left untouched. Characters mapped to None are deleted.

upper ()
 Return a copy of the string converted to uppercase.

zfill (width, /)
 Pad a numeric string with zeros on the left, to fill a field of the given width.
 The string is never truncated.

class FactoryCoefficientsLocation (value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)
 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.

capitalize ()
 Return a capitalized version of the string.
 More specifically, make the first character have upper case and the rest lower case.

casefold ()
 Return a version of the string suitable for caseless comparisons.

center (*width*, *fillchar*='', /)

Return a centered string of length *width*.

Padding is done using the specified fill character (default is a space).

count (*sub*[, *start*[, *end*]]) → **int**

Return the number of non-overlapping occurrences of substring *sub* in string *S*[*start*:*end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

encode (*encoding*='utf-8', *errors*='strict')

Encode the string using the codec registered for encoding.

encoding

The encoding in which to encode the string.

errors

The error handling scheme to use for encoding errors. The default is ‘strict’ meaning that encoding errors raise a `UnicodeEncodeError`. Other possible values are ‘ignore’, ‘replace’ and ‘xmlcharrefreplace’ as well as any other name registered with `codecs.register_error` that can handle `UnicodeEncodeErrors`.

endswith (*suffix*[, *start*[, *end*]]) → **bool**

Return True if *S* ends with the specified suffix, False otherwise. With optional *start*, test *S* beginning at that position. With optional *end*, stop comparing *S* at that position. *suffix* can also be a tuple of strings 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 *S* where substring *sub* is found, such that *sub* is contained within *S*[*start*:*end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

Return -1 on failure.

format (**args*, ***kwargs*) → **str**

Return a formatted version of *S*, using substitutions from *args* and *kwargs*. The substitutions are identified by braces ('{' and '}').

format_map (*mapping*) → **str**

Return a formatted version of *S*, using substitutions from *mapping*. The substitutions are identified by braces ('{' and '}').

index (*sub*[, *start*[, *end*]]) → **int**

Return the lowest index in *S* where substring *sub* is found, such that *sub* is contained within *S*[*start*:*end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

Raises `ValueError` when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join(iterable, /)

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `''.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust (width, fillchar=' ', /)

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower ()

Return a copy of the string converted to lowercase.

lstrip (chars=None, /)

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

static maketrans ()

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition (sep, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string. 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 string and two empty strings.

removeprefix (prefix, /)

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(prefix):]. Otherwise, return a copy of the original string.

removesuffix (suffix, /)

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:-len(suffix)]. Otherwise, return a copy of the original string.

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 S where substring sub is found, such that sub is contained within S[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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust (width, fillchar=' ', /)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition (sep, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string, 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 strings and the original string.

rsplit (sep=None, maxsplit=-1)

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

rstrip (chars=None, /)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split (sep=None, maxsplit=-1)

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines (keepends=False)

Return a list of the lines in the string, 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 S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip (chars=None, /)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

swapcase ()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

title()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate(*table*, /)

Replace each character in the string using the given translation table.

table

Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via `__getitem__`, for instance a dictionary or list. If this operation raises `LookupError`, the character is left untouched. Characters mapped to None are deleted.

upper()

Return a copy of the string converted to uppercase.

zfill(*width*, /)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

clone()

Performs shallow copy of the object.

Return type

`TypeVar(SYMBOLICTYPE, bound= Symbolic)`

collapse_as_linear_combination(*ignore_outer_coefficients=False*)

Treating this instance as a linear combination, return it in the form of an `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 constituent `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 this instance, multiplied by their associated coefficients if requested.

collapse_as_product(*reverse=False, ignore_outer_coefficients=False*)

Treating this instance 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 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)]

```

copy()

Returns a deep copy of this instance.

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 coefficients, 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, a value of "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

`TypeVar(OperatorListT, bound= OperatorList)` – 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.

Returns

`str` – A string representation of this instance.

map(mapping)

Updates right values of items 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.

property num_spin_orbs: int

Return the number of spin-orbitals that this operator explicitly acts on.

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 `self` represents a product of exponentials, with each constituent `Operator` corresponding to the exponentiated term of a single exponential in a 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 desired order of the final Trotter-Suzuki expansion. Currently, only a first order (*ABABAB...*) or second order (*ABBAABBA...*) expansion is supported.
- `initial_trotter_order` (`int`, default: 1) – The order of the original Trotter-Suzuki expansion used. 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 in the original object.

Parameters

`sublist_indices` (`list[int]`) – Indices of elements in this instance selected to constitute a new object.

Returns

`TypeVar(OperatorListT, bound= OperatorList)` – A sublist of this instance.

Raises

- `ValueError` – If `sublist_indices` contains indices not contained in this instance, or if this instance
- `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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

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`) – Maps symbol-representing keys to the value the symbol should be substituted for.

Returns

`LinearListCombiner` – This instance with symbols key symbols replaced by their values.

`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` – 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 `self` represents the exponential of a linear combination of `Operator` objects, each corresponding to a term in this linear combination. Trotterization is performed at the level of these `Operator` instances. The `Operator` objects 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. The first- and the second-order options are 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 (*precision*=15, *partial*=False)

Unsympifies dictionary values.

Replaces symbolic expressions with their corresponding numeric values.

Parameters

- ***precision*** (`int`, default: 15) – The number of decimal digits of precision used for evaluation.
- ***partial*** (default: `False`) – Set to `True` to allow partial unsympification where terms containing free symbols are present. By default, free symbols in any coefficient will cause an exception.

Returns

`LinearListCombiner` – Updated instance of `LinearListCombiner`.

Raises

`TypeError` – Unsympification fails.

untrotterize (*trotter_number*, *trotter_order*=1)

Reverse a Trotter-Suzuki expansion given a product of exponentials as an `OperatorList`.

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.

i **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 product of exponentials as an `OperatorList`, maintaining separation of exponents.

This method assumes that `self` represents a product of exponentials, with each constituent `Operator` corresponding to the exponentiated term of a single exponential within a product. Scalar factors within this `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 ($ABABAB\dots$) 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)

Bases: tuple

Represents a single fermionic string of creation and annihilation operators.

Internally this is a tuple of tuple objects, 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 operators correspondingly. Defined to constitute a single term in `inquanto.operators.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 `inquanto.states.FermionState` representing a bra.

This transforms the provided `inquanto.states.FermionState` with the operator string on the right. Conjugation on the `inquanto.states.FermionState` object is not performed.

Parameters

- **fock_state** (`FermionState`) – state object representing a bra.
- **power** (`int`, default: 1) – Power of operation (how many times operator acts on a bra state).

Returns

`FermionState` – A representation of the post-operation bra.

Examples

```
>>> f_op_string = FermionOperatorString.from_string("F1^ F0")
>>> bra = FermionState([0, 1])
>>> print(bra)
(1.0, {0: 0, 1: 1})
>>> print(f_op_string.apply_bra(bra))
(1.0, {0: 1, 1: 0})
>>> bra = FermionState([1, 0])
```

```
>>> print(f_op_string.apply_bra(bra))
()
```

apply_ket(*fock_state*, *power*=1)

Performs an operation on a ket `inquanto.states.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.

i Examples

```
>>> f_op_string = FermionOperatorString.from_string("F1^ F0")
>>> ket = FermionState([1, 0])
>>> print(ket)
(1.0, {0: 1, 1: 0})
>>> print(f_op_string.apply_ket(ket))
(1.0, {0: 0, 1: 1})
>>> ket = FermionState([0, 1])
>>> print(f_op_string.apply_ket(ket))
()
```

apply_state(*fock_state*, *state_type*, *power*)

Implements a general operation on a `FermionState` object representing a bra or a ket.

Parameters

- **fock_state** (`FermionState`) – Input fermion state.
- **state_type** (`StateType`) – Indicates whether the input state object represents a bra or a ket.
- **power** (`int`) – Power of operation.

Returns

`FermionState` – New state object.

dagger()

Performs a conjugation operation on a fermion creation-annihilation operator string.

Reverses order and swaps creation and annihilation labels.

Returns

`FermionOperatorString` – The Hermitian conjugate of the current operator string.

i Examples

```
>>> f_op_string = FermionOperatorString.from_string("F3^ F1^ F2 F0")
>>> print(f_op_string.dagger())
F0^ F2^ F1 F3
```

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.

i Examples

```
>>> FermionOperatorString.from_string("F3^ F2")
((3, 1), (2, 0))
>>> FermionOperatorString.from_string("3^ 2")
((3, 1), (2, 0))
```

is_empty()

Checks if object is empty.

Return type

`bool`

is_particle_conserving()

Checks if operator string is particle-conserving.

Returns

`bool` – True if the operator string conserves particle number, `False` if not.

i Examples

```
>>> FermionOperatorString.from_string("F3^ F2").is_particle_conserving()
True
>>> FermionOperatorString.from_string("F3^ F2 F0").is_particle_conserving()
False
```

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.

❶ 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 IntegralType(*value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None*)

Bases: `Enum`

Describes spin formalism used in integral operators.

`RESTRICTED = 'r'`

`UNRESTRICTED = 'u'`

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` (`array`, default: `None`) – Initial orbital coefficients.
- `occ` (`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` (`Callable`, default: `None`) – The objective function to be minimized.
- `minimizer` (`GeneralMinimizer`, default: `None`) – An InQuanto minimizer.
- `point_group` (`Union[PointGroup, str]`, default: `None`) – If passed, symmetry information will be used to reduce free parameters.
- `orbital_irreps` (`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, check_unitary=True, check_unitary_atol=1e-15*)

Computes the unitary relating square matrices `v_init` and `v_final` whose columns contain the initial and final orbitals.

Computes the matrix $U = V_{\text{init}}^{-1} V_{\text{final}}$.

Parameters

- `v_init` (`Optional[ndarray]`, default: `None`) – Initial orbitals.
- `v_final` (`Optional[ndarray]`, default: `None`) – Final orbitals.

- `check_unitary` (`bool`, default: `True`) – Whether to check for the unitarity of the resulting matrix.
- `check_unitary_atol` (`float`, default: `1e-15`) – Absolute tolerance of unitarity check.

Returns

`ndarray` – Unitary matrix relating initial and final orbitals.

`construct_random_variables` (`v=None`, `low=-0.1`, `high=0.1`, `seed=None`)

Constructs $n(n - 1)/2$ variables sampled from uniform distribution where n is the number of orbitals.

Uniform distribution is specified by given high, low and seed.

Parameters

- `v` (`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`)

Orthogonalises column vectors in `v` using Gram-Schmidt algorithm with respect to an overlap matrix.

Parameters

- `v` (`ndarray`) – Orbitals/vectors to be orthonormalised.
- `overlap` (`ndarray`, default: `None`) – Overlap matrix.

Returns

`ndarray` – Orthogonalised vectors.

`map_variables_to_rotation_matrix` (`variables=None`)

Maps $n(n - 1)/2$ variables to an $n \times n$ 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 an $n \times n$ skew-symmetric matrix from $n(n - 1)/2$ variables.

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*, *check_unitary=True*, *check_unitary_atol=1e-15*)

Computes the unitary relating square matrices `v_init` and `v_final` whose columns contain the initial and final orbitals.

Computes the matrix $U = V_{\text{init}}^{-1} V_{\text{final}}$.

Parameters

- **v_init** (`Optional[ndarray]`, default: `None`) – Initial orbitals.

- **v_final** (`Optional[ndarray]`, default: `None`) – Final orbitals.
- **check_unitary** (`bool`, default: `True`) – Whether to check for the unitarity of the resulting matrix.
- **check_unitary_atol** (`float`, default: `1e-15`) – Absolute tolerance of unitarity check.

Returns

`ndarray` – Unitary matrix relating initial and final orbitals.

static gram_schmidt (*v*, *overlap*=`None`)

Orthogonalises column vectors in *v* using Gram-Schmidt algorithm with respect to an overlap matrix.

Parameters

- **v** (`ndarray`) – Orbitals/vectors to be orthonormalised.
- **overlap** (`ndarray`, default: `None`) – Overlap matrix.

Returns

`ndarray` – Orthogonalised 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`, 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, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None*)

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.

capitalize()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()

Return a version of the string suitable for caseless comparisons.

center(*width, fillchar=' ', /*)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count (*sub*[*, start*[*, end*]]) → *int*

Return the number of non-overlapping occurrences of substring *sub* in string *S*[*start*:*end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

encode (*encoding*=‘utf-8’, *errors*=‘strict’)

Encode the string using the codec registered for encoding.

encoding

The encoding in which to encode the string.

errors

The error handling scheme to use for encoding errors. The default is ‘strict’ meaning that encoding errors raise a `UnicodeEncodeError`. Other possible values are ‘ignore’, ‘replace’ and ‘xmlcharrefreplace’ as well as any other name registered with `codecs.register_error` that can handle `UnicodeEncodeErrors`.

endswith (*suffix*[*, start*[*, end*]]) → *bool*

Return True if *S* ends with the specified suffix, False otherwise. With optional *start*, test *S* beginning at that position. With optional *end*, stop comparing *S* at that position. *suffix* can also be a tuple of strings 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 *S* where substring *sub* is found, such that *sub* is contained within *S*[*start*:*end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

Return -1 on failure.

format (**args*, ***kwargs*) → *str*

Return a formatted version of *S*, using substitutions from *args* and *kwargs*. The substitutions are identified by braces (‘{’ and ‘}’).

format_map (*mapping*) → *str*

Return a formatted version of *S*, using substitutions from *mapping*. The substitutions are identified by braces (‘{’ and ‘}’).

index (*sub*[*, start*[*, end*]]) → *int*

Return the lowest index in *S* where substring *sub* is found, such that *sub* is contained within *S*[*start*:*end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

Raises `ValueError` when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join(iterable, /)

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: ‘.’.join(['ab', ‘pq’, ‘rs’]) -> ‘ab.pq.rs’

`ljust(width, fillchar=' ', /)`

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

`lower()`

Return a copy of the string converted to lowercase.

`lstrip(chars=None, /)`

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

`static maketrans()`

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

`partition(sep, /)`

Partition the string into three parts using the given separator.

This will search for the separator in the string. 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 string and two empty strings.

`removeprefix(prefix, /)`

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(prefix):]. Otherwise, return a copy of the original string.

`removesuffix(suffix, /)`

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:-len(suffix)]. Otherwise, return a copy of the original string.

`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 S where substring sub is found, such that sub is contained within S[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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust (width, fillchar=' ', /)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition (sep, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string, 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 strings and the original string.

rsplit (sep=None, maxsplit=-1)

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

rstrip (chars=None, /)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split (sep=None, maxsplit=-1)

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines (keepends=False)

Return a list of the lines in the string, 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 S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip (chars=None, /)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

swapcase ()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

title()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate(*table*, /)

Replace each character in the string using the given translation table.

table

Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via `__getitem__`, for instance a dictionary or list. If this operation raises `LookupError`, the character is left untouched. Characters mapped to None are deleted.

upper()

Return a copy of the string converted to uppercase.

zfill(*width*, /)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

property all_nontrivial_qubits: set[Qubit]

Returns a set of all qubits acted upon by this operator nontrivially (i.e. with an X,Y or Z).

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. `:rtype: QubitOperator`

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)
```

`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` – Comparison of two values can't be done due to types mismatch.

Return type

`bool`

`approx_equal_to_by_random_substitution(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 against 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 of terms and will return zero only if all coefficients are zero.

Parameters

- **abs_tol** (`float`, default: `None`) – Tolerance for checking if coefficients are zero. Set to `None` to test using a standard

:param 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.

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.

Return type

`None`

⚠ Warning

When `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 ansatzes and digital quantum simulation. Otherwise, it may remove terms relevant to computation. Each expression is of the form $f(a_1, a_2, \dots, a_n)$ for some symbols a_i . $|f(a_1, a_2, \dots, a_n)|$ is assumed to monotonically increase in both real and imaginary components for all $a_i \in [0, 1]$.

`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 and operators, but the use of parametrised states and operators is untested. In this case, the optional `qubits` parameter is ignored.

For a `numpy.ndarray`, we delegate 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.

eigenspectrum (`hamming_weight=None, nroots=None, threshold=1e-5, check_hermitian=False, check_hermitian_atol=1e-10`)

Returns the eigenspectrum of a Hermitian operator, optionally filtered by a given Hamming weight.

More precisely, if `hamming_weight` is provided, only those eigenvalues whose eigenstates' computational components have coefficients larger than `threshold` and match the provided Hamming weight will be returned.

If argument `nroots` is provided, an iterative sparse matrix diagonalisation procedure is invoked. Otherwise, the whole dense matrix is diagonalised.

Notes

If this operator results from a Jordan-Wigner fermion-qubit encoding, filtering by Hamming weight corresponds to particle conservation. The operator is assumed to be Hermitian.

⚠ Warning

This method scales exponentially. Use only for testing on small systems.

Parameters

- **hamming_weight** (`Optional[int]`, default: `None`) – Hamming weight for filtering the roots.
- **nroots** (`Optional[int]`, default: `None`) – How many roots to calculate (invokes iterative diagonalisation).
- **threshold** (`float`, default: `1e-5`) – State coefficient threshold for checking Hamming weight.
- **check_hermitian** (`bool`, default: `False`) – Whether to check the hermiticity of the operator. Uses `is_hermitian()`.
- **check_hermitian_atol** (`float`, default: `1e-10`) – Absolute tolerance for hermiticity check. Passed to `is_hermitian()`.

Returns

`ndarray` – Array of (filtered) eigenvalues.

empty()

Checks if dictionary is empty.

Return type

`bool`

ensure_hermitian (`tolerance=1e-10`)

Eliminate all insignificant imaginary parts of numeric coefficients.

Raises

`ValueError` – imaginary symbolic, or significant imaginary numeric coefficient.

Parameters

`tolerance` (`float`, default: `1e-10`) – Threshold determining whether a numerical imag coefficient is significant.

Returns

`QubitOperator` – This instance.

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` – 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` – 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_QubitPauliOperator(qop, unsympify_coefficients=True)

Generate an instance of this class from a tket QubitPauliOperator.

Component QubitPauliStrings will be converted to `QubitOperatorStrings`. By default, coefficients without free symbols will be converted to float (if real) or complex (otherwise). To disable this, set `unsympify_coefficients = False`. For finer grained control over unsympification, generate with `unsympify_coefficients = False` and `unsympify` after generation. See `unsympify()` for further details.

Parameters

- `qop` (`QubitPauliOperator`) – The `QubitPauliOperator` to be converted.
- `unsympify_coefficients` (`bool`, default: `True`) – Whether to cast coefficents to standard python float/complex. Set to `False` to disable.

Returns

The converted operator.

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)

Get the coefficient value of the provided Pauli string.

Parameters

`key` (`QubitPauliString`)

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. :rtype: `QubitOperator`

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)
```

classmethod identity()

Return an identity operator. :rtype: *QubitOperator*

i Examples

```
>>> print(QubitOperator.identity())
(1.0, )
```

is_all_coeff_complex()

Check if all coefficients have complex values.

Returns

`bool` – False if a non-complex value occurs before any free symbols in the `dict` values, or True if no non-complex values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any non-complex values in the coefficients.

is_all_coeff_imag()

Check if all coefficients have purely imaginary values.

Returns

`bool` – False if a non-complex value occurs before any free symbols in the `dict` values, or True if no non-complex values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any non-imaginary values in the coefficients.

is_all_coeff_real()

Check if all coefficients have real values.

Returns

`bool` – False if a non-real value occurs before any free symbols in the `dict` values, or True if no non-real values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any non-real values in the `dict` coefficients.

is_all_coeff_symbolic()

Check if all coefficients contain free symbols.

Returns

`bool` – Whether all coefficients contain free symbols.

`is_antihermitian`(*tolerance*=*1e-10*)

Check if operator is anti-Hermitian (purely imaginary coefficients).

Check is performed by looking for symbolic or significant numeric real part in at least one coefficient

Returns

`bool` – True if anti-Hermitian, `False` otherwise.

Parameters

`tolerance` (`float`, default: `1e-10`)

`is_any_coeff_complex()`

Check if any coefficients have complex values.

Returns

`bool` – True if a complex value occurs before any free symbols in the `dict` values, or `False` if no complex values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any complex values in the coefficients.

`is_any_coeff_imag()`

Check if any coefficients have imaginary values.

Returns

`bool` – True if an imaginary value occurs before any free symbols in the `dict` values, or `False` if no imaginary values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any imaginary values in the coefficients.

`is_any_coeff_real()`

Check if any coefficients have real values.

Returns

`bool` – True if a real value occurs before any free symbols in the `dict` values, or `False` if no real values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any real values in the coefficients.

`is_any_coeff_symbolic()`

Check if any coefficients contain free symbols.

Returns

`bool` – Whether any coefficients contain free symbols.

`is_commuting_operator()`

Returns `True` if every term in operator commutes with every other term, otherwise `False`.

Return type`bool`**`is_hermitian`(*tolerance*= $1e-10$)**

Check if operator is Hermitian.

Check is performed by looking for symbolic or significant numeric imaginary part in at least one coefficient

Returns`bool` – True if Hermitian, False otherwise.**Parameters**`tolerance` (`float`, default: $1e-10$)**`static is_hermitian_coeff`(*coeff*, *tolerance*= $1e-10$)**Determine whether the given coefficient can be present in a Hermitian *QubitOperator*.**Parameters**

- `coeff` (`Union[int, float, complex, Expr]`) – Coefficient to check
- `tolerance` (`float`, default: $1e-10$) – Threshold determining whether a numerical imaginary coefficient is significant.

Returns`bool` – Whether coeff can be present in a Hermitian *QubitOperator*.**`is_normalized`(*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_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 (default: `1e-10`) – 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.

Returns

`str` – A string representation of this instance.

`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` – This instance.

`property n_symbols: int`

Returns the number of free symbols in the object.

`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 on $[0, N]$, 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.

Return type

None

```
>>> 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)
```

`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 on $[0, N]$, 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*.

❶ 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.

`print_table()`

Print dictionary formatted as a table.

Return type

`NoReturn`

`qubitwise_anticomutes_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 `QubitOperator` objects 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` – 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` – either operator consists of more than a single term.

`remove_global_phase()`

Returns a copy with a global phase applied such that the first element has a real coefficient.

This is an alias for `set_global_phase()` - see this method for greater control over the phase to be applied.

Returns

`LinearDictCombiner` – A copy of the object with a global phase applied such that the first element has a real coefficient.

`reversed_order()`

Reverses the order of terms and returns it as a new object.

Return type

`LinearDictCombiner`

`set_global_phase(phase=0.0)`

Returns a copy with a global phase applied such that the first element has the desired phase.

Parameters

- `phase` (`Union[int, float, complex, Expr]`, default: `0.0`) – The phase to yield on the first element, in half-turns (i.e. multiples of pi).
- `phase.` (A symbolic expression can be assigned to)

Returns

`LinearDictCombiner` – A copy of the object with the desired global phase applied.

`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 and operators, but the use of parametrised states and operators is untested. For a `numpy.ndarray`, we delegate 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*)

Substitution for symbolic expressions.

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` – A new `QubitOperator` object with symbols substituted.

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` – Sympification fails.

symplectic_representation (*qubits=None*)

Generate the symplectic representation of the operator.

This is a binary ($M \times 2N$) matrix where M is the number of terms and N is the number of qubits. In the leftmost half of the matrix, the element indexed by $[i, j]$ 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 indexed by $[i, j]$ is `True` where qubit $j - \text{num_qubits}$ 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

Raises

`ValueError` – operator contains qubits which are not in the provided qubits argument.

property terms: List[Any]

Returns dictionary keys.

to_QubitPauliOperator()

Converts this object to a tket `QubitPauliOperator`.

Component `QubitOperatorStrings` will be converted to tket `QubitPauliStrings`. Note that coefficients will implicitly be converted to `sympy` objects in accordance with the `QubitPauliOperator` API.

Returns

The converted operator.

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} 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}
```

```
>>> 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]),
   - ("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}\text{ - }
\text{c} 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)
(a, X0 Y1), (re(c), Z0 Z1)
>>> print(antiherm_qo)
(1.0*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 `QubitOperator`s. 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

`Tuple[QubitOperator, QubitOperator]` – 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 instance with each element 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 supports values 1 (i.e. AB) and 2 (i.e. $A/2 B/2 B/2 A/2$)

- **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 returned instance, with the coefficient of each inner `Operator` set to 1. This behaviour can be controlled with this argument - set to "outer" for default behaviour. On the other hand, setting this parameter to "inner" will store all scalars in the inner `operator` coefficient, with the outer coefficients of the returned instance set to 1. Setting this parameter to "mixed" will store the Trotter factor in the outer coefficients of the returned instance, 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` (`precision=15, partial=False`)

Unsympifies dictionary values.

Replaces symbolic expressions with their corresponding numeric values.

Parameters

- **precision** (`int`, default: 15) – The number of decimal digits of precision used for evaluation.
- **partial** (`bool`, default: False) – Set to True to allow partial unsympification where terms containing free symbols are present. By default, free symbols in any coefficient will cause an exception.

Returns

`LinearDictCombiner` – Updated instance of `LinearDictCombiner`.

Raises

`TypeError` – Unsympification fails.

`classmethod zero()`

Return object with a zero `dict` entry.

Examples

```
>>> print(LinearDictCombiner.zero())
()
```

`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 `QubitOperators`.

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]]]`, 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, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None*)

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.

capitalize()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()

Return a version of the string suitable for caseless comparisons.

center(*width, fillchar=' ', /*)

Return a centered string of length *width*.

Padding is done using the specified fill character (default is a space).

count(*sub[, start[, end]]*) → int

Return the number of non-overlapping occurrences of substring *sub* in string *S[start:end]*. Optional arguments *start* and *end* are interpreted as in slice notation.

encode(*encoding='utf-8', errors='strict'*)

Encode the string using the codec registered for encoding.

encoding

The encoding in which to encode the string.

errors

The error handling scheme to use for encoding errors. The default is ‘strict’ meaning that encoding errors raise a `UnicodeEncodeError`. Other possible values are ‘ignore’, ‘replace’ and ‘xmlcharrefreplace’ as well as any other name registered with `codecs.register_error` that can handle `UnicodeEncodeErrors`.

endswith(*suffix[, start[, end]]*) → bool

Return True if *S* ends with the specified suffix, False otherwise. With optional *start*, test *S* beginning at that position. With optional *end*, stop comparing *S* at that position. *suffix* can also be a tuple of strings 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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

`format (*args, **kwargs) → str`

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

`format_map (mapping) → str`

Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

`index (sub[, start[, end]]) → int`

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

`isalnum ()`

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

`isalpha ()`

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

`isascii ()`

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

`isdecimal ()`

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

`isdigit ()`

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

`isidentifier ()`

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

`islower()`

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

`isnumeric()`

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

`isprintable()`

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in `repr()` or if it is empty.

`isspace()`

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

`istitle()`

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

`isupper()`

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

`join(iterable, /)`

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

`ljust(width, fillchar=' ', /)`

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

`lower()`

Return a copy of the string converted to lowercase.

`lstrip(chars=None, /)`

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

`static maketrans()`

Return a translation table usable for `str.translate()`.

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will

be mapped to the character at the same position in *y*. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string. 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 string and two empty strings.

removeprefix(*prefix*, /)

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(*prefix*) :]. Otherwise, return a copy of the original string.

removesuffix(*suffix*, /)

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:len(*suffix*)]. Otherwise, return a copy of the original string.

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 *S* where substring *sub* is found, such that *sub* is contained within *S*[*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 *S* where substring *sub* is found, such that *sub* is contained within *S*[*start*:*end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust(*width*, *fillchar*='', /)

Return a right-justified string of length *width*.

Padding is done using the specified fill character (default is a space).

rpartition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string, 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 strings and the original string.

rsplit(*sep*=None, *maxsplit*=-1)

Return a list of the substrings in the string, using *sep* as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

rstrip (chars=None, /)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split (sep=None, maxsplit=-1)

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines (keepends=False)

Return a list of the lines in the string, 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 S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip (chars=None, /)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

swapcase ()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

title ()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate (table, /)

Replace each character in the string using the given translation table.

table

Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via `__getitem__`, for instance a dictionary or list. If this operation raises `LookupError`, the character is left untouched. Characters mapped to None are deleted.

upper ()

Return a copy of the string converted to uppercase.

zfill (width, /)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

```
class ExpandExponentialProductCoefficientsBehavior(value, names=None, *, module=None,
                                                qualname=None, type=None, start=1,
                                                boundary=None)

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.

capitalize()
    Return a capitalized version of the string.

    More specifically, make the first character have upper case and the rest lower case.

casefold()
    Return a version of the string suitable for caseless comparisons.

center(width, fillchar=' ', /)
    Return a centered string of length width.

    Padding is done using the specified fill character (default is a space).

count(sub[, start[, end]]) → int
    Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode(encoding='utf-8', errors='strict')
    Encode the string using the codec registered for encoding.

encoding
    The encoding in which to encode the string.

errors
    The error handling scheme to use for encoding errors. The default is ‘strict’ meaning that encoding errors raise a UnicodeEncodeError. Other possible values are ‘ignore’, ‘replace’ and ‘xmlcharrefreplace’ as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(suffix[, start[, end]]) → bool
    Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings 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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(**args*, ***kwargs*) → str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*) → str

Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index(*sub*[, *start*[, *end*]]) → int

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as "def" or "class".

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

`isnumeric()`

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

`isprintable()`

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in `repr()` or if it is empty.

`isspace()`

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

`istitle()`

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

`isupper()`

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

`join(iterable, /)`

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

`ljust(width, fillchar=' ', /)`

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

`lower()`

Return a copy of the string converted to lowercase.

`lstrip(chars=None, /)`

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

`static maketrans()`

Return a translation table usable for `str.translate()`.

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition(sep, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string. 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 string and two empty strings.

removeprefix(prefix, /)

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(prefix):]. Otherwise, return a copy of the original string.

removesuffix(suffix, /)

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:-len(suffix)]. Otherwise, return a copy of the original string.

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 S where substring sub is found, such that sub is contained within S[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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust(width, fillchar=' ', /)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition(sep, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string, 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 strings and the original string.

rsplit(sep=None, maxsplit=-1)

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

rstrip (*chars=None*, /)

Return a copy of the string with trailing whitespace removed.

If *chars* is given and not None, remove characters in *chars* instead.

split (*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using *sep* as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines (*keepends=False*)

Return a list of the lines in the string, 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 S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. *prefix* can also be a tuple of strings to try.

strip (*chars=None*, /)

Return a copy of the string with leading and trailing whitespace removed.

If *chars* is given and not None, remove characters in *chars* instead.

swapcase ()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

title ()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate (*table*, /)

Replace each character in the string using the given translation table.

table

Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via `__getitem__`, for instance a dictionary or list. If this operation raises `LookupError`, the character is left untouched. Characters mapped to None are deleted.

upper ()

Return a copy of the string converted to uppercase.

zfill (*width*, /)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

```
class FactoryCoefficientsLocation(value, names=None, *, module=None, qualname=None,
                                         type=None, start=1, boundary=None)

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.

capitalize()
    Return a capitalized version of the string.

    More specifically, make the first character have upper case and the rest lower case.

casefold()
    Return a version of the string suitable for caseless comparisons.

center(width, fillchar=' ', /)
    Return a centered string of length width.

    Padding is done using the specified fill character (default is a space).

count(sub[, start[, end ]]) → int
    Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode(encoding='utf-8', errors='strict')
    Encode the string using the codec registered for encoding.

encoding
    The encoding in which to encode the string.

errors
    The error handling scheme to use for encoding errors. The default is ‘strict’ meaning that encoding errors raise a UnicodeEncodeError. Other possible values are ‘ignore’, ‘replace’ and ‘xmlcharrefreplace’ as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(suffix[, start[, end ]]) → bool
    Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings 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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

    Return -1 on failure.

format(*args, **kwargs) → str
    Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').
```

`format_map(mapping) → str`

Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

`index(sub[, start[, end]]) → int`

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

`isalnum()`

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

`isalpha()`

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

`isascii()`

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

`isdecimal()`

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

`isdigit()`

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

`isidentifier()`

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

`islower()`

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

`isnumeric()`

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

`isprintable()`

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

`isspace()`

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

`istitle()`

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

`isupper()`

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

`join(iterable, /)`

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `''.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

`ljust(width, fillchar=' ', /)`

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

`lower()`

Return a copy of the string converted to lowercase.

`lstrip(chars=None, /)`

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

`static maketrans()`

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

`partition(sep, /)`

Partition the string into three parts using the given separator.

This will search for the separator in the string. 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 string and two empty strings.

`removeprefix(prefix, /)`

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(prefix):]. Otherwise, return a copy of the original string.

`removesuffix(suffix, /)`

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:-len(suffix)]. Otherwise, return a copy of the original string.

`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 S where substring sub is found, such that sub is contained within S[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 S where substring sub is found, such that sub is contained within S[start:end].

Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

`rjust(width, fillchar=' ', /)`

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

`rpartition(sep, /)`

Partition the string into three parts using the given separator.

This will search for the separator in the string, 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 strings and the original string.

`rsplit(sep=None, maxsplit=-1)`

Return a list of the substrings in the string, using sep as the separator string.

`sep`

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

`maxsplit`

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

`rstrip(chars=None, /)`

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

`split(sep=None, maxsplit=-1)`

Return a list of the substrings in the string, using sep as the separator string.

`sep`

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines (keepends=False)

Return a list of the lines in the string, 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 S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip (chars=None, /)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

swapcase ()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

title ()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate (table, /)

Replace each character in the string using the given translation table.

table

Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via `__getitem__`, for instance a dictionary or list. If this operation raises `LookupError`, the character is left untouched. Characters mapped to `None` are deleted.

upper ()

Return a copy of the string converted to uppercase.

zfill (width, /)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

property all_nontrivial_qubits: set [Qubit]

Returns a set of all qubits acted upon by the operators in this `QubitOperatorList` nontrivially (i.e. with an X,Y or Z).

property all_qubits: Set [Qubit]

Returns a set of all qubits included in any operator in the `QubitOperatorList`.

build_subset (indices)

Builds a subset of the class based on a list of indices passed by a user.

Parameters

`indices` (`List[int]`) – Indices of the items which are selected to for the subset as a new `QubitOperatorList` instance.

Raises

`ValueError` – the number of requested indices is larger than the number of elements in this instance.

Returns

`QubitOperatorList` – Subset of terms of the initial object.

clone()

Performs shallow copy of the object.

Return type

`TypeVar(SYMBOLICTYPE, bound= Symbolic)`

collapse_as_linear_combination(ignore_outer_coefficients=False)

Treating this instance as a linear combination, return it in the form of an `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 constituent `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 this instance, multiplied by their associated coefficients if requested.

collapse_as_product(reverse=False, ignore_outer_coefficients=False)

Treating this instance 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 ($N \times N$) matrix where : N is the number of operators in the `OperatorList`. Element indexed by $[n, m]$ is True if operators indexed by 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 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 resultant matrix has a shape (N, M) and :math:`J_{i,j} = \frac{\partial c_i}{\partial \theta_j}

Parameters

- **symbols** (`List[Symbol]`) – Symbols w.r.t. which each coefficient is differentiated.
- **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 the form `(i, j, J_ij)` (this format is referred as a row-sorted list of non-zero elements of the matrix.)

❶ 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 this instance.

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 delegate 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 delegate 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 ($N \times N$) matrix where N is the number of operators in the `OperatorList`. Element indexed by $[n, m]$ is `True` if operators indexed by 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*=*ExpandExponentialProductCoefficientsBehavior*, *additional_exponent*=1.0, *check_commuting*=`True`, *combine_scalars*=`True`)

Trigonometrically expand a `QubitOperatorList` representing 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, \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 `compress_scalars_as_product()`.

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` (default: `True`) – Set to `False` to skip checking whether all terms in each component operator commute.
- `combine_scalars` (default: `True`) – Set to `False` to disable combination of identity and zero terms.

Returns

`QubitOperatorList` – The trigonometrically expanded form of the input `QubitOperatorList`.

❶ 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 coefficients, 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, a value of "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]*) – *QubitOperators* which will comprise the terms of the generated *QubitOperatorList*.

- **symbol_format** (default: `r"term{ }"`) – A raw string containing one positional substitution (in which a numerical index will be placed), used to generate each symbolic coefficient.

Returns

`QubitOperatorList` – 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

`TypeVar(OperatorListT, bound= OperatorList)` – 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 ($N \times N$)matrix where :math: `N` is the number of operators in the `OperatorList`. Element indexed by [n, m] is `False` if operators indexed by 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.

Returns

`str` – A string representation of this instance.

map(*mapping*)

Updates right values of items 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.

operator_class

alias of `QubitOperator`

`parallelity_matrix`(*abs_tol*=1e-10)

Returns the “parallelity matrix” of the operator list.

This matrix is a boolean ($N \times N$) matrix where N is the number of operators in the `OperatorList`. Element indexed by $[n, m]$ is `True` if operators indexed by n and m are parallel - i.e. they are scalar multiples of each other, 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 “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 ($N \times N$) matrix where N is the number of operators in the `OperatorList`. Element indexed by $[n, m]$ is `True` if operators indexed by n and m qubit-wise commute, otherwise `False`. See `QubitOperator.qubitwise_commutates_with()` for further details on qubit-wise commutativity.

Note that individual `QubitOperator` objects 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 ($N \times N$) matrix where N is the number of operators in the `OperatorList`.

Element indexed by $[n, m]$ is `False` if operators indexed by n and m qubit-wise commute, otherwise `True`. See `QubitOperator.qubitwise_commutates_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, \hat{O}_i) , it is interpreted as $\prod_i e^{c_i \hat{O}_i}$. This method attempts to combine terms which are scalar multiples of one another. Each term is commuted backwards through the `QubitOperatorList` until it reaches an operator with which it does not commute with. If it encounters an operator which is a scalar 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 `self` represents a product of exponentials, with each constituent `Operator` corresponding to the exponentiated term of a single exponential in a 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 desired order of the final Trotter-Suzuki expansion. Currently, only a first order (*ABABAB...*) or second order (*ABBAABBA...*) expansion is supported.
- `initial_trotter_order` (`int`, default: 1) – The order of the original Trotter-Suzuki expansion used. 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.

n.b. the constructor's `data` argument expects (`coeff, operator`) ordering of the elements if it is passed as a `list`, but `self._list` is stored in (`operator, coeff`) ordering

Return type

`QubitOperatorList`

`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` instances - 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

A pair of `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 in the original object.

Parameters

`sublist_indices` (`list[int]`) – Indices of elements in this instance selected to constitute a new object.

Returns

`TypeVar(OperatorListT, bound= OperatorList)` – A sublist of this instance.

Raises

- `ValueError` – If `sublist_indices` contains indices not contained in this instance, or if this instance
- `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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

`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)` – Maps symbol-representing keys to the value the symbol should be substituted for.

Returns

`LinearListCombiner` – This instance with symbols key symbols replaced by their values.

`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 `sympify()`.
- `kwargs (Any)` – Kwargs to be passed to `sympify()`.

Returns

`LinearListCombiner` – Updated instance of `LinearListCombiner`.

Raises

`RuntimeError` – 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 `self` represents the exponential of a linear combination of `Operator` objects, each corresponding to a term in this linear combination. Trotterization is performed at the level of these `Operator` instances. The `Operator` objects 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. The first- and the second-order options are 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 (*precision*=15, *partial*=*False*)

Unsympifies dictionary values.

Replaces symbolic expressions with their corresponding numeric values.

Parameters

- **precision** (`int`, default: 15) – The number of decimal digits of precision used for evaluation.
- **partial** (`default: False`) – Set to `True` to allow partial unsympification where terms containing free symbols are present. By default, free symbols in any coefficient will cause an exception.

Returns

`LinearListCombiner` – Updated instance of `LinearListCombiner`.

Raises

`TypeError` – Unsimplification fails.

`untrotterize(trotter_number, trotter_order=1)`

Reverse a Trotter-Suzuki expansion given a product of exponentials as an `OperatorList`.

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 product of exponentials as an `OperatorList`, maintaining separation of exponents.

This method assumes that `self` represents a product of exponentials, with each constituent `Operator` corresponding to the exponentiated term of a single exponential within a product. Scalar factors within this `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`.

property `all_nontrivial_qubits: set[Qubit]`

Returns a set of all qubits acted upon by this string nontrivially (i.e. with an X,Y or Z).

`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 and operators, but the use of parametrised states and operators is untested.

For a `numpy.ndarray`, we delegate 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_QubitPauliString(qps)

Generates an instance of this class from a tket `QubitPauliString`.

Parameters

`qps` (`QubitPauliString`) – A `pytket.pauli.QubitPauliString` to be converted.

Returns

A converted Pauli string.

classmethod from_list(s)

Construct `QubitOperatorString` from list of tuples.

Parameters

`s` (`List[Tuple[Tuple[str, List[int]], str]]`) – `tuples` 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.

The 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.

❶ 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 is 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 from 0 to *self.num_qubits-1* 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 two-element *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.

❶ 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*.

`property qubit_id_list: List[Qubit]`

Return list of Qubits contained in a *QubitOperatorString*.

`property qubit_list: List[Qubit]`

Return list of Qubits contained in a *QubitOperatorString*.

`qubitwise_anticomutes_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_commutates_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.

`register_size(at_least_one=False)`

Return the size of the qubit register acted on by this string.

The maximum qubit index plus one.

Parameters

at_least_one (`bool`, default: False) – Always return at least 1. Relevant when the string is identity.

Returns

`int` – Number of qubits.

`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: Sequence[pytket._tket.unit_id.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_QubitPauliString()

Returns self converted to a tket QubitPauliString.

to_circuit(*n_qubits=None*)

Generate a circuit which implements the Pauli string as a set of single-qubit gates.

Parameters

n_qubits (`Optional[int]`, default: `None`) – Number of qubits in the circuit. If `None` the number of qubits will be inferred from the max qubit index in the string.

Returns

`Circuit` – A circuit which represents the Pauli string.

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.

❶ 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*) → list

A JSON-serializable representation of the QubitPauliString.

Returns

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 than `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: Sequence[pytket._tket.unit_id.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. $1\text{RDM}_{ij} = \langle a_{i,\alpha}^\dagger a_{j,\alpha} \rangle + \langle a_{i,\beta}^\dagger a_{j,\beta} \rangle$
- `rdms` (`Optional[ndarray]`, default: `None`) – Spin density matrix. If unspecified, spin symmetry is assumed and so the alpha and beta parts of the RDM exactly cancel. $1\text{RDM}_{ij}^s = \langle a_{i,\alpha}^\dagger a_{j,\alpha} \rangle - \langle a_{i,\beta}^\dagger a_{j,\beta} \rangle$

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) – 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) – Whether the results should 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) – 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. $2\text{RDM}_{ijkl} = \sum_{\sigma, \sigma' \in \{\alpha, \beta\}} \langle a_{i,\sigma}^\dagger a_{k,\sigma'}^\dagger a_{l,\sigma'} a_{j,\sigma} \rangle$

`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*

Represents a symmetry operator in a fermionic Hilbert space.

This is 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, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)

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.

capitalize()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()

Return a version of the string suitable for caseless comparisons.

center (width, fillchar=' ', /)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count (*sub*[*, start*[*, end*]]) → *int*

Return the number of non-overlapping occurrences of substring *sub* in string *S*[*start*:*end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

encode (*encoding*=‘utf-8’, *errors*=‘strict’)

Encode the string using the codec registered for encoding.

encoding

The encoding in which to encode the string.

errors

The error handling scheme to use for encoding errors. The default is ‘strict’ meaning that encoding errors raise a `UnicodeEncodeError`. Other possible values are ‘ignore’, ‘replace’ and ‘xmlcharrefreplace’ as well as any other name registered with `codecs.register_error` that can handle `UnicodeEncodeErrors`.

endswith (*suffix*[*, start*[*, end*]]) → *bool*

Return True if *S* ends with the specified suffix, False otherwise. With optional *start*, test *S* beginning at that position. With optional *end*, stop comparing *S* at that position. *suffix* can also be a tuple of strings 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 *S* where substring *sub* is found, such that *sub* is contained within *S*[*start*:*end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

Return -1 on failure.

format (**args*, ***kwargs*) → *str*

Return a formatted version of *S*, using substitutions from *args* and *kwargs*. The substitutions are identified by braces (‘{’ and ‘}’).

format_map (*mapping*) → *str*

Return a formatted version of *S*, using substitutions from *mapping*. The substitutions are identified by braces (‘{’ and ‘}’).

index (*sub*[*, start*[*, end*]]) → *int*

Return the lowest index in *S* where substring *sub* is found, such that *sub* is contained within *S*[*start*:*end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

Raises `ValueError` when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

`isdecimal()`

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

`isdigit()`

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

`isidentifier()`

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

`islower()`

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

`isnumeric()`

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

`isprintable()`

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

`isspace()`

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

`istitle()`

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

`isupper()`

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

`join(iterable, /)`

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: ‘.’.join(['ab', ‘pq’, ‘rs’]) -> ‘ab.pq.rs’

`ljust(width, fillchar=' ', /)`

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

`lower()`

Return a copy of the string converted to lowercase.

`lstrip(chars=None, /)`

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

`static maketrans()`

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

`partition(sep, /)`

Partition the string into three parts using the given separator.

This will search for the separator in the string. 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 string and two empty strings.

`removeprefix(prefix, /)`

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(prefix):]. Otherwise, return a copy of the original string.

`removesuffix(suffix, /)`

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:-len(suffix)]. Otherwise, return a copy of the original string.

`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 S where substring sub is found, such that sub is contained within S[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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust (width, fillchar=' ', /)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition (sep, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string, 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 strings and the original string.

rsplit (sep=None, maxsplit=-1)

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

rstrip (chars=None, /)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split (sep=None, maxsplit=-1)

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines (keepends=False)

Return a list of the lines in the string, 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 S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip (chars=None, /)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

swapcase ()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

title()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate(table, /)

Replace each character in the string using the given translation table.

table

Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via `__getitem__`, for instance a dictionary or list. If this operation raises `LookupError`, the character is left untouched. Characters mapped to None are deleted.

upper()

Return a copy of the string converted to uppercase.

zfill(width, /)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

apply_bra(fock_state)

Performs an operation on a `inquanto.states.FermionState` representing a bra.

This transforms the provided `inquanto.states.FermionState` with the operator on the right. Conjugation on the `inquanto.states.FermionState` object is not performed.

Parameters

- `fock_state` (`FermionState`) – Object representing a bra.

Returns

`FermionState` – A representation of the post-operation bra.

apply_ket(fock_state)

Performs an operation on a ket `inquanto.states.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 this instance and another `FermionOperator`.

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 against 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 of terms and will return zero only if all coefficients are zero.

Parameters

- **abs_tol** (`float`, default: `None`) – Tolerance for checking if coefficients are zero. Set to `None` to test using a standard

:param 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 its `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 its `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 its `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.

commutator(other_operator)

Returns the commutator of this instance 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 this instance 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 `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 ansatzes and digital quantum simulation. Otherwise, it may remove terms relevant to computation. Each expression is of the form $f(a_1, a_2, \dots, a_n)$ for some symbols a_i . $|f(a_1, a_2, \dots, a_n)|$ 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 its `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]]`) – 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.

❶ 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 annihilation 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.

❶ 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 its *dict*.

❶ 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 have complex values.

Returns

`bool` – `False` if a non-complex value occurs before any free symbols in the `dict` values, or `True` if no non-complex values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any non-complex values in the coefficients.

`is_all_coeff_imag()`

Check if all coefficients have purely imaginary values.

Returns

`bool` – `False` if a non-complex value occurs before any free symbols in the `dict` values, or `True` if no non-complex values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any non-imaginary values in the coefficients.

`is_all_coeff_real()`

Check if all coefficients have real values.

Returns

`bool` – `False` if a non-real value occurs before any free symbols in the `dict` values, or `True` if no non-real values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any non-real values in the `dict` coefficients.

`is_all_coeff_symbolic()`

Check if all coefficients contain free symbols.

Returns

`bool` – Whether all coefficients contain free symbols.

`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 anti-Hermitian, otherwise `False`.

Danger

This method uses the `normal_ordered()` method internally, which may be exponentially costly.

`is_any_coeff_complex()`

Check if any coefficients have complex values.

Returns

`bool` – `True` if a complex value occurs before any free symbols in the `dict` values, or `False` if no complex values occur.

Warning

Returns `None` if a free symbol occurs before any complex values in the coefficients.

`is_any_coeff_imag()`

Check if any coefficients have imaginary values.

Returns

`bool` – `True` if an imaginary value occurs before any free symbols in the `dict` values, or `False` if no imaginary values occur.

Warning

Returns `None` if a free symbol occurs before any imaginary values in the coefficients.

`is_any_coeff_real()`

Check if any coefficients have real values.

Returns

`bool` – `True` if a real value occurs before any free symbols in the `dict` values, or `False` if no real values occur.

Warning

Returns `None` if a free symbol occurs before any real values in the coefficients.

is_any_coeff_symbolic()

Check if any coefficients contain free symbols.

Returns

`bool` – Whether any coefficients contain free symbols.

is_commuting_operator()

Returns `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()

Returns whether or not term is in normal-ordered form.

The assumed convention for normal ordering is for all creation operators to be 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_normalized(`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_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 a molecular Hamiltonian.

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`, default: `False`) – Whether to check if operator conserves spin.

Returns

`bool` – True if operator conserves electron number (and, optionally, spin), `False` otherwise.

i 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 -
:param see `FermionOperatorString.from_string()` for more detail.:

Returns

`FermionOperatorString` – A generated `FermionOperatorString` instance.

list_class

alias of `FermionOperatorList`

make_hashable()

Return a hashable representation of the object.

Returns

`str` – A string representation of this instance.

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` – This instance.

property n_symbols: int

Returns the number of free symbols in the object.

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).

Danger

This may be exponential in runtime.

Returns

FermionOperator – The operator in normal-ordered form.

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 )
```

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.

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]]`) – Mapping from the old operator terms indices to the new ones. In case if a `list` is given, its indices acts as keys (old indices) and its values correspond to the new indices 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 class, of this instance.

Parameters

- **mapping** (*QubitMapping*, default: None) – *QubitMapping*-derived instance. Default mapping procedure is *QubitMappingJordanWigner*.
- **qubits** (*Optional[List[Qubit]]*, default: None) – The qubit register. If left as *None*, a default register will be assumed if possible. See *QubitMapping* documentation for further details.

Returns

QubitOperator – Mapped *QubitOperator* object.

remove_global_phase()

Returns a copy with a global phase applied such that the first element has a real coefficient.

This is an alias for *set_global_phase()* - see this method for greater control over the phase to be applied.

Returns

LinearDictCombiner – A copy of the object with a global phase applied such that the first element has a real coefficient.

reversed_order()

Reverses the order of terms and returns it as a new object.

Return type

LinearDictCombiner

set_global_phase (phase=0.0)

Returns a copy with a global phase applied such that the first element has the desired phase.

Parameters

- **phase** (*Union[int, float, complex, Expr]*, default: 0.0) – The phase to yield on the first element, in half-turns (i.e. multiples of pi).
- **phase.** (A symbolic expression can be assigned to)

Returns

LinearDictCombiner – A copy of the object with the desired global phase applied.

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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

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. \mathbb{Z}_2 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 `sympify()`.
- **kwargs** (`Any`) – Kwargs to be passed to `sympify()`.

Returns

`LinearDictCombiner` – Updated instance of `LinearDictCombiner`.

Raises

`RuntimeError` – Sympification fails.

property terms: List[Any]

Returns dictionary keys.

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{i}) f_{5} f_{5}^{\dagger} + -2.0\text{i} f_{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]),
   - ("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}\text{e}
\text{r} Z_{2}
>>> qo = QubitOperator({qos3: 3j})
>>> print(qo.to_latex(imaginary_unit="j", show_labels=True))
3.0j I^{\text{a}} Z^{\text{b}} Z^{\text{c}}
```

**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 instance with each element 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 supports values 1 (i.e. AB) and 2 (i.e. $A/2B/2B/2A/2$)
- **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 returned instance, with the coefficient of each inner Operator set to 1. This behaviour can be controlled with this argument - set to "outer" for default behaviour. On the other hand, setting this parameter to "inner" will store all scalars in the inner Operator coefficient, with the outer coefficients of the returned instance set to 1. Setting this parameter to "mixed" will store the Trotter factor in the outer coefficients of the returned instance, 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** (default: `1e-8`) – Threshold below which terms are removed.
- **normal_ordered** (default: `True`) – 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(*precision*=*15*, *partial*=*False*)

Unsympifies dictionary values.

Replaces symbolic expressions with their corresponding numeric values.

Parameters

- **precision** (`int`, default: 15) – The number of decimal digits of precision used for evaluation.
- **partial** (`bool`, default: `False`) – Set to `True` to allow partial unsympification where terms containing free symbols are present. By default, free symbols in any coefficient will cause an exception.

Returns

`LinearDictCombiner` – Updated instance of `LinearDictCombiner`.

Raises

`TypeError` – Unsympification fails.

classmethod zero()

Return object with a zero `dict` entry.

Examples

```
>>> print(LinearDictCombiner.zero())
()
```

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 those contained in the `operators` member. 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]]]`, default: `None`)
- **coeff** (`Union[int, float, complex, Expr]`, default: `1.0`)

class CompressScalarsBehavior(value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)

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.

`capitalize()`

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()
 Return a version of the string suitable for caseless comparisons.

center(width, fillchar=' ', /)
 Return a centered string of length width.
 Padding is done using the specified fill character (default is a space).

count(sub[, start[, end]]) → int
 Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode(encoding='utf-8', errors='strict')
 Encode the string using the codec registered for encoding.

encoding
 The encoding in which to encode the string.

errors
 The error handling scheme to use for encoding errors. The default is ‘strict’ meaning that encoding errors raise a UnicodeEncodeError. Other possible values are ‘ignore’, ‘replace’ and ‘xmlcharrefreplace’ as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(suffix[, start[, end]]) → bool
 Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings 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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.
 Return -1 on failure.

format(*args, **kwargs) → str
 Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(mapping) → str
 Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index(sub[, start[, end]]) → int
 Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.
 Raises ValueError when the substring is not found.

isalnum()
 Return True if the string is an alpha-numeric string, False otherwise.
 A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

`isalpha()`

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

`isascii()`

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

`isdecimal()`

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

`isdigit()`

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

`isidentifier()`

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

`islower()`

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

`isnumeric()`

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

`isprintable()`

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

`isspace()`

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

`istitle()`

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

`isupper()`

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join(*iterable*, /)

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

ljust(*width*, *fillchar*='', /)

Return a left-justified string of length *width*.

Padding is done using the specified fill character (default is a space).

lower()

Return a copy of the string converted to lowercase.

lstrip(*chars*=None, /)

Return a copy of the string with leading whitespace removed.

If *chars* is given and not None, remove characters in *chars* instead.

static maketrans()

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in *x* will be mapped to the character at the same position in *y*. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string. 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 string and two empty strings.

removeprefix(*prefix*, /)

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(prefix):]. Otherwise, return a copy of the original string.

removesuffix(*suffix*, /)

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:-len(suffix)]. Otherwise, return a copy of the original string.

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 *S* where substring *sub* is found, such that *sub* is contained within *S*[*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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust(*width*, *fillchar*='', /)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string, 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 strings and the original string.

rsplit(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

rstrip(*chars=None*, /)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split(*sep=None*, *maxsplit=-1*)

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines(*keepends=False*)

Return a list of the lines in the string, 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 S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip(*chars=None*, /)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

swapcase()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

title()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate(table, /)

Replace each character in the string using the given translation table.

table

Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via `__getitem__`, for instance a dictionary or list. If this operation raises `LookupError`, the character is left untouched. Characters mapped to None are deleted.

upper()

Return a copy of the string converted to uppercase.

zfill(width, /)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

class FactoryCoefficientsLocation(value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)

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.

capitalize()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()

Return a version of the string suitable for caseless comparisons.

center(width, fillchar=' ', /)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count(sub[, start[, end]]) → int

Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode(encoding='utf-8', errors='strict')

Encode the string using the codec registered for encoding.

encoding

The encoding in which to encode the string.

errors

The error handling scheme to use for encoding errors. The default is ‘strict’ meaning that encoding errors raise a `UnicodeEncodeError`. Other possible values are ‘ignore’, ‘replace’ and ‘xmlcharrefreplace’ as well as any other name registered with `codecs.register_error` that can handle `UnicodeEncodeErrors`.

`endswith(suffix[, start[, end]]) → bool`

Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings 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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

`format(*args, **kwargs) → str`

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces (‘{’ and ‘}’).

`format_map(mapping) → str`

Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces (‘{’ and ‘}’).

`index(sub[, start[, end]]) → int`

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises `ValueError` when the substring is not found.

`isalnum()`

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

`isalpha()`

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

`isascii()`

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

`isdecimal()`

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

`isdigit()`

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

`isidentifier()`

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

`islower()`

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

`isnumeric()`

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

`isprintable()`

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

`isspace()`

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

`istitle()`

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

`isupper()`

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

`join(iterable, /)`

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: ‘.’.join(['ab', ‘pq’, ‘rs’]) -> ‘ab.pq.rs’

`ljust(width, fillchar=' ', /)`

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

`lower()`

Return a copy of the string converted to lowercase.

`lstrip(chars=None, /)`

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

`static maketrans()`

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

`partition(sep, /)`

Partition the string into three parts using the given separator.

This will search for the separator in the string. 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 string and two empty strings.

`removeprefix(prefix, /)`

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(prefix):]. Otherwise, return a copy of the original string.

`removesuffix(suffix, /)`

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:-len(suffix)]. Otherwise, return a copy of the original string.

`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 S where substring sub is found, such that sub is contained within S[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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

`rjust(width, fillchar=' ', /)`

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

`rpartition(sep, /)`

Partition the string into three parts using the given separator.

This will search for the separator in the string, 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 strings and the original string.

`rsplit(sep=None, maxsplit=-1)`

Return a list of the substrings in the string, using sep as the separator string.

`sep`

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

`maxsplit`

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

`rstrip(chars=None, /)`

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

`split(sep=None, maxsplit=-1)`

Return a list of the substrings in the string, using sep as the separator string.

`sep`

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

`maxsplit`

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

`splitlines(keepends=False)`

Return a list of the lines in the string, 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 S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

`strip(chars=None, /)`

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

`swapcase()`

Convert uppercase characters to lowercase and lowercase characters to uppercase.

`title()`

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate (table, /)

Replace each character in the string using the given translation table.

table

Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via `__getitem__`, for instance a dictionary or list. If this operation raises `LookupError`, the character is left untouched. Characters mapped to None are deleted.

upper ()

Return a copy of the string converted to uppercase.

zfill (width, /)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

clone ()

Performs shallow copy of the object.

Return type

`TypeVar(SYMBOLICTYPE, bound= Symbolic)`

collapse_as_linear_combination (ignore_outer_coefficients=False)

Treating this instance as a linear combination, return it in the form of an `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 constituent `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 this instance, multiplied by their associated coefficients if requested.

collapse_as_product (reverse=False, ignore_outer_coefficients=False)

Treating this instance 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 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)]

```

copy()

Returns a deep copy of this instance.

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 coefficients, 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, a value of "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

`TypeVar(OperatorListT, bound= OperatorList)` – 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.

Returns

`str` – A string representation of this instance.

map(mapping)

Updates right values of items 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.

property num_spin_orbs: int

Return the number of spin-orbitals that this operator explicitly acts on.

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 `self` represents a product of exponentials, with each constituent `Operator` corresponding to the exponentiated term of a single exponential in a 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 desired order of the final Trotter-Suzuki expansion. Currently, only a first order ($ABABAB\dots$) or second order ($ABBAABBA\dots$) expansion is supported.
- **`initial_trotter_order`** (`int`, default: 1) – The order of the original Trotter-Suzuki expansion used. Currently, only a first order ($ABABAB\dots$) 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 in the original object.

Parameters

`sublist_indices` (`list[int]`) – Indices of elements in this instance selected to constitute a new object.

Returns

`TypeVar(OperatorListT, bound=OperatorList)` – A sublist of this instance.

Raises

- `ValueError` – If `sublist_indices` contains indices not contained in this instance, or if this instance
- `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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

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`) – Maps symbol-representing keys to the value the symbol should be substituted for.

Returns

`LinearListCombiner` – This instance with symbols key symbols replaced by their values.

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. \mathbb{Z}_2 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 `sympify()`.
- **kwargs** (`Any`) – Kwargs to be passed to `sympify()`.

Returns

`LinearListCombiner` – Updated instance of `LinearListCombiner`.

Raises

`RuntimeError` – Sympification fails.

to_symmetry_operator_fermionic()

Convert to a *SymmetryOperatorFermionic*.

Danger

This may scale exponentially depending on the operator!

Returns

SymmetryOperatorFermionic – The expanded form of the symmetry 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 `self` represents the exponential of a linear combination of `Operator` objects, each corresponding to a term in this linear combination. Trotterization is performed at the level of these `Operator` instances. The `Operator` objects 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. The first- and the second-order options are 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 (*precision*=15, *partial*=False)

Unsympifies dictionary values.

Replaces symbolic expressions with their corresponding numeric values.

Parameters

- **precision** (`int`, default: 15) – The number of decimal digits of precision used for evaluation.
- **partial** (default: `False`) – Set to True to allow partial unsympification where terms containing free symbols are present. By default, free symbols in any coefficient will cause an exception.

Returns

`LinearListCombiner` – Updated instance of `LinearListCombiner`.

Raises

`TypeError` – Unsympification fails.

untrotterize (*trotter_number*, *trotter_order*=1)

Reverse a Trotter-Suzuki expansion given a product of exponentials as an `OperatorList`.

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 product of exponentials as an `OperatorList`, maintaining separation of exponents.

This method assumes that `self` represents a product of exponentials, with each constituent `Operator` corresponding to the exponentiated term of a single exponential within a product. Scalar factors within this `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 ($ABABAB\dots$) 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`

Represents a symmetry operator in a qubit Hilbert space.

This is 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, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)
```

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.

capitalize()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()

Return a version of the string suitable for caseless comparisons.

center(width, fillchar=' ', /)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count(sub[, start[, end]]) → int

Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode(encoding='utf-8', errors='strict')

Encode the string using the codec registered for encoding.

encoding

The encoding in which to encode the string.

errors

The error handling scheme to use for encoding errors. The default is ‘strict’ meaning that encoding errors raise a `UnicodeEncodeError`. Other possible values are ‘ignore’, ‘replace’ and ‘xmlcharrefreplace’ as well as any other name registered with `codecs.register_error` that can handle `UnicodeEncodeErrors`.

endswith(suffix[, start[, end]]) → bool

Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings 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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

`format (*args, **kwargs) → str`

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

`format_map (mapping) → str`

Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

`index (sub[, start[, end]]) → int`

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

`isalnum ()`

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

`isalpha ()`

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

`isascii ()`

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

`isdecimal ()`

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

`isdigit ()`

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

`isidentifier ()`

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

`islower()`

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

`isnumeric()`

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

`isprintable()`

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in `repr()` or if it is empty.

`isspace()`

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

`istitle()`

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

`isupper()`

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

`join(iterable, /)`

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

`ljust(width, fillchar=' ', /)`

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

`lower()`

Return a copy of the string converted to lowercase.

`lstrip(chars=None, /)`

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

`static maketrans()`

Return a translation table usable for `str.translate()`.

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will

be mapped to the character at the same position in *y*. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string. 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 string and two empty strings.

removeprefix(*prefix*, /)

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return string[len(*prefix*) :]. Otherwise, return a copy of the original string.

removesuffix(*suffix*, /)

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return string[:len(*suffix*)]. Otherwise, return a copy of the original string.

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 *S* where substring *sub* is found, such that *sub* is contained within *S*[*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 *S* where substring *sub* is found, such that *sub* is contained within *S*[*start*:*end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust(*width*, *fillchar*='', /)

Return a right-justified string of length *width*.

Padding is done using the specified fill character (default is a space).

rpartition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string, 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 strings and the original string.

rsplit(*sep*=None, *maxsplit*=-1)

Return a list of the substrings in the string, using *sep* as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

rstrip (chars=None, /)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split (sep=None, maxsplit=-1)

Return a list of the substrings in the string, using sep as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines (keepends=False)

Return a list of the lines in the string, 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 S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip (chars=None, /)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

swapcase ()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

title ()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate (table, /)

Replace each character in the string using the given translation table.

table

Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via `__getitem__`, for instance a dictionary or list. If this operation raises `LookupError`, the character is left untouched. Characters mapped to None are deleted.

upper ()

Return a copy of the string converted to uppercase.

zfill (width, /)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

property all_nontrivial_qubits: set[Qubit]

Returns a set of all qubits acted upon by this operator nontrivially (i.e. with an X,Y or Z).

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. `:rtype: QubitOperator`

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)
```

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` – 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 against 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 of terms and will return zero only if all coefficients are zero.

Parameters

- **abs_tol** (`float`, default: `None`) – Tolerance for checking if coefficients are zero. Set to `None` to test using a standard

:param 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.

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.

Return type

`None`

⚠ Warning

When `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 ansatzes and digital quantum simulation. Otherwise, it may remove terms relevant to computation. Each expression is of the form $f(a_1, a_2, \dots, a_n)$ for some symbols a_i . $|f(a_1, a_2, \dots, a_n)|$ is assumed to monotonically increase in both real and imaginary components for all $a_i \in [0, 1]$.

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 and operators, but the use of parametrised states and operators is untested. In this case, the optional *qubits* parameter is ignored.

For a `numpy.ndarray`, we delegate 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.

eigenspectrum(hamming_weight=None, nroots=None, threshold=1e-5, check_hermitian=False, check_hermitian_atol=1e-10)

Returns the eigenspectrum of a Hermitian operator, optionally filtered by a given Hamming weight.

More precisely, if *hamming_weight* is provided, only those eigenvalues whose eigenstates' computational components have coefficients larger than *threshold* and match the provided Hamming weight will be returned.

If argument *nroots* is provided, an iterative sparse matrix diagonalisation procedure is invoked. Otherwise, the whole dense matrix is diagonalised.

Notes

If this operator results from a Jordan-Wigner fermion-qubit encoding, filtering by Hamming weight corresponds to particle conservation. The operator is assumed to be Hermitian.

Warning

This method scales exponentially. Use only for testing on small systems.

Parameters

- `hamming_weight` (`Optional[int]`, default: `None`) – Hamming weight for filtering the roots.
- `nroots` (`Optional[int]`, default: `None`) – How many roots to calculate (invokes iterative diagonalisation).
- `threshold` (`float`, default: `1e-5`) – State coefficient threshold for checking Hamming weight.
- `check_hermitian` (`bool`, default: `False`) – Whether to check the hermiticity of the operator. Uses `is_hermitian()`.
- `check_hermitian_atol` (`float`, default: `1e-10`) – Absolute tolerance for hermiticity check. Passed to `is_hermitian()`.

Returns

`ndarray` – Array of (filtered) eigenvalues.

`empty()`

Checks if dictionary is empty.

Return type

`bool`

`ensure_hermitian(tolerance=1e-10)`

Eliminate all insignificant imaginary parts of numeric coefficients.

Raises

`ValueError` – imaginary symbolic, or significant imaginary numeric coefficient.

Parameters

- `tolerance` (`float`, default: `1e-10`) – Threshold determining whether a numerical imag coefficient is significant.

Returns

`QubitOperator` – This instance.

`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` – 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` – 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_QubitPauliOperator`(*qop*, *unsympify_coefficients*=True)

Generate an instance of this class from a tket `QubitPauliOperator`.

Component `QubitPauliStrings` will be converted to `QubitOperatorStrings`. By default, coefficients without free symbols will be converted to float (if real) or complex (otherwise). To disable this, set `unsympify_coefficients = False`. For finer grained control over unsympification, generate with `unsympify_coefficients = False` and `unsympify` after generation. See `unsympify()` for further details.

Parameters

- **qop** (QubitPauliOperator) – The QubitPauliOperator to be converted.
- **unsympify_coefficients** (bool, default: True) – Whether to cast coefficients to standard python float/complex. Set to False to disable.

Returns

The converted operator.

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)

Get the coefficient value of the provided Pauli string.

Parameters

key (QubitPauliString)

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.

❶ 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. :rtype: *QubitOperator*

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)
```

classmethod identity()

Return an identity operator. :rtype: *QubitOperator*

Examples

```
>>> print(QubitOperator.identity())
(1.0, )
```

is_all_coeff_complex()

Check if all coefficients have complex values.

Returns

`bool` – False if a non-complex value occurs before any free symbols in the `dict` values, or True if no non-complex values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any non-complex values in the coefficients.

`is_all_coeff_imag()`

Check if all coefficients have purely imaginary values.

Returns

`bool` – `False` if a non-complex value occurs before any free symbols in the `dict` values, or `True` if no non-complex values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any non-imaginary values in the coefficients.

`is_all_coeff_real()`

Check if all coefficients have real values.

Returns

`bool` – `False` if a non-real value occurs before any free symbols in the `dict` values, or `True` if no non-real values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any non-real values in the `dict` coefficients.

`is_all_coeff_symbolic()`

Check if all coefficients contain free symbols.

Returns

`bool` – Whether all coefficients contain free symbols.

`is_antihermitian(tolerance=1e-10)`

Check if operator is anti-Hermitian (purely imaginary coefficients).

Check is performed by looking for symbolic or significant numeric real part in at least one coefficient

Returns

`bool` – `True` if anti-Hermitian, `False` otherwise.

Parameters

`tolerance` (`float`, default: `1e-10`)

`is_any_coeff_complex()`

Check if any coefficients have complex values.

Returns

`bool` – `True` if a complex value occurs before any free symbols in the `dict` values, or `False` if no complex values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any complex values in the coefficients.

is_any_coeff_imag()

Check if any coefficients have imaginary values.

Returns

`bool` – True if an imaginary value occurs before any free symbols in the `dict` values, or `False` if no imaginary values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any imaginary values in the coefficients.

is_any_coeff_real()

Check if any coefficients have real values.

Returns

`bool` – True if a real value occurs before any free symbols in the `dict` values, or `False` if no real values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any real values in the coefficients.

is_any_coeff_symbolic()

Check if any coefficients contain free symbols.

Returns

`bool` – Whether any coefficients contain free symbols.

is_commuting_operator()

Returns `True` if every term in operator commutes with every other term, otherwise `False`.

Return type

`bool`

is_hermitian(tolerance=1e-10)

Check if operator is Hermitian.

Check is performed by looking for symbolic or significant numeric imaginary part in at least one coefficient

Returns

`bool` – True if Hermitian, `False` otherwise.

Parameters

`tolerance (float, default: 1e-10)`

static is_hermitian_coeff(coef, tolerance=1e-10)

Determine whether the given coefficient can be present in a Hermitian `QubitOperator`.

Parameters

- `coeff (Union[int, float, complex, Expr])` – Coefficient to check
- `tolerance (float, default: 1e-10)` – Threshold determining whether a numerical imaginary coefficient is significant.

Returns

`bool` – Whether coeff can be present in a Hermitian `QubitOperator`.

is_normalized(*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_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` (default: `1e-10`) – 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.

Returns`str` – A string representation of this instance.`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` – This instance.

property n_symbols: int

Returns the number of free symbols in the object.

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 on $[0, N)$, 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.

Return type

`None`

```
>>> 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)
```

`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 on $[0, N]$, 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.

print_table()

Print dictionary formatted as a table.

Return type

NoReturn

qubitwise_anticomutes_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 *QubitOperator* objects 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` – 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` – either operator consists of more than a single term.

remove_global_phase()

Returns a copy with a global phase applied such that the first element has a real coefficient.

This is an alias for `set_global_phase()` - see this method for greater control over the phase to be applied.

Returns

`LinearDictCombiner` – A copy of the object with a global phase applied such that the first element has a real coefficient.

`reversed_order()`

Reverses the order of terms and returns it as a new object.

Return type

`LinearDictCombiner`

`set_global_phase(phase=0.0)`

Returns a copy with a global phase applied such that the first element has the desired phase.

Parameters

- **phase** (`Union[int, float, complex, Expr]`, default: `0.0`) – The phase to yield on the first element, in half-turns (i.e. multiples of π).
- **phase.** (*A symbolic expression can be assigned to*)

Returns

`LinearDictCombiner` – A copy of the object with the desired global phase applied.

`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 and operators, but the use of parametrised states and operators is untested. For a `numpy.ndarray`, we delegate 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*)

Substitution for symbolic expressions.

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` – A new `QubitOperator` object with symbols substituted.

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. \mathbb{Z}_2 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` – Sympification fails.

`symplectic_representation(qubits=None)`

Generate the symplectic representation of the operator.

This is a binary ($M \times 2N$) matrix where M is the number of terms and N is the number of qubits. In the leftmost half of the matrix, the element indexed by $[i, j]$ 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 indexed by $[i, j]$ is `True` where qubit $j - \text{num_qubits}$ 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

Raises

`ValueError` – operator contains qubits which are not in the provided `qubits` argument.

`property terms: List[Any]`

Returns dictionary keys.

`to_QubitPauliOperator()`

Converts this object to a tket `QubitPauliOperator`.

Component `QubitOperatorStrings` will be converted to tket `QubitPauliStrings`. Note that coefficients will implicitly be converted to `sympy` objects in accordance with the `QubitPauliOperator` API.

Returns

The converted operator.

`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} 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} + (0.5-8.0\text{j}) f_{5}^{\dagger} + 2.0\text{j} f_{1}^{\dagger}
-> f_{1}^{\dagger} + (0.5-8.0\text{j}) f_{5}^{\dagger} + 2.0\text{j} 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)
(a, X0 Y1), (re(c), Z0 Z1)
>>> print(antiherm_qo)
(1.0*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 *QubitOperator*s. 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

Tuple[QubitOperator, QubitOperator] – 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 instance with each element 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 supports values 1 (i.e. AB) and 2 (i.e. $A/2 B/2 B/2 A/2$)
- **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 returned instance, with the coefficient of each inner `Operator` set to 1. This behaviour can be controlled with this argument - set to "outer" for default behaviour. On the other hand, setting this parameter to "inner" will store all scalars in the inner `operator` coefficient, with the outer coefficients of the returned instance set to 1. Setting this parameter to "mixed" will store the Trotter factor in the outer coefficients of the returned instance, 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 (*precision*=15, *partial*=False)

Unsympifies dictionary values.

Replaces symbolic expressions with their corresponding numeric values.

Parameters

- ***precision*** (`int`, default: 15) – The number of decimal digits of precision used for evaluation.
- ***partial*** (`bool`, default: `False`) – Set to `True` to allow partial unsympification where terms containing free symbols are present. By default, free symbols in any coefficient will cause an exception.

Returns

`LinearDictCombiner` – Updated instance of `LinearDictCombiner`.

Raises

`TypeError` – Unsympification fails.

classmethod zero()

Return object with a zero `dict` entry.

Examples

```
>>> print(LinearDictCombiner.zero())
()
```

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 the those contained in the `operators` member. 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]]]`, 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, names=None, *, module=None, qualname=None, type=None,
                                start=1, boundary=None)
```

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.

capitalize()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()

Return a version of the string suitable for caseless comparisons.

center(width, fillchar=' ', /)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count(sub[, start[, end]]) → int

Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode(encoding='utf-8', errors='strict')

Encode the string using the codec registered for encoding.

encoding

The encoding in which to encode the string.

errors

The error handling scheme to use for encoding errors. The default is ‘strict’ meaning that encoding errors raise a `UnicodeEncodeError`. Other possible values are ‘ignore’, ‘replace’ and ‘xmlcharrefreplace’ as well as any other name registered with `codecs.register_error` that can handle `UnicodeEncodeErrors`.

endswith(suffix[, start[, end]]) → bool

Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings 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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(*args, **kwargs) → str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(mapping) → str

Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index(sub[, start[, end]]) → int

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

`isprintable()`

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

`isspace()`

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

`istitle()`

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

`isupper()`

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

`join(iterable, /)`

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

`ljust(width, fillchar=' ', /)`

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

`lower()`

Return a copy of the string converted to lowercase.

`lstrip(chars=None, /)`

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

`static maketrans()`

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

`partition(sep, /)`

Partition the string into three parts using the given separator.

This will search for the separator in the string. 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 string and two empty strings.

`removeprefix(prefix, /)`

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return `string[len(prefix):]`. Otherwise, return a copy of the original string.

`removesuffix(suffix, /)`

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return `string[:-len(suffix)]`. Otherwise, return a copy of the original string.

`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 S where substring sub is found, such that sub is contained within S[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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

`rjust(width, fillchar=' ', /)`

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

`rpartition(sep, /)`

Partition the string into three parts using the given separator.

This will search for the separator in the string, 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 strings and the original string.

`rsplit(sep=None, maxsplit=-1)`

Return a list of the substrings in the string, using sep as the separator string.

`sep`

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

`maxsplit`

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

`rstrip(chars=None, /)`

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split (sep=None, maxsplit=-1)
 Return a list of the substrings in the string, using sep as the separator string.

sep
 The separator used to split the string.
 When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit
 Maximum number of splits (starting from the left). -1 (the default value) means no limit.
 Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines (keepends=False)
 Return a list of the lines in the string, 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 S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip (chars=None, /)
 Return a copy of the string with leading and trailing whitespace removed.
 If chars is given and not None, remove characters in chars instead.

swapcase ()
 Convert uppercase characters to lowercase and lowercase characters to uppercase.

title ()
 Return a version of the string where each word is titlecased.
 More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate (table, /)
 Replace each character in the string using the given translation table.

table
 Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.
 The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

upper ()
 Return a copy of the string converted to uppercase.

zfill (width, /)
 Pad a numeric string with zeros on the left, to fill a field of the given width.
 The string is never truncated.

class ExpandExponentialProductCoefficientsBehavior (value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)
 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.

capitalize()
Return a capitalized version of the string.
More specifically, make the first character have upper case and the rest lower case.

casefold()
Return a version of the string suitable for caseless comparisons.

center(width, fillchar=' ', /)
Return a centered string of length width.
Padding is done using the specified fill character (default is a space).

count(sub[, start[, end]]) → int
Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode(encoding='utf-8', errors='strict')
Encode the string using the codec registered for encoding.

encoding
The encoding in which to encode the string.

errors
The error handling scheme to use for encoding errors. The default is ‘strict’ meaning that encoding errors raise a UnicodeEncodeError. Other possible values are ‘ignore’, ‘replace’ and ‘xmlcharrefreplace’ as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(suffix[, start[, end]]) → bool
Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings 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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.
Return -1 on failure.

format (*args, **kwargs) → str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map (mapping) → str

Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index (sub[, start[, end]]) → int

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

`isprintable()`

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

`isspace()`

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

`istitle()`

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

`isupper()`

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

`join(iterable, /)`

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

`ljust(width, fillchar=' ', /)`

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

`lower()`

Return a copy of the string converted to lowercase.

`lstrip(chars=None, /)`

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

`static maketrans()`

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

`partition(sep, /)`

Partition the string into three parts using the given separator.

This will search for the separator in the string. 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 string and two empty strings.

`removeprefix(prefix, /)`

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return `string[len(prefix):]`. Otherwise, return a copy of the original string.

`removesuffix(suffix, /)`

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return `string[:-len(suffix)]`. Otherwise, return a copy of the original string.

`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 S where substring sub is found, such that sub is contained within S[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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

`rjust(width, fillchar=' ', /)`

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

`rpartition(sep, /)`

Partition the string into three parts using the given separator.

This will search for the separator in the string, 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 strings and the original string.

`rsplit(sep=None, maxsplit=-1)`

Return a list of the substrings in the string, using sep as the separator string.

`sep`

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

`maxsplit`

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

`rstrip(chars=None, /)`

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split (sep=None, maxsplit=-1)
 Return a list of the substrings in the string, using sep as the separator string.

sep
 The separator used to split the string.
 When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit
 Maximum number of splits (starting from the left). -1 (the default value) means no limit.
 Note, str.split() is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines (keepends=False)
 Return a list of the lines in the string, 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 S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip (chars=None, /)
 Return a copy of the string with leading and trailing whitespace removed.
 If chars is given and not None, remove characters in chars instead.

swapcase ()
 Convert uppercase characters to lowercase and lowercase characters to uppercase.

title ()
 Return a version of the string where each word is titlecased.
 More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate (table, /)
 Replace each character in the string using the given translation table.

table
 Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.
 The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

upper ()
 Return a copy of the string converted to uppercase.

zfill (width, /)
 Pad a numeric string with zeros on the left, to fill a field of the given width.
 The string is never truncated.

class FactoryCoefficientsLocation (value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)
 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.

capitalize()
Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()
Return a version of the string suitable for caseless comparisons.

center(width, fillchar=' ', /)
Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count(sub[, start[, end ]]) → int
Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode(encoding='utf-8', errors='strict')
Encode the string using the codec registered for encoding.

encoding
The encoding in which to encode the string.

errors
The error handling scheme to use for encoding errors. The default is ‘strict’ meaning that encoding errors raise a UnicodeEncodeError. Other possible values are ‘ignore’, ‘replace’ and ‘xmlcharrefreplace’ as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(suffix[, start[, end ]]) → bool
Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings 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 S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(*args, **kwargs) → str
Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(mapping) → str
Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').
```

index(*sub*[, *start*[, *end*]]) → **int**

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

`istitle()`

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

`isupper()`

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

`join(iterable, /)`

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: `'.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'`

`ljust(width, fillchar=' ', /)`

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

`lower()`

Return a copy of the string converted to lowercase.

`lstrip(chars=None, /)`

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

`static maketrans()`

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

`partition(sep, /)`

Partition the string into three parts using the given separator.

This will search for the separator in the string. 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 string and two empty strings.

`removeprefix(prefix, /)`

Return a str with the given prefix string removed if present.

If the string starts with the prefix string, return `string[len(prefix):]`. Otherwise, return a copy of the original string.

`removesuffix(suffix, /)`

Return a str with the given suffix string removed if present.

If the string ends with the suffix string and that suffix is not empty, return `string[:-len(suffix)]`. Otherwise, return a copy of the original string.

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 *S* where substring *sub* is found, such that *sub* is contained within *S*[*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 *S* where substring *sub* is found, such that *sub* is contained within *S*[*start*:*end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

Raises *ValueError* when the substring is not found.

rjust(*width*, *fillchar*='', /)

Return a right-justified string of length *width*.

Padding is done using the specified fill character (default is a space).

rpartition(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string, 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 strings and the original string.

rsplit(*sep*=None, *maxsplit*=-1)

Return a list of the substrings in the string, using *sep* as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Splitting starts at the end of the string and works to the front.

rstrip(*chars*=None, /)

Return a copy of the string with trailing whitespace removed.

If *chars* is given and not None, remove characters in *chars* instead.

split(*sep*=None, *maxsplit*=-1)

Return a list of the substrings in the string, using *sep* as the separator string.

sep

The separator used to split the string.

When set to None (the default value), will split on any whitespace character (including \n \r \t \f and spaces) and will discard empty strings from the result.

maxsplit

Maximum number of splits (starting from the left). -1 (the default value) means no limit.

Note, *str.split()* is mainly useful for data that has been intentionally delimited. With natural text that includes punctuation, consider using the regular expression module.

splitlines (keepends=False)
 Return a list of the lines in the string, 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 S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip (chars=None, /)
 Return a copy of the string with leading and trailing whitespace removed.
 If chars is given and not None, remove characters in chars instead.

swapcase ()
 Convert uppercase characters to lowercase and lowercase characters to uppercase.

title ()
 Return a version of the string where each word is titlecased.
 More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate (table, /)
 Replace each character in the string using the given translation table.
table
 Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.
 The table must implement lookup/indexing via `__getitem__`, for instance a dictionary or list. If this operation raises `LookupError`, the character is left untouched. Characters mapped to None are deleted.

upper ()
 Return a copy of the string converted to uppercase.

zfill (width, /)
 Pad a numeric string with zeros on the left, to fill a field of the given width.
 The string is never truncated.

property all_nontrivial_qubits: set[Qubit]
 Returns a set of all qubits acted upon by the operators in this `QubitOperatorList` nontrivially (i.e. with an X, Y or Z).

property all_qubits: Set[Qubit]
 Returns a set of all qubits included in any operator in the `QubitOperatorList`.

build_subset (indices)
 Builds a subset of the class based on a list of indices passed by a user.

Parameters
`indices (List[int])` – Indices of the items which are selected to for the subset as a new `QubitOperatorList` instance.

Raises
`ValueError` – the number of requested indices is larger than the number of elements in this instance.

Returns
`QubitOperatorList` – Subset of terms of the initial object.

clone()

Performs shallow copy of the object.

Return type

`TypeVar(SYMBOLICTYPE, bound= Symbolic)`

collapse_as_linear_combination(*ignore_outer_coefficients=False*)

Treating this instance as a linear combination, return it in the form of an `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 constituent `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 this instance, multiplied by their associated coefficients if requested.

collapse_as_product(*reverse=False, ignore_outer_coefficients=False*)

Treating this instance 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 ($N \times N$) matrix where :math: `N` is the number of operators in the `OperatorList`. Element indexed by [n, m] is `True` if operators indexed by 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 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 resultant matrix has a shape (N, M) and :math:`J_{i,j} = \frac{\partial c_i}{\partial \theta_j}

Parameters

- `symbols` (`List[Symbol]`) – Symbols w.r.t. which each coefficient is differentiated.
- `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 the form `(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 this instance.

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 delegate 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 delegate 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 ($N \times N$) matrix where N is the number of operators in the `OperatorList`. Element indexed by $[n, m]$ is `True` if operators indexed by 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=ExpandExponentialProductCoefficientsBehavior`,
`additional_exponent=1.0`,
`check_commuting=True`,
`combine_scalars=True`)

Trigonometrically expand a `QubitOperatorList` representing 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 $(c_i; \text{math}: \hat{O}_i)$, 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 `compress_scalars_as_product()`.

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** (default: `True`) – Set to `False` to skip checking whether all terms in each component operator commute.
- **combine_scalars** (default: `True`) – 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 coefficients, 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, a value of "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]`) – `QubitOperators` which will comprise the terms of the generated `QubitOperatorList`.
- `symbol_format` (default: `r"term{ }"`) – A raw string containing one positional substitution (in which a numerical index will be placed), used to generate each symbolic coefficient.

Returns

`QubitOperatorList` – 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

`TypeVar(OperatorListT, bound= OperatorList)` – 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 $(N \times N)$ matrix where :math: `N is the number of operators in the `OperatorList`. Element indexed by $[n, m]$ is `False` if operators indexed by 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 \mathbb{Z}_2 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.

Returns

`str` – A string representation of this instance.

map (mapping)

Updates right values of items 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.

operator_class

alias of `QubitOperator`

parallelity_matrix (abs_tol=1e-10)

Returns the “parallelity matrix” of the operator list.

This matrix is a boolean ($N \times N$) matrix where N is the number of operators in the `OperatorList`. Element indexed by $[n, m]$ is `True` if operators indexed by n and m are parallel - i.e. they are scalar multiples of each other, 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 “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 ($N \times N$) matrix where N is the number of operators in the `OperatorList`. Element indexed by $[n, m]$ is `True` if operators indexed by n and m qubit-wise commute, otherwise `False`. See `QubitOperator.qubitwise_commutates_with()` for further details on qubit-wise commutativity.

Note that individual `QubitOperator` objects 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 ($N \times N$) matrix where N is the number of operators in the `OperatorList`.

Element indexed by $[n, m]$ is `False` if operators indexed by n and m qubit-wise commute, otherwise `True`. See `QubitOperator.qubitwise_commutates_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 $(c_i : \text{math:hat}{O}_i)$, it is interpreted as $\prod_i e^{c_i \hat{O}_i}$. This method attempts to combine terms which are scalar multiples of one another. Each term is commuted backwards through the `QubitOperatorList` until it reaches an operator with which it does not commute with. If it encounters an operator which is a scalar 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 `self` represents a product of exponentials, with each constituent `Operator` corresponding to the exponentiated term of a single exponential in a 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 desired order of the final Trotter-Suzuki expansion. Currently, only a first order (*ABABAB...*) or second order (*ABBAABBA...*) expansion is supported.
- `initial_trotter_order` (`int`, default: 1) – The order of the original Trotter-Suzuki expansion used. 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.

n.b. the constructor's `data` argument expects (coeff, operator) ordering of the elements if it is passed as a `list`, but `self._list` is stored in (operator, coeff) ordering

Return type

`QubitOperatorList`

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` instances - 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

A pair of `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 in the original object.

Parameters

- **sublist_indices** (`list[int]`) – Indices of elements in this instance selected to constitute a new object.

Returns

`TypeVar(OperatorListT, bound= OperatorList)` – A sublist of this instance.

Raises

- **ValueError** – If `sublist_indices` contains indices not contained in this instance, or if this instance
- **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]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

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`) – Maps symbol-representing keys to the value the symbol should be substituted for.

Returns

`LinearListCombiner` – This instance with symbols key symbols replaced by their values.

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. \mathbb{Z}_2 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 `sympify()`.
- **kwargs** (`Any`) – Kwargs to be passed to `sympify()`.

Returns

`LinearListCombiner` – Updated instance of `LinearListCombiner`.

Raises

`RuntimeError` – 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*. :rtype: *SymmetryOperatorPauli*

⚠ Warning

For standard chemical purposes using \mathbb{Z}_2 symmetries this should be OK, but may scale exponentially when in advanced usage.

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 *self* represents the exponential of a linear combination of *Operator* objects, each corresponding to a term in this linear combination. Trotterization is performed at the level of these *Operator* instances. The *Operator* objects 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. The first- and the second-order options are 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 (*precision*=15, *partial*=*False*)

Unsympifies dictionary values.

Replaces symbolic expressions with their corresponding numeric values.

Parameters

- `precision` (`int`, default: 15) – The number of decimal digits of precision used for evaluation.
- `partial` (`default: False`) – Set to True to allow partial unsympification where terms containing free symbols are present. By default, free symbols in any coefficient will cause an exception.

Returns

`LinearListCombiner` – Updated instance of `LinearListCombiner`.

Raises

`TypeError` – Unsympification fails.

untrotterize (*trotter_number*, *trotter_order*=1)

Reverse a Trotter-Suzuki expansion given a product of exponentials as an `OperatorList`.

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 product of exponentials as an `OperatorList`, maintaining separation of exponents.

This method assumes that `self` represents a product of exponentials, with each constituent `Operator` corresponding to the exponentiated term of a single exponential within a product. Scalar factors within this `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`.

❶ 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.
 $1\text{RDM}_{ij}^{\alpha} = \langle a_{i,\alpha}^{\dagger} a_{j,\alpha} \rangle$
- **rdm1_bb** (ndarray) – Reduced one-body density matrix for the beta spin channel.
 $1\text{RDM}_{ij}^{\beta} = \langle a_{i,\beta}^{\dagger} a_{j,\beta} \rangle$

copy()

Performs a deep copy of object.

Return type

RDM

get_block(mask)

Return a new RDM spanning a subset of the original RDM's orbitals.

All orbitals not specified in `mask` are ignored.

Parameters

mask (ndarray) – 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`) – 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.

The arguments are two-body reduced density matrices (2-RDM) for spatial orbitals in the aaaa, bbbb, aabb, bbaa spin channels, where a and b stand for alpha and beta.

Parameters

- **rdm2_aaaa** (ndarray) – 2-RDM for spatial orbitals in the aaaa spin channels as a 4D array.
 $2\text{RDM}_{ijkl}^{\alpha\alpha} = \langle a_{i,\alpha}^\dagger a_{k,\alpha}^\dagger a_{l,\alpha} a_{j,\alpha} \rangle$
- **rdm2_bbbb** (ndarray) – 2-RDM for spatial orbitals in the bbbb spin channels as a 4D array.
 $2\text{RDM}_{ijkl}^{\beta\beta} = \langle a_{i,\beta}^\dagger a_{k,\beta}^\dagger a_{l,\beta} a_{j,\beta} \rangle$
- **rdm2_aabb** (ndarray) – 2-RDM for spatial orbitals in the aabb spin channels as a 4D array.
 $2\text{RDM}_{ijkl}^{\alpha\beta} = \langle a_{i,\alpha}^\dagger a_{k,\beta}^\dagger a_{l,\beta} a_{j,\alpha} \rangle$
- **rdm2_bbaa** (ndarray) – 2-RDM for spatial orbitals in the bbaa spin channels as a 4D array.
 $2\text{RDM}_{ijkl}^{\beta\alpha} = \langle a_{i,\beta}^\dagger a_{k,\alpha}^\dagger a_{l,\alpha} a_{j,\beta} \rangle$

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`

27.11 inquanto.protocols

27.11.1 Protocols for Expectation Values

```
class PauliAveraging(backend=None, shots_per_circuit=8000,
                      pauli_partition_strategy=PauliPartitionStrat.NonConflictingSets,
                      pauli_colour_method=GraphColourMethod.Lazy)
```

Bases: `ProtocolListItem`, `PartiallyPickleable`, `QermittRunMixin`, `ComputableCompliantMixin`

Calculates the expectation value of a Hermitian operator by operator averaging the system register.

Implements the ‘Operator Averaging’ procedure (see: [arXiv:1407.7863](#), [arXiv:1510.04279](#)).

Parameters

- **backend** (`Optional[Backend]`, default: `None`) – The backend to use for quantum computations.
- **shots_per_circuit** (`int`, default: 8000) – Number of shots for each circuit. Default is 8000.
- **pauli_partition_strategy** (`Optional[PauliPartitionStrat]`, default: `PauliPartitionStrat.NonConflictingSets`) – Strategy to partition Pauli operators.
- **pauli_colour_method** (`Optional[GraphColourMethod]`, default: `GraphColourMethod.Lazy`) – Method to perform graph colouring.

```
build(parameters, state, *operators, noise_mitigation=None, optimisation_level=1)
```

Builds the necessary circuits and measurement data for the state and Pauli strings in the operators.

Note

The coefficients in the operators are ignored.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – A dictionary or `SymbolDict` containing the parameter values for the circuits.
- **state** (`GeneralAnsatz`) – Parametrized input state.
- **operators** (`QubitOperator`) – Qubit operators, the Pauli strings in these `QubitOperator` objects are used to generate measurement circuits.
- **noise_mitigation** (`Optional[NoiseMitigation]`, default: `None`) – The noise mitigation instance containing pre and post mitigation strategies.
- **optimisation_level** (`int`, default: 1) – Passed to the backend’s `get_compiled_circuits()` method.

Returns

PauliAveraging – Self instance.

build_from(parameters, computable, exclude=None, noise_mitigation=None)

Build the protocol based on given parameters and computable expression.

This method walks over the computable expression tree and collects all qubit operators appearing in nodes of type:

- *ExpectationValue*
- *ExpectationValueNonHermitian*

After it has walked over the tree, it calls the *build* method.

Raises

NotImplementedError – If not all *ExpectationValue* nodes in the computable tree contain the same state.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Parameters for building.
- **computable** (`ComputableNode`) – A root node of a computable expression tree, simply it is a computable expression.
- **exclude** (`Optional[Callable[[ComputableNode], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.
- **noise_mitigation** (`Optional[NoiseMitigation]`, default: `None`) – The noise mitigation instance containing pre and post mitigation strategies.

Returns

PauliAveraging – Modified instance after building.

classmethod build_protocols_from(parameters, computable, exclude=None, noise_mitigation=None, *args, **kwargs)

Build a list of protocols based on the given parameters and computable expression.

This method walks over the computable expression tree and collects all qubit operator kernels appearing in nodes of type:

- *ExpectationValue*
- *ExpectationValueNonHermitian*

After it has walked over the tree, it creates an instance of *PauliAveraging* for each distinct state and calls the *build()* method with the kernel operators associated with the state. It collects all created protocols into a *ProtocolList* object to be returned.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Values for the parameters in the computable expression.
- **computable** (`ComputableNode`) – A root node of a computable expression tree.
- **exclude** (`Optional[Callable[[ComputableNode], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.
- **noise_mitigation** (`Optional[NoiseMitigation]`, default: `None`) – The noise mitigation instance containing pre and post mitigation strategies.
- ***args** – Arguments passed to the constructor of *PauliAveraging*.

- ****kwargs** – Keyword arguments passed to the constructor of *PauliAveraging*.

Returns

ProtocolList – A list, containing all the newly instantiated and built protocols.

`clear()`

Resets the internal state of the object by clearing all stored data.

This method clears all stored data including state hashes, operator hashes, measurement setups, parameters, and associated dataframes and circuits.

Returns

PauliAveraging – self.

`cost()`

Calculate a simple cost metric running the protocol.

The cost is calculated as the sum of the depth of each circuit multiplied by the number of shots.

Returns

`int` – The cost value as an integer.

`credits(syntax_checker=None, use_websocket=None)`

Evaluate an approximate cost for the measurement circuits built in Quantinuum credits.

Syntax checker will usually be automatically selected, but in some cases needs to be provided. If the measurement circuits are not built yet, the credits will be zero.

Note

This works only with QuantinuumBackend compatible backends.

Parameters

- **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 total cost in Quantinuum credits to run the circuits.

`dataframe_circuit_shot()`

Create a pandas DataFrame with circuit, shot, and depth information.

Returns

`DataFrame` – A pandas DataFrame containing the circuit, shot, and depth information.

`dataframe_measurements()`

Create a DataFrame consisting of computational details.

Returns

`DataFrame` – A pandas DataFrame with columns '`pauli_string`', '`mean`', and '`stddev`'. Each row represents a unique Pauli string and its associated mean and standard error.

`dataframe_partitioning()`

Create a DataFrame consisting of partitioning info.

Returns

`DataFrame` – A pandas DataFrame with columns 'pauli_string', 'circ_index', and 'circ_name'. Each row represents a Pauli string and its circuit index pairs.

dump (file)

Save the object to a file using pickle.

Parameters

`file` (`Union[str, BinaryIO]`) – The file path or file object to write the pickled data to.

Return type

`None`

dumps ()

Returns the object pickled as a bytes object.

Returns

`bytes` – The object's state pickled as a bytes object.

evaluate_expectation_uvalue (state, kernel)

Evaluates the expectation value of a Hermitian kernel with linear error propagation theory.

Similarly to `evaluate_expectation_value ()`, this can be only performed if before calling this protocol has been built for the state and kernel (or other operators composed of the same Pauli strings).

Note

It is assumed the measurements for each Pauli strings are independent, that is not generally the case if measurement reduction is applied.

Parameters

- `state` (`GeneralAnsatz`) – The quantum state for which the expectation value of the kernel is to be calculated.
- `kernel` (`QubitOperator`) – The operator for which the expectation value is being computed.

Returns

`ufloat` – The expectation value of the kernel with respect to the given state with standard error.

evaluate_expectation_value (state, kernel)

Evaluates the provided expectation value.

Computes the expectation value with the kernel and state provided.

This method can only be performed if, prior to calling this method, the protocol has been built for the input state and kernel (or other operators composed of the same Pauli strings), and the protocol has also been run.

Parameters

- `state` (`GeneralAnsatz`) – The quantum state for which the expectation value of the kernel is to be calculated.
- `kernel` (`QubitOperator`) – The operator for which the expectation value is being computed.

Returns

`float` – The expectation value of the kernel with respect to the given state.

get_circuits()

Returns the quantum circuits built for this protocol.

Returns

`List[Circuit]` – List of circuits.

get_circuitshots()

Generate the circuit shot pairs.

Yields

Pair of circuit and the associated number of shots.

Return type

`Iterator[CircuitShots]`

get_evaluator(allow_partial=False)

Returns an evaluator function to evaluate quantum computables.

This method creates and returns a function (evaluator) that takes in a specific quantum computable and evaluates it based on its type.

Quantum computables the returned evaluator can handle:

- `ExpectationValue`
- `ExpectationValueNonHermitian`

Parameters

`allow_partial` (`bool`, default: `False`) – If `False`, evaluation will fail when an unsupported computable node is encountered. If `True`, unsupported nodes will be skipped.

Note

This evaluator works for quantum computables the protocol has been built from.

Raises

`NotImplementedError` – If an unsupported computable is encountered, and `allow_partial==False`.

Returns

`Callable[[Evaluatable], Union[Evaluatable, Any]]` – A function that can evaluate quantum computables. If a computable is supported by this protocol, it is computed; otherwise, it returns the computable itself.

get_runner(qc, compile_symbolic=False, *args, **kwargs)

Returns an end-to-end executor function for a quantum computable.

All nodes in the quantum computable must be evaluable by this protocol. That is, the leaf nodes need to be of one of the supported types:

- `ExpectationValue`
- `ExpectationValueNonHermitian`

Note

At every call of the returned function the internal state of the protocol changes.

Parameters

- `qc` (`ComputableNode`) – The quantum computable node to be measured and evaluated at every call of the returned function.
- `compile_symbolic` (`bool`, default: `False`) – If `True`, circuits are immediately built and compiled symbolically. Calling the returned function substitutes symbols, runs circuits, and evaluates the output. If `False`, circuits are built and compiled only when the returned function is called and parameters are provided. Note that circuits may be deeper when `compile_symbolic=True`, because circuit compilation cannot perform optimizations based on numerical parameter values. The advantage of this case is that compilation only takes place once.

Returns

`Callable[[Union[SymbolDict, Dict]], Any]` – A function that takes the parameters, builds and measures the necessary circuits, and returns the evaluated result.

`get_shots()`

Returns the number shots to be used for each circuit.

Returns

`List[int]` – List of number of shots, the length of this list is the same as the number of circuits.

`property is_built: bool`

Boolean flag indicating if the instance has been built.

Returns

`True` if the `build()` method has been successfully invoked, otherwise `False`.

`property is_numeric: bool`

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`False` if any built measurement circuit contains unsubstituted symbols, otherwise `True`.

`property is_run: bool`

Boolean flag indicating if the instance has been run.

Returns

`True` if the `run()` method has been successfully invoked, otherwise `False`.

`property is_symbolic: bool`

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`True` if any built measurement circuit contains unsubstituted symbols, otherwise `False`.

`launch(*args, **kwargs)`

Launch the circuits to the backend and return the handles for the results.

This method processes all the circuits and returns a list of `ResultHandle` objects representing the handles for the results.

Parameters

- `*args` – Additional arguments to be passed to `self.backend.process_circuits()`.
- `**kwargs` – Additional keyword arguments to be passed to `self.backend.process_circuits()`.

Returns

`List[ResultHandle]` – A list of pytket `ResultHandle` objects representing the handles for the launched circuits.

classmethod `load(file, *args, **kwargs)`

Load a pickled object from a file.

Parameters

- `file (Union[str, BinaryIO])` – The file path or file object to load the pickled data from.
- `*args (Any)` – Additional arguments passed to the class constructor.
- `**kwargs (Any)` – Additional keyword arguments passed to the class constructor.

Returns

`TypeVar(T, bound= PartiallyPickleable)` – An instance of the class with its state loaded from the pickled data.

classmethod `loads(pickled_data, *args, **kwargs)`

Load a pickled object from a bytes object.

Parameters

- `pickled_data (bytes)` – The pickled data to load the object from.
- `*args (Any)` – Additional arguments passed to the class constructor.
- `**kwargs (Any)` – Additional keyword arguments passed to the class constructor.

Returns

`TypeVar(T, bound= PartiallyPickleable)` – An instance of the class with its state loaded from the pickled data.

property `n_circuit: int`

Returns the total number of circuits.

rebuild(*args, **kwargs)

Rebuild the internal data structure.

It is equivalent to `clear().build(*args, **kwargs)`.

Parameters

- `*args (Any)` – Arguments to be passed to `build()`.
- `**kwargs (Any)` – Keyword arguments to be passed to `build()`.

Returns

`TypeVar(TBuildClearMixin, bound= BuildClearMixin)` – self.

retrieve(source, *args, **kwargs)

Retrieve distributions from the backend for the given source.

If the `source` is a list of pytket `ResultHandle`, the distributions are retrieved using `self.backend.get_results()` method. If the `source` is a list of pytket `BackendResult`, it is assumed that the results are already provided.

Parameters

- `source (Union[List[ResultHandle], List[BackendResult]])` – A list of pytket `ResultHandle` or `BackendResult` objects representing the source of the distributions.
- `**kwargs` – Additional keyword arguments to be passed to `self.backend.get_results()` when retrieving results.

Returns

PauliAveraging – Returns self instance.

run (*args, **kwargs)

Run the protocol and waiting for the results.

This method executes the following steps: 1. Launches the measurement circuits to obtain the circuit handles.
2. Retrieves the measurement distributions using the circuit handles.

Parameters

- ***args** (Any) – Arguments to be passed to `launch()`.
- ****kwargs** (Any) – Keyword arguments to be passed to `launch()`.

Returns

TypeVar(TLaunchRetrieveMixin, bound=LaunchRetrieveMixin) – self.

run_mitex (mitex, characterisation)

Run via Qermit `MitEx` instance with provided characterisation.

The following steps will be executed: Step 1: State and Pauli strings are converted to `ObservableExperiment` experiments. Step 2: The experiments are run via the `MitEx` instance. Step 3: Results are processed and internal data structures are updated.

Parameters

- **mitex** (`MitEx`) – The instance of `MitEx` to run.
- **characterisation** (Dict) – The characterisation to use for the run.

Returns

PauliAveraging – An instance of the class with updated data.

run_mitres (mitres, characterisation)

Run via Qermit `MitRes` instance with provided characterisation.

The following steps will be executed: Step 1: circuits and shots will be generated and passed to `MitRes` instance to run. Step 2: the backend results from the `MitRes` run is retrieved and internal data is updated.

Parameters

- **mitres** (`MitRes`) – The instance of `MitRes` to run.
- **characterisation** (Dict) – The characterisation to use for the run.

Returns

PauliAveraging – An instance of the class with updated data.

class HadamardTest (backend, shots_per_circuit=8000)

Bases: `ProtocolListItem`, `PartiallyPickleable`, `ComputableCompliantMixin`

Calculate the expectation value of a qubit operator using Hadamard tests.

Given an operator as a linear combination of Pauli strings, $H = \sum_i c_i P_i$, computes the expectation value $\langle H \rangle$ by performing a set of Hadamard tests to measure the real expectation value of each pauli string (see also the [wikipedia](#) article).

Parameters

- **backend** (Backend) – The backend to use for quantum computations.
- **shots_per_circuit** (int, default: 8000) – Number of shots for each circuit.

`build(parameters, state, *operators, optimisation_level=1)`

Builds Hadamard test measurement circuits for each Pauli string.

Note

The coefficients in the operators are ignored.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – A dictionary or `SymbolDict` containing the parameter values for the circuits.
- **state** (`GeneralAnsatz`) – Parametrized input state.
- **operators** (`QubitOperator`) – Qubit operators, the Pauli strings in these `QubitOperator` objects are used to generate measurement circuits.
- **optimisation_level** (`int`, default: 1) – Passed to the backend's `get_compiled_circuits()` method.

Returns

`TypeVar(T, bound= HadamardTest)` – Self instance.

`build_from(parameters, computable, exclude=None)`

Build the protocol based on given parameters and computable expression.

This method walks over the computable expression tree and collects all qubit operators appearing in nodes of type:

- `ExpectationValue`
- `ExpectationValueNonHermitian`

After it has walked over the tree, it calls the `build` method.

Raises

`NotImplementedError` – If not all `ExpectationValue` nodes in the computable tree contain the same state.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Parameters for building.
- **computable** (`ComputableNode`) – A root node of a computable expression tree, simply it is a computable expression.
- **exclude** (`Optional[Callable[[ComputableNode], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.

Returns

`HadamardTest` – The modified instance after building.

`classmethod build_protocols_from(parameters, computable, exclude=None, *args, **kwargs)`

Build a `ProtocolList` based on the given parameters and a computable expression.

This method walks over the computable expression tree and collects all qubit operator kernels appearing in nodes of type:

- `ExpectationValue`
- `ExpectationValueNonHermitian`

After walking over the tree, it creates an instance of `HadamardTest` for each distinct state and calls the `build()` method with the kernel operators associated with the state. It collects all created protocols into a `ProtocolList` object to be returned.

Parameters

- `parameters` (`Union[SymbolDict, Dict]`) – Values for the parameters in the computable expression.
- `computable` (`ComputableNode`) – A root node of a computable expression tree.
- `exclude` (`Optional[Callable[[ComputableNode], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing
- `*args` – Arguments passed to the constructor of `HadamardTest`.
- `**kwargs` – Keyword arguments passed to the constructor of `HadamardTest`.

Returns

`ProtocolList` – `ProtocolList` of the newly instantiated and built protocols.

`clear()`

Resets the internal state of the object by clearing all stored data.

This method clears all stored data including state hashes, operator hashes, measurement setups, parameters, and associated dataframes and circuits.

Returns

`TypeVar(T, bound= HadamardTest)` – self.

`cost()`

Calculate a simple cost metric running the protocol.

The cost is calculated as the sum of the depth of each circuit multiplied by the number of shots.

Returns

`int` – The cost value as an integer.

`credits(syntax_checker=None, use_websocket=None)`

Evaluate an approximate cost for the measurement circuits built in Quantinuum credits.

Syntax checker will usually be automatically selected, but in some cases needs to be provided. If the measurement circuits are not built yet, the credits will be zero.

Note

This works only with `QuantinuumBackend` compatible backends.

Parameters

- `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 total cost in Quantinuum credits to run the circuits.

`dataframe_circuit_shot()`

Create a pandas `DataFrame` with circuit, shot, and depth information.

Returns

`DataFrame` – A pandas `DataFrame` containing the circuit, shot, and depth information.

dataframe_measurements()

Create a `DataFrame` consisting of computational details.

Returns

`DataFrame` – A pandas `DataFrame` with columns '`pauli_string`', '`mean`', and '`stderr`'. Each row represents a unique Pauli string and its associated mean and standard error.

dump (file)

Save the object to a file using pickle.

Parameters

`file` (`Union[str, BinaryIO]`) – The file path or file object to write the pickled data to.

Return type

`None`

dumps ()

Returns the object pickled as a bytes object.

Returns

`bytes` – The object's state pickled as a bytes object.

evaluate_expectation_value (state, kernel)

Evaluates the provided expectation value.

Computes the expectation value with the kernel and state provided.

This method can only be performed if, prior to calling this method, the protocol has been built for the input state and kernel (or other operators composed of the same Pauli strings), and the protocol has also been run.

Parameters

- `state` (`GeneralAnsatz`) – The quantum state for which the expectation value of the kernel is to be calculated.
- `kernel` (`Union[QubitOperator, QubitOperatorString]`) – The operator for which the expectation value is being computed.

Returns

`float` – The expectation value of the kernel with respect to the given state.

get_circuits()

Returns the quantum circuits built for this protocol.

Returns

`List[Circuit]` – List of circuits.

get_circuitshots()

Generate the circuit shot pairs.

Yields

Pair of circuit and the associated number of shots.

Return type

`Iterator[CircuitShots]`

get_evaluator (allow_partial=False)

Returns an evaluator function to evaluate quantum computables.

This method creates and returns a function (evaluator) that takes in a specific quantum computable and evaluates it based on its type.

Quantum computables the returned evaluator can handle:

- *ExpectationValue*
- *ExpectationValueNonHermitian*

Parameters

`allow_partial (bool, default: False)` – If `False`, evaluation will fail when an unsupported computable node is encountered. If `True`, unsupported nodes will be skipped.

Raises

`NotImplementedError` – If an unsupported computable is encountered, and `allow_partial==False`.

Note

This evaluator works for quantum computables the protocol has been built from.

Returns

`Callable[[Evaluable], Union[Evaluatable, Any]]` – A function that can evaluate quantum computables. If a computable is supported by this protocol, it is computed; otherwise, it returns the computable itself.

get_runner (qc, compile_symbolic=False, *args, **kwargs)

Returns an end-to-end executor function for a quantum computable.

All nodes in the quantum computable must be evaluable by this protocol. That is, the leaf nodes need to be of one of the supported types:

- *ExpectationValue*
- *ExpectationValueNonHermitian*

Note

At every call of the returned function the internal state of the protocol changes.

Parameters

- `qc (ComputableNode)` – The quantum computable node to be measured and evaluated at every call of the returned function.
- `compile_symbolic (bool, default: False)` – If `True`, circuits are immediately built and compiled symbolically. Calling the returned function substitutes symbols, runs circuits, and evaluates the output. If `False`, circuits are built and compiled only when the returned function is called and parameters are provided. Note that circuits may be deeper when `compile_symbolic=True`, because circuit compilation cannot perform optimizations based on numerical parameter values. The advantage of this case is that compilation only takes place once.

Returns

`Callable[[Union[SymbolDict, Dict]], Any]` – A function that takes the parameters, builds and measures the necessary circuits, and returns the evaluated result. If the result is not a `float`, it returns `math.nan`.

get_shots()

Returns the number shots to be used for each circuit.

Returns

`Iterator[int]` – List of number of shots, the length of this list is the same as the number of circuits.

property is_built: bool

Boolean flag indicating if the instance has been built.

Returns

`True` if the `build()` method has been successfully invoked, otherwise `False`.

property is_numeric: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`False` if any built measurement circuit contains unsubstituted symbols, otherwise `True`.

property is_run: bool

Boolean flag indicating if the instance has been run.

Returns

`True` if the `run()` method has been successfully invoked, otherwise `False`.

property is_symbolic: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`True` if any built measurement circuit contains unsubstituted symbols, otherwise `False`.

launch(*args, **kwargs)

Launch the circuits to the backend and return the handles for the results.

This method processes all the circuits and returns a list of `ResultHandle` objects representing the handles for the results.

Parameters

- `*args` – Additional arguments to be passed to `self.backend.process_circuits()`.
- `**kwargs` – Additional keyword arguments to be passed to `self.backend.process_circuits()`.

Returns

`List[ResultHandle]` – A list of pytket `ResultHandle` objects representing the handles for the launched circuits.

classmethod load(file, *args, **kwargs)

Load a pickled object from a file.

Parameters

- `file (Union[str, BinaryIO])` – The file path or file object to load the pickled data from.
- `*args (Any)` – Additional arguments passed to the class constructor.

- ****kwargs** ([Any](#)) – Additional keyword arguments passed to the class constructor.

Returns

[TypeVar](#)(*T*, bound= PartiallyPickleable) – An instance of the class with its state loaded from the pickled data.

classmethod loads (*pickled_data*, **args*, ***kwargs*)

Load a pickled object from a bytes object.

Parameters

- **pickled_data** ([bytes](#)) – The pickled data to load the object from.
- ***args** ([Any](#)) – Additional arguments passed to the class constructor.
- ****kwargs** ([Any](#)) – Additional keyword arguments passed to the class constructor.

Returns

[TypeVar](#)(*T*, bound= PartiallyPickleable) – An instance of the class with its state loaded from the pickled data.

property n_circuit: int

Returns the total number of circuits.

rebuild (**args*, ***kwargs*)

Rebuild the internal data structure.

It is equivalent to `clear().build(*args, **kwargs)`.

Parameters

- ***args** ([Any](#)) – Arguments to be passed to `build()`.
- ****kwargs** ([Any](#)) – Keyword arguments to be passed to `build()`.

Returns

[TypeVar](#)(TBuildClearMixin, bound= BuildClearMixin) – self.

retrieve (*source*, **args*, ***kwargs*)

Retrieve distributions from the backend for the given source.

If the *source* is a list of pytket ResultHandle, the distributions are retrieved using `self.backend.get_results()` method. If the *source* is a list of pytket BackendResult, it is assumed that the results are already provided.

Parameters

- **source** ([Union\[List\[ResultHandle\], List\[BackendResult\]\]](#)) – A list of pytket ResultHandle or BackendResult objects representing the source of the distributions.
- ****kwargs** – Additional keyword arguments to be passed to `self.backend.get_results()` when retrieving results.

Returns

[TypeVar](#)(*T*, bound= HadamardTest) – Returns self instance.

run (**args*, ***kwargs*)

Run the protocol and waiting for the results.

This method executes the following steps: 1. Launches the measurement circuits to obtain the circuit handles. 2. Retrieves the measurement distributions using the circuit handles.

Parameters

- ***args** ([Any](#)) – Arguments to be passed to `launch()`.

- ****kwargs** ([Any](#)) – Keyword arguments to be passed to `launch()`.

Returns

`TypeVar(TLaunchRetrieveMixin, bound=LaunchRetrieveMixin)` – self.

27.11.2 Protocols for Overlap Squared

```
class ComputeUncompute(backend, n_shots=8000)
```

Bases: `BaseOverlapSquaredProtocol`

Calculates the overlap squared with the compute-uncompute method.

For input states $|\psi\rangle = U|0\rangle$ and $|\phi\rangle = V|0\rangle$, prepares the state $U^\dagger V|0\rangle$. The overlap squared $|\langle\psi|\phi\rangle|^2$ is then given by the probability of measuring a vacuum state $|0\rangle$. See <https://arxiv.org/abs/1810.02327>.

Supports calculation of overlap squared of the form: $|\langle\psi|cP|\phi\rangle|^2$, where P is a Pauli word and c is a numeric constant.

Parameters

- **backend** (`Backend`) – The backend to use for quantum computations.
- **n_shots** (`int`, default: 8000) – Number of shots to perform.

```
build(parameters, bra_state, ket_state, kernel=None, optimisation_level=1)
```

Builds the necessary circuits and measurement data.

 **Note**

Any coefficient in the operator kernel is ignored.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Parameter values for the circuits.
- **bra_state** (`GeneralAnsatz`) – Left-hand state.
- **ket_state** (`GeneralAnsatz`) – Right-hand state.
- **kernel** (`Union[QubitOperator, QubitOperatorString, None]`, default: `None`) – Optional operator kernel. Must be a single Pauli string.
- **optimization_level** – Passed as the `optimisation_level` arg to the backend's `get_compiled_circuits()` method.
- **optimisation_level** (`int`, default: 1)

Returns

`TypeVar(T, bound=BaseOverlapSquaredProtocol)` – Self instance after building the necessary circuits and measurement data.

```
build_from(parameters, computable, exclude=None)
```

Builds the protocol based on given parameters and computable expression.

This method walks over the computable expression tree and calls the `build()` method when it encounters a node of type `OverlapSquared`.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Parameters for building.

- **computable** (*ComputableNode*) – A root node of a computable expression tree, simply it is a computable expression.
- **exclude** (*Optional[Callable[[ComputableNode], bool]]*, default: *None*) – Optional callable function to exclude certain nodes from processing.

Returns

TypeVar(*T*, bound= *BaseOverlapSquaredProtocol*) – Returns the modified instance after building.

classmethod build_protocols_from(*parameters*, *computable*, *exclude=None*, **args*, ***kwargs*)

Build a list of protocols based on the given parameters and computable expression.

This method walks over the computable expression tree and collects leaf nodes of type:

- *OverlapSquared*

After it has walked over the tree, it creates an instance of this protocol for each distinct bra, ket, kernel trio and calls the *build()* method. It collects all created protocols into a *ProtocolList* object to be returned.

Parameters

- **parameters** (*Union[SymbolDict, Dict]*) – Values for the parameters in the computable expression.
- **computable** (*ComputableNode*) – A root node of a computable expression tree.
- **exclude** (*Optional[Callable[[ComputableNode], bool]]*, default: *None*) – Optional callable function to exclude certain nodes from processing.
- ***args** – Arguments passed to the constructor of the overlap squared protocol.
- ****kwargs** – Keyword arguments passed to the constructor of the overlap squared protocol.

Returns

ProtocolList – A list, containing all the newly instantiated and built protocols.

circuit: Optional[Circuit]

clear()

Resets the internal state of the object by clearing all stored data.

This method clears all stored data including state hashes, operator hashes, measurement setups, parameters, and associated dataframes and circuits.

Returns

TypeVar(*T*, bound= *BaseOverlapSquaredProtocol*) – self.

cost()

Calculate a simple cost metric running the protocol.

The cost is calculated as the sum of the depth of each circuit multiplied by the number of shots.

Returns

int – The cost value as an integer.

credits (*syntax_checker=None*, *use_websocket=None*)

Evaluate an approximate cost for the measurement circuits built in Quantinuum credits.

Syntax checker will usually be automatically selected, but in some cases needs to be provided. If the measurement circuits are not built yet, the credits will be zero.

Note

This works only with QuantinuumBackend compatible backends.

Parameters

- **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 total cost in Quantinuum credits to run the circuits.

dataframe_circuit_shot()

Create a pandas DataFrame with circuit, shot, and depth information.

Returns

`DataFrame` – A pandas DataFrame containing the circuit, shot, and depth information.

dump(file)

Save the object to a file using pickle.

Parameters

- **file** (`Union[str, BinaryIO]`) – The file path or file object to write the pickled data to.

Return type

`None`

dumps()

Returns the object pickled as a bytes object.

Returns

`bytes` – The object's state pickled as a bytes object.

evaluate_overlap_squared(bra_state, ket_state, kernel)

Evaluates the overlap squared for given states and kernel.

Parameters

- **bra_state** (`GeneralAnsatz`) – Left-hand state.
- **ket_state** (`GeneralAnsatz`) – Right-hand state.
- **kernel** (`Union[QubitOperator, QubitOperatorString]`) – Overlap kernel. Must be a single Pauli string.

Note

Input states and pauli string in the kernel need to match with the states and pauli string the circuit was built for to correctly interpret the circuit output distribution.

Returns

Overlap squared.

`get_circuits()`
Returns the quantum circuits built for this protocol.

Returns

`List[Circuit]` – List of circuits.

`get_circuitshots()`
Generate the circuit shot pairs.

Yields

Pair of circuit and the associated number of shots.

Return type

`Iterator[CircuitShots]`

`get_evaluator(allow_partial=False)`
Returns an evaluator function to evaluate quantum computables.

This method creates and returns a function (evaluator) that takes in a specific quantum computable and evaluates it based on its type.

Quantum computables the returned evaluator can handle:

- `OverlapSquared`

Parameters

`allow_partial(bool, default: False)` – If `False`, evaluation will fail when an unsupported computable node is encountered. If `True`, unsupported nodes will be skipped.

Raises

`NotImplementedError` – If an unsupported computable is encountered, and `allow_partial==False`.

Note

This evaluator works for quantum computables the protocol has been built from.

Returns

`Callable[[Evaluable], Union[Evaluable, Any]]` – A function that can evaluate quantum computables. If a computable is supported by this protocol, it is computed; otherwise, it returns the computable itself.

`get_runner(c, compile_symbolic=False, *args, **kwargs)`

Returns an end-to-end executor function for a quantum computable.

Note

All nodes in the quantum computable must be evaluable by this protocol. That is, the leaf nodes need to be `OverlapSquared`.

Note

At every call of the returned function the internal state of the protocol changes.

Parameters

- **c** (*ComputableNode*) – The quantum computable node to be measured and evaluated at every call of the returned function.
- **compile_symbolic** (*bool*, default: `False`) – If `True`, circuits are immediately built and compiled symbolically. Calling the returned function substitutes symbols, runs circuits, and evaluates the output. If `False`, circuits are built and compiled only when the returned function is called and parameters are provided. Note that circuits may be deeper when `compile_symbolic=True`, because circuit compilation cannot perform optimizations based on numerical parameter values. The advantage of this case is that compilation only takes place once.
- **args** (*Any*)
- **kwargs** (*Any*)

Returns

`Callable[[Union[SymbolDict, Dict]], Any]` – A function that takes the parameters, builds and measures the necessary circuits, and returns the evaluated result. If the result is not a `float`, it returns `math.nan`.

get_shots()

Returns the number shots to be used for each circuit.

Returns

`List[int]` – List of number of shots, the length of this list is the same as the number of circuits.

property is_built: bool

Boolean flag indicating if the instance has been built.

Returns

`True` if the `build()` method has been successfully invoked, otherwise `False`.

property is_numeric: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`False` if any built measurement circuit contains unsubstituted symbols, otherwise `True`.

property is_run: bool

Boolean flag indicating if the instance has been run.

Returns

`True` if the `run()` method has been successfully invoked, otherwise `False`.

property is_symbolic: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`True` if any built measurement circuit contains unsubstituted symbols, otherwise `False`.

launch(*args, **kwargs)

Launch the circuits to the backend and return the handles for the results.

This method processes all the circuits and returns a list of `ResultHandle` objects representing the handles for the results.

Parameters

- ***args** (*Any*) – Additional arguments to be passed to `self.backend.process_circuits()`.

- ****kwargs** (`Any`) – Additional keyword arguments to be passed to `self.backend.process_circuits()`.

Returns

`List[ResultHandle]` – A list of pytket `ResultHandle` objects representing the handles for the launched circuits.

classmethod `load(file, *args, **kwargs)`

Load a pickled object from a file.

Parameters

- **file** (`Union[str, BinaryIO]`) – The file path or file object to load the pickled data from.
- ***args** (`Any`) – Additional arguments passed to the class constructor.
- ****kwargs** (`Any`) – Additional keyword arguments passed to the class constructor.

Returns

`TypeVar(T, bound= PartiallyPickleable)` – An instance of the class with its state loaded from the pickled data.

classmethod `loads(pickled_data, *args, **kwargs)`

Load a pickled object from a bytes object.

Parameters

- **pickled_data** (`bytes`) – The pickled data to load the object from.
- ***args** (`Any`) – Additional arguments passed to the class constructor.
- ****kwargs** (`Any`) – Additional keyword arguments passed to the class constructor.

Returns

`TypeVar(T, bound= PartiallyPickleable)` – An instance of the class with its state loaded from the pickled data.

property `n_circuit: int`

Returns the total number of circuits.

`rebuild(*args, **kwargs)`

Rebuild the internal data structure.

It is equivalent to `clear().build(*args, **kwargs)`.

Parameters

- ***args** (`Any`) – Arguments to be passed to `build()`.
- ****kwargs** (`Any`) – Keyword arguments to be passed to `build()`.

Returns

`TypeVar(TBuildClearMixin, bound= BuildClearMixin)` – `self`.

`retrieve(source, **kwargs)`

Retrieve distributions from the backend for the given source.

If the `source` is a list of pytket `ResultHandle`, the distributions are retrieved using `self.backend.get_results()` method. If the `source` is a list of pytket `BackendResult`, it is assumed that the results are already provided.

Parameters

- **source** (`Union[List[ResultHandle], List[BackendResult]]`) – A list of pytket `ResultHandle` or `BackendResult` objects representing the source of the distributions.

- ****kwargs** (`Any`) – Additional keyword arguments to be passed to `self.backend.get_results()` when retrieving results.

Returns

`TypeVar(T, bound= BaseOverlapSquaredProtocol)` – Returns self instance.

run (*args, **kwargs)

Run the protocol and waiting for the results.

This method executes the following steps: 1. Launches the measurement circuits to obtain the circuit handles. 2. Retrieves the measurement distributions using the circuit handles.

Parameters

- ***args** (`Any`) – Arguments to be passed to `launch()`.
- ****kwargs** (`Any`) – Keyword arguments to be passed to `launch()`.

Returns

`TypeVar(TLaunchRetrieveMixin, bound= LaunchRetrieveMixin)` – self.

class DestructiveSwapTest (backend, n_shots=8000)

Bases: `BaseOverlapSquaredProtocol`

Calculate the overlap squared with the destructive swap test.

Based on <https://arxiv.org/abs/1303.6814>. Prepares both bra and ket states in parallel and performs a destructive swap test i.e. no ancilla qubit is required. The test succeeds when a bitwise AND between the output bra and ket state registers has even parity. The probability of success is given by $\frac{1}{2}(1 + |\langle\psi|\phi\rangle|^2)$.

The bra state register is used as the control, while the ket register is the target.

Supports calculation of overlap squared of the form: $|\langle\psi|cP|\phi\rangle|^2$, where P is a Pauli word and c is a numeric constant.

Parameters

- **backend** (`Backend`) – The backend to use for quantum computations.
- **n_shots** (`int`, default: 8000) – Number of shots to perform.

build (parameters, bra_state, ket_state, kernel=None, optimisation_level=1)

Builds the necessary circuits and measurement data.

Note

Any coefficient in the operator kernel is ignored.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Parameter values for the circuits.
- **bra_state** (`GeneralAnsatz`) – Left-hand state.
- **ket_state** (`GeneralAnsatz`) – Right-hand state.
- **kernel** (`Union[QubitOperator, QubitOperatorString, None]`, default: `None`) – Optional operator kernel. Must be a single Pauli string.
- **optimization_level** – Passed as the `optimisation_level` arg to the backend's `get_compiled_circuits()` method.
- **optimisation_level** (`int`, default: 1)

Returns

`TypeVar(T, bound= BaseOverlapSquaredProtocol)` – Self instance after building the necessary circuits and measurement data.

`build_from` (*parameters, computable, exclude=None*)

Builds the protocol based on given parameters and computable expression.

This method walks over the computable expression tree and calls the `build()` method when it encounters a node of type `OverlapSquared`.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Parameters for building.
- **computable** (`ComputableNode`) – A root node of a computable expression tree, simply it is a computable expression.
- **exclude** (`Optional[Callable[[ComputableNode], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.

Returns

`TypeVar(T, bound= BaseOverlapSquaredProtocol)` – Returns the modified instance after building.

`classmethod build_protocols_from` (*parameters, computable, exclude=None, *args, **kwargs*)

Build a list of protocols based on the given parameters and computable expression.

This method walks over the computable expression tree and collects leaf nodes of type:

- `OverlapSquared`

After it has walked over the tree, it creates an instance of this protocol for each distinct bra, ket, kernel trio and calls the `build()` method. It collects all created protocols into a `ProtocolList` object to be returned.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Values for the parameters in the computable expression.
- **computable** (`ComputableNode`) – A root node of a computable expression tree.
- **exclude** (`Optional[Callable[[ComputableNode], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.
- ***args** – Arguments passed to the constructor of the overlap squared protocol.
- ****kwargs** – Keyword arguments passed to the constructor of the overlap squared protocol.

Returns

`ProtocolList` – A list, containing all the newly instantiated and built protocols.

`circuit: Optional[Circuit]`**`clear()`**

Resets the internal state of the object by clearing all stored data.

This method clears all stored data including state hashes, operator hashes, measurement setups, parameters, and associated dataframes and circuits.

Returns

`TypeVar(T, bound= BaseOverlapSquaredProtocol)` – self.

cost ()

Calculate a simple cost metric running the protocol.

The cost is calculated as the sum of the depth of each circuit multiplied by the number of shots.

Returns

`int` – The cost value as an integer.

credits (syntax_checker=None, use_websocket=None)

Evaluate an approximate cost for the measurement circuits built in Quantinuum credits.

Syntax checker will usually be automatically selected, but in some cases needs to be provided. If the measurement circuits are not built yet, the credits will be zero.

Note

This works only with QuantinuumBackend compatible backends.

Parameters

- `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 total cost in Quantinuum credits to run the circuits.

dataframe_circuit_shot ()

Create a pandas DataFrame with circuit, shot, and depth information.

Returns

`DataFrame` – A pandas DataFrame containing the circuit, shot, and depth information.

dump (file)

Save the object to a file using pickle.

Parameters

`file` (`Union[str, BinaryIO]`) – The file path or file object to write the pickled data to.

Return type

`None`

dumps ()

Returns the object pickled as a bytes object.

Returns

`bytes` – The object's state pickled as a bytes object.

evaluate_overlap_squared (bra_state, ket_state, kernel)

Evaluates the overlap squared for given states and kernel.

Parameters

- `bra_state` (`GeneralAnsatz`) – Left-hand state.
- `ket_state` (`GeneralAnsatz`) – Right-hand state.
- `kernel` (`Union[QubitOperator, QubitOperatorString]`) – Overlap kernel. Must be a single Pauli string.

Note

Input states and pauli string in the kernel need to match with the states and pauli string the circuit was built for to correctly interpret the circuit output distribution.

Returns

Overlap squared.

`get_circuits()`

Returns the quantum circuits built for this protocol.

Returns

`List[Circuit]` – List of circuits.

`get_circuitshots()`

Generate the circuit shot pairs.

Yields

Pair of circuit and the associated number of shots.

Return type

`Iterator[CircuitShots]`

`get_evaluator(allow_partial=False)`

Returns an evaluator function to evaluate quantum computables.

This method creates and returns a function (evaluator) that takes in a specific quantum computable and evaluates it based on its type.

Quantum computables the returned evaluator can handle:

- `OverlapSquared`

Parameters

`allow_partial(bool, default: False)` – If `False`, evaluation will fail when an unsupported computable node is encountered. If `True`, unsupported nodes will be skipped.

Raises

`NotImplementedError` – If an unsupported computable is encountered, and `allow_partial==False`.

Note

This evaluator works for quantum computables the protocol has been built from.

Returns

`Callable[[Evaluable], Union[Evaluable, Any]]` – A function that can evaluate quantum computables. If a computable is supported by this protocol, it is computed; otherwise, it returns the computable itself.

`get_runner(c, compile_symbolic=False, *args, **kwargs)`

Returns an end-to-end executor function for a quantum computable.

Note

All nodes in the quantum computable must be evaluable by this protocol. That is, the leaf nodes need to be *OverlapSquared*.

Note

At every call of the returned function the internal state of the protocol changes.

Parameters

- **c** (*ComputableNode*) – The quantum computable node to be measured and evaluated at every call of the returned function.
- **compile_symbolic** (*bool*, default: `False`) – If `True`, circuits are immediately built and compiled symbolically. Calling the returned function substitutes symbols, runs circuits, and evaluates the output. If `False`, circuits are built and compiled only when the returned function is called and parameters are provided. Note that circuits may be deeper when `compile_symbolic=True`, because circuit compilation cannot perform optimizations based on numerical parameter values. The advantage of this case is that compilation only takes place once.
- **args** (*Any*)
- **kwargs** (*Any*)

Returns

Callable[[Union[SymbolDict, Dict]], Any] – A function that takes the parameters, builds and measures the necessary circuits, and returns the evaluated result. If the result is not a *float*, it returns `math.nan`.

get_shots()

Returns the number shots to be used for each circuit.

Returns

List[int] – List of number of shots, the length of this list is the same as the number of circuits.

property is_built: bool

Boolean flag indicating if the instance has been built.

Returns

`True` if the `build()` method has been successfully invoked, otherwise `False`.

property is_numeric: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`False` if any built measurement circuit contains unsubstituted symbols, otherwise `True`.

property is_run: bool

Boolean flag indicating if the instance has been run.

Returns

`True` if the `run()` method has been successfully invoked, otherwise `False`.

```
property is_symbolic: bool
```

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`True` if any built measurement circuit contains unsubstituted symbols, otherwise `False`.

```
launch(*args, **kwargs)
```

Launch the circuits to the backend and return the handles for the results.

This method processes all the circuits and returns a list of `ResultHandle` objects representing the handles for the results.

Parameters

- `*args (Any)` – Additional arguments to be passed to `self.backend.process_circuits()`.
- `**kwargs (Any)` – Additional keyword arguments to be passed to `self.backend.process_circuits()`.

Returns

`List[ResultHandle]` – A list of pytket `ResultHandle` objects representing the handles for the launched circuits.

```
classmethod load(file, *args, **kwargs)
```

Load a pickled object from a file.

Parameters

- `file (Union[str, BinaryIO])` – The file path or file object to load the pickled data from.
- `*args (Any)` – Additional arguments passed to the class constructor.
- `**kwargs (Any)` – Additional keyword arguments passed to the class constructor.

Returns

`TypeVar(T, bound= PartiallyPickleable)` – An instance of the class with its state loaded from the pickled data.

```
classmethod loads(pickled_data, *args, **kwargs)
```

Load a pickled object from a bytes object.

Parameters

- `pickled_data (bytes)` – The pickled data to load the object from.
- `*args (Any)` – Additional arguments passed to the class constructor.
- `**kwargs (Any)` – Additional keyword arguments passed to the class constructor.

Returns

`TypeVar(T, bound= PartiallyPickleable)` – An instance of the class with its state loaded from the pickled data.

```
property n_circuit: int
```

Returns the total number of circuits.

```
rebuild(*args, **kwargs)
```

Rebuild the internal data structure.

It is equivalent to `clear().build(*args, **kwargs)`.

Parameters

- `*args (Any)` – Arguments to be passed to `build()`.

- ****kwargs** (Any) – Keyword arguments to be passed to `build()`.

Returns

`TypeVar(TBuildClearMixin, bound= BuildClearMixin)` – self.

`retrieve(source, **kwargs)`

Retrieve distributions from the backend for the given source.

If the `source` is a list of pytket ResultHandle, the distributions are retrieved using `self.backend.get_results()` method. If the `source` is a list of pytket BackendResult, it is assumed that the results are already provided.

Parameters

- **source** (`Union[List[ResultHandle], List[BackendResult]]`) – A list of pytket ResultHandle or BackendResult objects representing the source of the distributions.
- ****kwargs** (Any) – Additional keyword arguments to be passed to `self.backend.get_results()` when retrieving results.

Returns

`TypeVar(T, bound= BaseOverlapSquaredProtocol)` – Returns self instance.

`run(*args, **kwargs)`

Run the protocol and waiting for the results.

This method executes the following steps: 1. Launches the measurement circuits to obtain the circuit handles.
2. Retrieves the measurement distributions using the circuit handles.

Parameters

- ***args** (Any) – Arguments to be passed to `launch()`.
- ****kwargs** (Any) – Keyword arguments to be passed to `launch()`.

Returns

`TypeVar(TLaunchRetrieveMixin, bound= LaunchRetrieveMixin)` – self.

`class SwapTest(backend, n_shots=8000)`

Bases: `BaseOverlapSquaredProtocol`

Calculate the overlap squared with the canonical swap test.

Input states $|\psi\rangle$ and $|\phi\rangle$ are prepared in parallel alongside an ancilla qubit to generate the state $|0, \psi, \phi\rangle$. A swap operation between ψ and ϕ is controlled on the ancilla and wrapped by Hadamard gates so that the probability of measuring $|0\rangle$ on the ancilla is given by $\frac{1}{2}(1 + |\langle\psi|\phi\rangle|^2)$. See https://en.wikipedia.org/wiki/Swap_test.

Supports calculation of overlap squared of the form: $|\langle\psi|cP|\phi\rangle|^2$, where P is a Pauli word and c is a numeric constant.

Parameters

- **backend** (Backend) – The backend to use for quantum computations.
- **n_shots** (int, default: 8000) – Number of shots to perform.

`build(parameters, bra_state, ket_state, kernel=None, optimisation_level=1)`

Builds the necessary circuits and measurement data.

Note

Any coefficient in the operator kernel is ignored.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Parameter values for the circuits.
- **bra_state** (`GeneralAnsatz`) – Left-hand state.
- **ket_state** (`GeneralAnsatz`) – Right-hand state.
- **kernel** (`Union[QubitOperator, QubitOperatorString, None]`, default: `None`) – Optional operator kernel. Must be a single Pauli string.
- **optimization_level** – Passed as the `optimisation_level` arg to the backend's `get_compiled_circuits()` method.
- **optimisation_level** (`int`, default: 1)

Returns

`TypeVar(T, bound= BaseOverlapSquaredProtocol)` – Self instance after building the necessary circuits and measurement data.

`build_from(parameters, computable, exclude=None)`

Builds the protocol based on given parameters and computable expression.

This method walks over the computable expression tree and calls the `build()` method when it encounters a node of type `OverlapSquared`.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Parameters for building.
- **computable** (`ComputableNode`) – A root node of a computable expression tree, simply it is a computable expression.
- **exclude** (`Optional[Callable[[ComputableNode], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.

Returns

`TypeVar(T, bound= BaseOverlapSquaredProtocol)` – Returns the modified instance after building.

`classmethod build_protocols_from(parameters, computable, exclude=None, *args, **kwargs)`

Build a list of protocols based on the given parameters and computable expression.

This method walks over the computable expression tree and collects leaf nodes of type:

- `OverlapSquared`

After it has walked over the tree, it creates an instance of this protocol for each distinct bra, ket, kernel trio and calls the `build()` method. It collects all created protocols into a `ProtocolList` object to be returned.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Values for the parameters in the computable expression.
- **computable** (`ComputableNode`) – A root node of a computable expression tree.
- **exclude** (`Optional[Callable[[ComputableNode], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.
- ***args** – Arguments passed to the constructor of the overlap squared protocol.
- ****kwargs** – Keyword arguments passed to the constructor of the overlap squared protocol.

Returns

`ProtocolList` – A list, containing all the newly instantiated and built protocols.

`circuit: Optional[Circuit]`

`clear()`

Resets the internal state of the object by clearing all stored data.

This method clears all stored data including state hashes, operator hashes, measurement setups, parameters, and associated dataframes and circuits.

Returns

`TypeVar(T, bound= BaseOverlapSquaredProtocol)` – self.

`cost()`

Calculate a simple cost metric running the protocol.

The cost is calculated as the sum of the depth of each circuit multiplied by the number of shots.

Returns

`int` – The cost value as an integer.

`credits(syntax_checker=None, use_websocket=None)`

Evaluate an approximate cost for the measurement circuits built in Quantinuum credits.

Syntax checker will usually be automatically selected, but in some cases needs to be provided. If the measurement circuits are not built yet, the credits will be zero.

 **Note**

This works only with QuantinuumBackend compatible backends.

Parameters

- `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 total cost in Quantinuum credits to run the circuits.

`dataframe_circuit_shot()`

Create a pandas DataFrame with circuit, shot, and depth information.

Returns

`DataFrame` – A pandas DataFrame containing the circuit, shot, and depth information.

`dump(file)`

Save the object to a file using pickle.

Parameters

`file` (`Union[str, BinaryIO]`) – The file path or file object to write the pickled data to.

Return type

`None`

`dumps()`

Returns the object pickled as a bytes object.

Returns

`bytes` – The object's state pickled as a bytes object.

`evaluate_overlap_squared(bra_state, ket_state, kernel)`

Evaluates the overlap squared for given states and kernel.

Parameters

- `bra_state` (*GeneralAnsatz*) – Left-hand state.
- `ket_state` (*GeneralAnsatz*) – Right-hand state.
- `kernel` (`Union[QubitOperator, QubitOperatorString]`) – Overlap kernel. Must be a single Pauli string.

Note

Input states and pauli string in the kernel need to match with the states and pauli string the circuit was built for to correctly interpret the circuit output distribution.

Returns

Overlap squared.

`get_circuits()`

Returns the quantum circuits built for this protocol.

Returns

`List[Circuit]` – List of circuits.

`get_circuitshots()`

Generate the circuit shot pairs.

Yields

Pair of circuit and the associated number of shots.

Return type

`Iterator[CircuitShots]`

`get_evaluator(allow_partial=False)`

Returns an evaluator function to evaluate quantum computables.

This method creates and returns a function (evaluator) that takes in a specific quantum computable and evaluates it based on its type.

Quantum computables the returned evaluator can handle:

- `OverlapSquared`

Parameters

`allow_partial` (`bool`, default: `False`) – If `False`, evaluation will fail when an unsupported computable node is encountered. If `True`, unsupported nodes will be skipped.

Raises

`NotImplementedError` – If an unsupported computable is encountered, and `allow_partial==False`.

Note

This evaluator works for quantum computables the protocol has been built from.

Returns

`Callable[[Evaluatable], Union[Evaluatable, Any]]` – A function that can evaluate quantum computables. If a computable is supported by this protocol, it is computed; otherwise, it returns the computable itself.

get_runner (*c*, *compile_symbolic=False*, **args*, ***kwargs*)

Returns an end-to-end executor function for a quantum computable.

Note

All nodes in the quantum computable must be evaluable by this protocol. That is, the leaf nodes need to be *OverlapSquared*.

Note

At every call of the returned function the internal state of the protocol changes.

Parameters

- ***c*** (*ComputableNode*) – The quantum computable node to be measured and evaluated at every call of the returned function.
- ***compile_symbolic* (bool, default: False)** – If `True`, circuits are immediately built and compiled symbolically. Calling the returned function substitutes symbols, runs circuits, and evaluates the output. If `False`, circuits are built and compiled only when the returned function is called and parameters are provided. Note that circuits may be deeper when `compile_symbolic=True`, because circuit compilation cannot perform optimizations based on numerical parameter values. The advantage of this case is that compilation only takes place once.
- ***args* (Any)**
- ***kwargs* (Any)**

Returns

`Callable[[Union[SymbolDict, Dict]], Any]` – A function that takes the parameters, builds and measures the necessary circuits, and returns the evaluated result. If the result is not a `float`, it returns `math.nan`.

get_shots()

Returns the number shots to be used for each circuit.

Returns

`List[int]` – List of number of shots, the length of this list is the same as the number of circuits.

property is_built: bool

Boolean flag indicating if the instance has been built.

Returns

`True` if the `build()` method has been successfully invoked, otherwise `False`.

```
property is_numeric: bool
```

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`False` if any built measurement circuit contains unsubstituted symbols, otherwise `True`.

```
property is_run: bool
```

Boolean flag indicating if the instance has been run.

Returns

`True` if the `run()` method has been successfully invoked, otherwise `False`.

```
property is_symbolic: bool
```

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`True` if any built measurement circuit contains unsubstituted symbols, otherwise `False`.

```
launch(*args, **kwargs)
```

Launch the circuits to the backend and return the handles for the results.

This method processes all the circuits and returns a list of `ResultHandle` objects representing the handles for the results.

Parameters

- `*args (Any)` – Additional arguments to be passed to `self.backend.process_circuits()`.
- `**kwargs (Any)` – Additional keyword arguments to be passed to `self.backend.process_circuits()`.

Returns

`List[ResultHandle]` – A list of pytket `ResultHandle` objects representing the handles for the launched circuits.

```
classmethod load(file, *args, **kwargs)
```

Load a pickled object from a file.

Parameters

- `file (Union[str, BinaryIO])` – The file path or file object to load the pickled data from.
- `*args (Any)` – Additional arguments passed to the class constructor.
- `**kwargs (Any)` – Additional keyword arguments passed to the class constructor.

Returns

`TypeVar(T, bound= PartiallyPickleable)` – An instance of the class with its state loaded from the pickled data.

```
classmethod loads(pickled_data, *args, **kwargs)
```

Load a pickled object from a bytes object.

Parameters

- `pickled_data (bytes)` – The pickled data to load the object from.
- `*args (Any)` – Additional arguments passed to the class constructor.
- `**kwargs (Any)` – Additional keyword arguments passed to the class constructor.

Returns

`TypeVar(T, bound= PartiallyPickleable)` – An instance of the class with its state loaded from the pickled data.

property n_circuit: int

Returns the total number of circuits.

rebuild(*args, **kwargs)

Rebuild the internal data structure.

It is equivalent to `clear().build(*args, **kwargs)`.

Parameters

- ***args (Any)** – Arguments to be passed to `build()`.
- ****kwargs (Any)** – Keyword arguments to be passed to `build()`.

Returns

`TypeVar(TBuildClearMixin, bound= BuildClearMixin)` – self.

retrieve(source, **kwargs)

Retrieve distributions from the backend for the given source.

If the `source` is a list of pytket ResultHandle, the distributions are retrieved using `self.backend.get_results()` method. If the `source` is a list of pytket BackendResult, it is assumed that the results are already provided.

Parameters

- **source (Union[List[ResultHandle], List[BackendResult]])** – A list of pytket ResultHandle or BackendResult objects representing the source of the distributions.
- ****kwargs (Any)** – Additional keyword arguments to be passed to `self.backend.get_results()` when retrieving results.

Returns

`TypeVar(T, bound= BaseOverlapSquaredProtocol)` – Returns self instance.

run(*args, **kwargs)

Run the protocol and waiting for the results.

This method executes the following steps: 1. Launches the measurement circuits to obtain the circuit handles.
2. Retrieves the measurement distributions using the circuit handles.

Parameters

- ***args (Any)** – Arguments to be passed to `launch()`.
- ****kwargs (Any)** – Keyword arguments to be passed to `launch()`.

Returns

`TypeVar(TLaunchRetrieveMixin, bound= LaunchRetrieveMixin)` – self.

27.11.3 Protocols for Overlaps

```
class HadamardTestOverlap(backend, shots_per_circuit, direct=False,
                           pauli_partition_strategy=PauliPartitionStrat.NonConflictingSets,
                           pauli_colour_method=GraphColourMethod.Lazy)
```

Bases: `BaseOverlapProtocol`

Calculates the overlap of two states using a specialized Hadamard test.

Computes the real or imaginary part of $\langle A|B\rangle$ where $|A\rangle = U_A|0\rangle$ using the “linear combination of unitaries” method from <https://journals.aps.org/prab/abstract/10.1103/PhysRevA.99.032331> (see figure 2). For `direct=True`, this method uses the approach from <https://iopscience.iop.org/article/10.1088/1367-2630/ab867b> to combine ancilla and state register measurements to compute overlaps with kernels.

Parameters

- `backend` (`Backend`) – The backend to use for quantum computations.
- `shots_per_circuit` (`int`) – Number of shots to perform.
- `direct` (`bool`, default: `False`) – Relevant for overlaps with non-identity kernels. If `True`, Pauli words are divided into simultaneously measurable sets and computed using direct measurement of expectation values on the state register.
- `pauli_partition_strategy` (`Optional[PauliPartitionStrat]`, default: `PauliPartitionStrat.NonConflictingSets`) – For `direct=True`. Strategy to partition Pauli operators.
- `pauli_colour_method` (`Optional[GraphColourMethod]`, default: `GraphColourMethod.Lazy`) – For `direct=True`. Method to perform graph colouring.

build (`parameters, bra_state, ket_state, *operators, component, optimisation_level=1`)

Builds and compiles the necessary circuits for computing an overlap.

Parameters

- `parameters` (`Union[SymbolDict, Dict]`) – Parameter values for the circuits.
- `bra_state` (`GeneralAnsatz`) – Left-hand state.
- `ket_state` (`GeneralAnsatz`) – Right-hand state.
- `operators` (`QubitOperator`) – Operator kernel, default is the identity. Coefficients in the kernel are ignored; only the Pauli strings are used in circuit construction.
- `component` (`Union[str, NumberType]`) – Component of overlap to measure: `real`, `imag`, or `complex`.
- `optimisation_level` (`int`, default: 1) – Passed to the backend’s `get_compiled_circuits()` method.

Returns

`BaseOverlapProtocol` – Self instance.

build_from (`parameters, computable, exclude=None`)

Builds the protocol based on given parameters and computable expression.

This method walks over the computable expression tree and calls the `build()` method for leaves of type `Overlap`, `OverlapReal`, and `OverlapImag`.

Parameters

- `parameters` (`Union[SymbolDict, Dict]`) – Parameters for building.
- `computable` (`Any`) – A root node of a computable expression tree, simply it is a computable expression.
- `exclude` (`Optional[Callable[[Any], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.

Returns

`BaseOverlapProtocol` – The modified instance after building.

```
classmethod build_protocols_from(parameters, computable, exclude=None, *args, **kwargs)
```

Build a list of protocols based on the given parameters and computable expression.

This method walks over the computable expression tree and collects all qubit operator kernels appearing in nodes of type:

- *Overlap*
- *OverlapReal*
- *OverlapImag*

After it has walked over the tree, it creates an instance of *HadamardTestOverlap* for each distinct bra, ket pair of states, and calls the *build()* method with the kernel operators associated with the states. It collects all created protocols into a *ProtocolList* object to be returned.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Values for the parameters in the computable expression.
- **computable** (`ComputableNode`) – A root node of a computable expression tree.
- **exclude** (`Optional[Callable[[ComputableNode], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.
- ***args** – Arguments passed to the constructor of *HadamardTestOverlap*.
- ****kwargs** – Keyword arguments passed to the constructor of *HadamardTestOverlap*.

Returns

`ProtocolList` – A list, containing all the newly instantiated and built protocols.

`circuits: Optional[List[Circuit]]`

`clear()`

Resets the internal state of the object by clearing all stored data.

This method clears all stored data including state hashes, operator hashes, measurement setups, parameters, and associated dataframes and circuits.

Returns

`BaseOverlapProtocol` – self.

`cost()`

Calculate a simple cost metric running the protocol.

The cost is calculated as the sum of the depth of each circuit multiplied by the number of shots.

Returns

`int` – The cost value as an integer.

`credits(syntax_checker=None, use_websocket=None)`

Evaluate an approximate cost for the measurement circuits built in Quantinuum credits.

Syntax checker will usually be automatically selected, but in some cases needs to be provided. If the measurement circuits are not built yet, the credits will be zero.

Note

This works only with QuantinuumBackend compatible backends.

Parameters

- **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 total cost in Quantinuum credits to run the circuits.

`dataframe_circuit_shot()`

Create a pandas `DataFrame` with circuit, shot, and depth information.

Returns

`DataFrame` – A pandas `DataFrame` containing the circuit, shot, and depth information.

`evaluate_overlap(bra_state, ket_state, kernel={(): 1.0})`

Evaluates the overlap for given states and kernel.

This method can only be performed if, prior to calling this method, the protocol has been built for `component='complex'`, with the same input states and kernel (or other operators composed of the same Pauli strings), and the protocol has also been run.

Parameters

- **bra_state** (`GeneralAnsatz`) – Left-hand state.
- **ket_state** (`GeneralAnsatz`) – Right-hand state.
- **kernel** (`QubitOperator`, default: `{() : 1.0}`) – Operator kernel.

Returns

`complex` – Real part of the overlap.

`evaluate_overlap_imag(bra_state, ket_state, kernel={(): 1.0})`

Evaluates the imaginary part of the overlap for given states and kernel.

This method can only be performed if, prior to calling this method, the protocol has been built for `component='imag'` or `'complex'`, with the same input states and kernel (or other operators composed of the same Pauli strings), and the protocol has also been run.

Parameters

- **bra_state** (`GeneralAnsatz`) – Left-hand state.
- **ket_state** (`GeneralAnsatz`) – Right-hand state.
- **kernel** (`QubitOperator`, default: `{() : 1.0}`) – Operator kernel.

Returns

`float` – Real part of the overlap.

`evaluate_overlap_real(bra_state, ket_state, kernel={(): 1.0})`

Evaluates the real part of the overlap for given states and kernel.

This method can only be performed if, prior to calling this method, the protocol has been built for `component='real'` or `'complex'`, with the same input states and kernel (or other operators composed of the same Pauli strings), and the protocol has also been run.

Parameters

- **bra_state** (`GeneralAnsatz`) – Left-hand state.
- **ket_state** (`GeneralAnsatz`) – Right-hand state.

- **kernel** (*QubitOperator*, default: `{ () : 1.0 }`) – Operator kernel.

Returns

`float` – Real part of the overlap.

get_circuits()

Returns the quantum circuits built for this protocol.

Returns

`List[Circuit]` – List of circuits.

get_circuitshots()

Generate the circuit shot pairs.

Yields

Pair of circuit and the associated number of shots.

Return type

`Iterator[CircuitShots]`

get_evaluator(*allow_partial=False*)

Returns an evaluator function to evaluate quantum computables.

This method creates and returns a function (evaluator) that takes in a specific quantum computable and evaluates it based on its type.

Quantum computables the returned evaluator can handle:

- *Overlap*
- *OverlapReal*
- *OverlapImag*

Parameters

`allow_partial` (`bool`, default: `False`) – If `False`, evaluation will fail when an unsupported computable node is encountered. If `True`, unsupported nodes will be skipped.

Note

This evaluator works for quantum computables the protocol has been built from.

Raises

`NotImplementedError` – If an unsupported computable is encountered, and `allow_partial==False`.

Returns

`Callable[[Evaluatable], Union[Evaluatable, Any]]` – A function that can evaluate quantum computables. If a computable is supported by this protocol, it is computed; otherwise, it returns the computable itself.

get_runner(*qc, compile_symbolic=False, *args, **kwargs*)

Returns an end-to-end executor function for a quantum computable.

All nodes in the quantum computable must be evaluable by this protocol. That is, the leaf nodes need to be of one of the supported types:

- *Overlap*

- *OverlapReal*
- *OverlapImag*

Note

At every call of the returned function the internal state of the protocol changes.

Parameters

- **qc** (`ComputableNode`) – The quantum computable node to be measured and evaluated at every call of the returned function.
- **compile_symbolic** (`bool`, default: `False`) – If `True`, circuits are immediately built and compiled symbolically. Calling the returned function substitutes symbols, runs circuits, and evaluates the output. If `False`, circuits are built and compiled only when the returned function is called and parameters are provided. Note that circuits may be deeper when `compile_symbolic=True`, because circuit compilation cannot perform optimizations based on numerical parameter values. The advantage of this case is that compilation only takes place once.
- **args** (`Any`)
- **kwargs** (`Any`)

Returns

`Callable[[Union[SymbolDict, Dict]], Any]` – A function that takes the parameters, builds and measures the necessary circuits, and returns the evaluated result. If the result is not a `float`, it returns `math.nan`.

`get_shots()`

Returns the number shots to be used for each circuit.

Returns

`List[int]` – List of number of shots, the length of this list is the same as the number of circuits.

`property is_built: bool`

Boolean flag indicating if the instance has been built.

Returns

`True` if the `build()` method has been successfully invoked, otherwise `False`.

`property is_numeric: bool`

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`False` if any built measurement circuit contains unsubstituted symbols, otherwise `True`.

`property is_run: bool`

Boolean flag indicating if the instance has been run.

Returns

`True` if the `run()` method has been successfully invoked, otherwise `False`.

`property is_symbolic: bool`

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`True` if any built measurement circuit contains unsubstituted symbols, otherwise `False`.

launch (*args, **kwargs)

Launch the circuits to the backend and return the handles for the results.

This method processes all the circuits and returns a list of `ResultHandle` objects representing the handles for the results.

Parameters

- `*args (Any)` – Additional arguments to be passed to `self.backend.process_circuits()`.
- `**kwargs (Any)` – Additional keyword arguments to be passed to `self.backend.process_circuits()`.

Returns

`List[ResultHandle]` – A list of pytket `ResultHandle` objects representing the handles for the launched circuits.

property n_circuit: int

Returns the total number of circuits.

rebuild (*args, **kwargs)

Rebuild the internal data structure.

It is equivalent to `clear().build(*args, **kwargs)`.

Parameters

- `*args (Any)` – Arguments to be passed to `build()`.
- `**kwargs (Any)` – Keyword arguments to be passed to `build()`.

Returns

`TypeVar(TBuildClearMixin, bound= BuildClearMixin)` – self.

retrieve (source, **kwargs)

Retrieve distributions from the backend for the given source.

If the `source` is a list of pytket `ResultHandle`, the distributions are retrieved using `self.backend.get_results()` method. If the `source` is a list of pytket `BackendResult`, it is assumed that the results are already provided.

Parameters

- `source (Union[List[ResultHandle], List[BackendResult]])` – A list of pytket `ResultHandle` or `BackendResult` objects representing the source of the distributions.
- `**kwargs (Any)` – Additional keyword arguments to be passed to `self.backend.get_results()` when retrieving results.

Returns

`BaseOverlapProtocol` – Returns self instance.

run (*args, **kwargs)

Run the protocol and waiting for the results.

This method executes the following steps: 1. Launches the measurement circuits to obtain the circuit handles.
2. Retrieves the measurement distributions using the circuit handles.

Parameters

- `*args (Any)` – Arguments to be passed to `launch()`.
- `**kwargs (Any)` – Keyword arguments to be passed to `launch()`.

Returns

TypeVar(TLaunchRetrieveMixin, bound= LaunchRetrieveMixin) – self.

```
class FactorizedOverlap(backend, shots_per_circuit, direct=False,
                       pauli_partition_strategy=PauliPartitionStrat.NonConflictingSets,
                       pauli_colour_method=GraphColourMethod.Lazy)
```

Bases: BaseOverlapProtocol

Abstract base class for specialized overlap protocols between ansatzes that factorize into a reference and a number conserving unitary.

Computes the real or imaginary part of $\langle A|B\rangle$ where $|A\rangle = U_A U_{\text{ref}} |0\rangle$ and $|B\rangle = U_B U_{\text{ref}} |0\rangle$ in the manner shown in Figures 1 and 2 of <https://journals.aps.org/prxquantum/abstract/10.1103/PRXQuantum.4.030307>. These protocols make use of the property that $U_A |0\rangle = |0\rangle$ to avoid the potentially large controlled state unitaries involved in e.g. [HadamardTestOverlap](#).

 **Warning**

Only ansatzes with the property that $U_A |0\rangle = |0\rangle$ are compatible with protocols of this type.

Parameters

- **backend** (Backend)
- **shots_per_circuit** (int)
- **direct** (bool, default: False)
- **pauli_partition_strategy** (Optional[PauliPartitionStrat], default: PauliPartitionStrat.NonConflictingSets)
- **pauli_colour_method** (Optional[GraphColourMethod], default: GraphColourMethod.Lazy)

build(parameters, bra_state, ket_state, *operators, component, optimisation_level=1)

Builds and compiles the necessary circuits for computing an overlap.

Parameters

- **parameters** (Union[SymbolDict, Dict]) – Parameter values for the circuits.
- **bra_state** (GeneralAnsatz) – Left-hand state.
- **ket_state** (GeneralAnsatz) – Right-hand state.
- **operators** (QubitOperator) – Operator kernel, default is the identity. Coefficients in the kernel are ignored; only the Pauli strings are used in circuit construction.
- **component** (Union[str, NumberType]) – Component of overlap to measure: real, imag, or complex.
- **optimisation_level** (int, default: 1) – Passed to the backend's get_compiled_circuits() method.

Returns

BaseOverlapProtocol – Self instance.

`build_from`(parameters, computable, exclude=None)

Builds the protocol based on given parameters and computable expression.

This method walks over the computable expression tree and calls the `build()` method for leaves of type `Overlap`, `OverlapReal`, and `OverlapImag`.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Parameters for building.
- **computable** (`Any`) – A root node of a computable expression tree, simply it is a computable expression.
- **exclude** (`Optional[Callable[[Any], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.

Returns

`BaseOverlapProtocol` – The modified instance after building.

`classmethod build_protocols_from`(parameters, computable, exclude=None, *args, **kwargs)

Build a list of protocols based on the given parameters and computable expression.

This method walks over the computable expression tree and collects all qubit operator kernels appearing in nodes of type:

- `Overlap`
- `OverlapReal`
- `OverlapImag`

After it has walked over the tree, it creates an instance of `HadamardTestOverlap` for each distinct bra, ket pair of states, and calls the `build()` method with the kernel operators associated with the states. It collects all created protocols into a `ProtocolList` object to be returned.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Values for the parameters in the computable expression.
- **computable** (`ComputableNode`) – A root node of a computable expression tree.
- **exclude** (`Optional[Callable[[ComputableNode], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.
- ***args** – Arguments passed to the constructor of `HadamardTestOverlap`.
- ****kwargs** – Keyword arguments passed to the constructor of `HadamardTestOverlap`.

Returns

`ProtocolList` – A list, containing all the newly instantiated and built protocols.

`circuits: Optional[List[Circuit]]`**`clear()`**

Resets the internal state of the object by clearing all stored data.

This method clears all stored data including state hashes, operator hashes, measurement setups, parameters, and associated dataframes and circuits.

Returns

`BaseOverlapProtocol` – self.

cost ()

Calculate a simple cost metric running the protocol.

The cost is calculated as the sum of the depth of each circuit multiplied by the number of shots.

Returns

`int` – The cost value as an integer.

credits (syntax_checker=None, use_websocket=None)

Evaluate an approximate cost for the measurement circuits built in Quantinuum credits.

Syntax checker will usually be automatically selected, but in some cases needs to be provided. If the measurement circuits are not built yet, the credits will be zero.

Note

This works only with `QuantinuumBackend` compatible backends.

Parameters

- `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 total cost in Quantinuum credits to run the circuits.

dataframe_circuit_shot ()

Create a pandas `DataFrame` with circuit, shot, and depth information.

Returns

`DataFrame` – A pandas `DataFrame` containing the circuit, shot, and depth information.

evaluate_overlap (bra_state, ket_state, kernel={(): 1.0})

Evaluates the overlap for given states and kernel.

This method can only be performed if, prior to calling this method, the protocol has been built for `component='complex'`, with the same input states and kernel (or other operators composed of the same Pauli strings), and the protocol has also been run.

Parameters

- `bra_state` (`GeneralAnsatz`) – Left-hand state.
- `ket_state` (`GeneralAnsatz`) – Right-hand state.
- `kernel` (`QubitOperator`, default: `{() : 1.0}`) – Operator kernel.

Returns

`complex` – Real part of the overlap.

evaluate_overlap_imag (bra_state, ket_state, kernel={(): 1.0})

Evaluates the imaginary part of the overlap for given states and kernel.

This method can only be performed if, prior to calling this method, the protocol has been built for `component='imag'` or `'complex'`, with the same input states and kernel (or other operators composed of the same Pauli strings), and the protocol has also been run.

Parameters

- **bra_state** (*GeneralAnsatz*) – Left-hand state.
- **ket_state** (*GeneralAnsatz*) – Right-hand state.
- **kernel** (*QubitOperator*, default: `{ () : 1.0 }`) – Operator kernel.

Returns

`float` – Real part of the overlap.

evaluate_overlap_real (*bra_state*, *ket_state*, *kernel*=`{ () : 1.0 }`)

Evaluates the real part of the overlap for given states and kernel.

This method can only be performed if, prior to calling this method, the protocol has been built for component='real' or 'complex', with the same input states and kernel (or other operators composed of the same Pauli strings), and the protocol has also been run.

Parameters

- **bra_state** (*GeneralAnsatz*) – Left-hand state.
- **ket_state** (*GeneralAnsatz*) – Right-hand state.
- **kernel** (*QubitOperator*, default: `{ () : 1.0 }`) – Operator kernel.

Returns

`float` – Real part of the overlap.

get_circuits ()

Returns the quantum circuits built for this protocol.

Returns

`List[Circuit]` – List of circuits.

get_circuitshots ()

Generate the circuit shot pairs.

Yields

Pair of circuit and the associated number of shots.

Return type

`Iterator[CircuitShots]`

get_evaluator (*allow_partial=False*)

Returns an evaluator function to evaluate quantum computables.

This method creates and returns a function (evaluator) that takes in a specific quantum computable and evaluates it based on its type.

Quantum computables the returned evaluator can handle:

- *Overlap*
- *OverlapReal*
- *OverlapImag*

Parameters

allow_partial (`bool`, default: `False`) – If `False`, evaluation will fail when an unsupported computable node is encountered. If `True`, unsupported nodes will be skipped.

Note

This evaluator works for quantum computables the protocol has been built from.

Raises

`NotImplementedError` – If an unsupported computable is encountered, and `allow_partial==False`.

Returns

`Callable[[Evaluatable], Union[Evaluatable, Any]]` – A function that can evaluate quantum computables. If a computable is supported by this protocol, it is computed; otherwise, it returns the computable itself.

`get_runner(qc, compile_symbolic=False, *args, **kwargs)`

Returns an end-to-end executor function for a quantum computable.

All nodes in the quantum computable must be evaluable by this protocol. That is, the leaf nodes need to be of one of the supported types:

- `Overlap`
- `OverlapReal`
- `OverlapImag`

Note

At every call of the returned function the internal state of the protocol changes.

Parameters

- `qc` (`ComputableNode`) – The quantum computable node to be measured and evaluated at every call of the returned function.
- `compile_symbolic` (`bool`, default: `False`) – If `True`, circuits are immediately built and compiled symbolically. Calling the returned function substitutes symbols, runs circuits, and evaluates the output. If `False`, circuits are built and compiled only when the returned function is called and parameters are provided. Note that circuits may be deeper when `compile_symbolic=True`, because circuit compilation cannot perform optimizations based on numerical parameter values. The advantage of this case is that compilation only takes place once.
- `args` (`Any`)
- `kwargs` (`Any`)

Returns

`Callable[[Union[SymbolDict, Dict]], Any]` – A function that takes the parameters, builds and measures the necessary circuits, and returns the evaluated result. If the result is not a `float`, it returns `math.nan`.

`get_shots()`

Returns the number shots to be used for each circuit.

Returns

`List[int]` – List of number of shots, the length of this list is the same as the number of circuits.

property is_built: bool

Boolean flag indicating if the instance has been built.

Returns

True if the `build()` method has been successfully invoked, otherwise False.

property is_numeric: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

False if any built measurement circuit contains unsubstituted symbols, otherwise True.

property is_run: bool

Boolean flag indicating if the instance has been run.

Returns

True if the `run()` method has been successfully invoked, otherwise False.

property is_symbolic: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

True if any built measurement circuit contains unsubstituted symbols, otherwise False.

launch(*args, **kwargs)

Launch the circuits to the backend and return the handles for the results.

This method processes all the circuits and returns a list of `ResultHandle` objects representing the handles for the results.

Parameters

- ***args (Any)** – Additional arguments to be passed to `self.backend.process_circuits()`.
- ****kwargs (Any)** – Additional keyword arguments to be passed to `self.backend.process_circuits()`.

Returns

`List[ResultHandle]` – A list of pytket `ResultHandle` objects representing the handles for the launched circuits.

property n_circuit: int

Returns the total number of circuits.

rebuild(*args, **kwargs)

Rebuild the internal data structure.

It is equivalent to `clear().build(*args, **kwargs)`.

Parameters

- ***args (Any)** – Arguments to be passed to `build()`.
- ****kwargs (Any)** – Keyword arguments to be passed to `build()`.

Returns

`TypeVar(TBuildClearMixin, bound= BuildClearMixin)` – self.

retrieve(source, **kwargs)

Retrieve distributions from the backend for the given source.

If the source is a list of pytket ResultHandle, the distributions are retrieved using self.backend.get_results() method. If the source is a list of pytket BackendResult, it is assumed that the results are already provided.

Parameters

- **source** (`Union[List[ResultHandle], List[BackendResult]]`) – A list of pytket ResultHandle or BackendResult objects representing the source of the distributions.
- ****kwargs** (`Any`) – Additional keyword arguments to be passed to self.backend.get_results() when retrieving results.

Returns

`BaseOverlapProtocol` – Returns self instance.

run(*args, **kwargs)

Run the protocol and waiting for the results.

This method executes the following steps: 1. Launches the measurement circuits to obtain the circuit handles. 2. Retrieves the measurement distributions using the circuit handles.

Parameters

- ***args** (`Any`) – Arguments to be passed to `launch()`.
- ****kwargs** (`Any`) – Keyword arguments to be passed to `launch()`.

Returns

`TypeVar(TLaunchRetrieveMixin, bound=LaunchRetrieveMixin)` – self.

class SwapFactorizedOverlap(backend, shots_per_circuit, direct=False, pauli_partition_strategy=PauliPartitionStrat.NonConflictingSets, pauli_colour_method=GraphColourMethod.Lazy)

Bases: `FactorizedOverlap`

Uses an ancillary state register to obviate controlled ansatz unitaries.

Computes the real or imaginary part of $\langle A|B \rangle$ where $|A\rangle = U_A U_{\text{ref}} |0\rangle$ and $|B\rangle = U_B U_{\text{ref}} |0\rangle$ in the manner shown in Figure 1 of <https://journals.aps.org/prxquantum/abstract/10.1103/PRXQuantum.4.030307>.

Parameters

- **backend** (`Backend`) – The backend to use for quantum computations.
- **shots_per_circuit** (`int`) – Number of shots to perform.
- **direct** (`bool`, default: `False`) – Relevant for overlaps with non-identity kernels. If `True`, Pauli words are divided into simultaneously measurable sets and computed using direct measurement of expectation values on the state register.
- **pauli_partition_strategy** (`Optional[PauliPartitionStrat]`, default: `PauliPartitionStrat.NonConflictingSets`) – For `direct=True`. Strategy to partition Pauli operators.
- **pauli_colour_method** (`Optional[GraphColourMethod]`, default: `GraphColourMethod.Lazy`) – For `direct=True`. Method to perform graph colouring.

build(parameters, bra_state, ket_state, *operators, component, optimisation_level=1)

Builds and compiles the necessary circuits for computing an overlap.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Parameter values for the circuits.
- **bra_state** (`GeneralAnsatz`) – Left-hand state.
- **ket_state** (`GeneralAnsatz`) – Right-hand state.
- **operators** (`QubitOperator`) – Operator kernel, default is the identity. Coefficients in the kernel are ignored; only the Pauli strings are used in circuit construction.
- **component** (`Union[str, NumberType]`) – Component of overlap to measure: `real`, `imag`, or `complex`.
- **optimisation_level** (`int`, default: 1) – Passed to the backend's `get_compiled_circuits()` method.

Returns

`BaseOverlapProtocol` – Self instance.

`build_from` (`parameters, computable, exclude=None`)

Builds the protocol based on given parameters and computable expression.

This method walks over the computable expression tree and calls the `build()` method for leaves of type `Overlap`, `OverlapReal`, and `OverlapImag`.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Parameters for building.
- **computable** (`Any`) – A root node of a computable expression tree, simply it is a computable expression.
- **exclude** (`Optional[Callable[[Any], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.

Returns

`BaseOverlapProtocol` – The modified instance after building.

`classmethod build_protocols_from` (`parameters, computable, exclude=None, *args, **kwargs`)

Build a list of protocols based on the given parameters and computable expression.

This method walks over the computable expression tree and collects all qubit operator kernels appearing in nodes of type:

- `Overlap`
- `OverlapReal`
- `OverlapImag`

After it has walked over the tree, it creates an instance of `HadamardTestOverlap` for each distinct bra, ket pair of states, and calls the `build()` method with the kernel operators associated with the states. It collects all created protocols into a `ProtocolList` object to be returned.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Values for the parameters in the computable expression.
- **computable** (`ComputableNode`) – A root node of a computable expression tree.
- **exclude** (`Optional[Callable[[ComputableNode], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.
- ***args** – Arguments passed to the constructor of `HadamardTestOverlap`.
- ****kwargs** – Keyword arguments passed to the constructor of `HadamardTestOverlap`.

Returns

`ProtocolList` – A list, containing all the newly instantiated and built protocols.

`circuits: Optional[List[Circuit]]`

`clear()`

Resets the internal state of the object by clearing all stored data.

This method clears all stored data including state hashes, operator hashes, measurement setups, parameters, and associated dataframes and circuits.

Returns

`BaseOverlapProtocol` – self.

`cost()`

Calculate a simple cost metric running the protocol.

The cost is calculated as the sum of the depth of each circuit multiplied by the number of shots.

Returns

`int` – The cost value as an integer.

`credits (syntax_checker=None, use_websocket=None)`

Evaluate an approximate cost for the measurement circuits built in Quantinuum credits.

Syntax checker will usually be automatically selected, but in some cases needs to be provided. If the measurement circuits are not built yet, the credits will be zero.

 **Note**

This works only with QuantinuumBackend compatible backends.

Parameters

- `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 total cost in Quantinuum credits to run the circuits.

`dataframe_circuit_shot()`

Create a pandas `DataFrame` with circuit, shot, and depth information.

Returns

`DataFrame` – A pandas `DataFrame` containing the circuit, shot, and depth information.

`evaluate_overlap(bra_state, ket_state, kernel={(): 1.0})`

Evaluates the overlap for given states and kernel.

This method can only be performed if, prior to calling this method, the protocol has been built for `component='complex'`, with the same input states and kernel (or other operators composed of the same Pauli strings), and the protocol has also been run.

Parameters

- `bra_state` (`GeneralAnsatz`) – Left-hand state.
- `ket_state` (`GeneralAnsatz`) – Right-hand state.

- **kernel** (*QubitOperator*, default: `{(): 1.0}`) – Operator kernel.

Returns

`complex` – Real part of the overlap.

evaluate_overlap_imag (*bra_state*, *ket_state*, *kernel*=`{(): 1.0}`)

Evaluates the imaginary part of the overlap for given states and kernel.

This method can only be performed if, prior to calling this method, the protocol has been built for component='imag' or 'complex', with the same input states and kernel (or other operators composed of the same Pauli strings), and the protocol has also been run.

Parameters

- **bra_state** (*GeneralAnsatz*) – Left-hand state.
- **ket_state** (*GeneralAnsatz*) – Right-hand state.
- **kernel** (*QubitOperator*, default: `{(): 1.0}`) – Operator kernel.

Returns

`float` – Real part of the overlap.

evaluate_overlap_real (*bra_state*, *ket_state*, *kernel*=`{(): 1.0}`)

Evaluates the real part of the overlap for given states and kernel.

This method can only be performed if, prior to calling this method, the protocol has been built for component='real' or 'complex', with the same input states and kernel (or other operators composed of the same Pauli strings), and the protocol has also been run.

Parameters

- **bra_state** (*GeneralAnsatz*) – Left-hand state.
- **ket_state** (*GeneralAnsatz*) – Right-hand state.
- **kernel** (*QubitOperator*, default: `{(): 1.0}`) – Operator kernel.

Returns

`float` – Real part of the overlap.

get_circuits ()

Returns the quantum circuits built for this protocol.

Returns

`List[Circuit]` – List of circuits.

get_circuitshots ()

Generate the circuit shot pairs.

Yields

Pair of circuit and the associated number of shots.

Return type

`Iterator[CircuitShots]`

get_evaluator (*allow_partial=False*)

Returns an evaluator function to evaluate quantum computables.

This method creates and returns a function (evaluator) that takes in a specific quantum computable and evaluates it based on its type.

Quantum computables the returned evaluator can handle:

- *Overlap*

- *OverlapReal*
- *OverlapImag*

Parameters

`allow_partial` (bool, default: False) – If False, evaluation will fail when an unsupported computable node is encountered. If True, unsupported nodes will be skipped.

Note

This evaluator works for quantum computables the protocol has been built from.

Raises

`NotImplementedError` – If an unsupported computable is encountered, and `allow_partial==False`.

Returns

`Callable[[Evaluable], Union[Evaluable, Any]]` – A function that can evaluate quantum computables. If a computable is supported by this protocol, it is computed; otherwise, it returns the computable itself.

`get_runner` (*qc*, `compile_symbolic=False`, **args*, ***kwargs*)

Returns an end-to-end executor function for a quantum computable.

All nodes in the quantum computable must be evaluable by this protocol. That is, the leaf nodes need to be of one of the supported types:

- *Overlap*
- *OverlapReal*
- *OverlapImag*

Note

At every call of the returned function the internal state of the protocol changes.

Parameters

- `qc` (`ComputableNode`) – The quantum computable node to be measured and evaluated at every call of the returned function.
- `compile_symbolic` (bool, default: False) – If True, circuits are immediately built and compiled symbolically. Calling the returned function substitutes symbols, runs circuits, and evaluates the output. If False, circuits are built and compiled only when the returned function is called and parameters are provided. Note that circuits may be deeper when `compile_symbolic=True`, because circuit compilation cannot perform optimizations based on numerical parameter values. The advantage of this case is that compilation only takes place once.
- `args` (`Any`)
- `kwargs` (`Any`)

Returns

`Callable[[Union[SymbolDict, Dict]], Any]` – A function that takes the parameters, builds

and measures the necessary circuits, and returns the evaluated result. If the result is not a `float`, it returns `math.nan`.

`get_shots()`

Returns the number shots to be used for each circuit.

>Returns

`List[int]` – List of number of shots, the length of this list is the same as the number of circuits.

`property is_built: bool`

Boolean flag indicating if the instance has been built.

>Returns

`True` if the `build()` method has been successfully invoked, otherwise `False`.

`property is_numeric: bool`

Boolean flag indicating if the measurement circuits contain free symbols.

>Returns

`False` if any built measurement circuit contains unsubstituted symbols, otherwise `True`.

`property is_run: bool`

Boolean flag indicating if the instance has been run.

>Returns

`True` if the `run()` method has been successfully invoked, otherwise `False`.

`property is_symbolic: bool`

Boolean flag indicating if the measurement circuits contain free symbols.

>Returns

`True` if any built measurement circuit contains unsubstituted symbols, otherwise `False`.

`launch(*args, **kwargs)`

Launch the circuits to the backend and return the handles for the results.

This method processes all the circuits and returns a list of `ResultHandle` objects representing the handles for the results.

Parameters

- `*args (Any)` – Additional arguments to be passed to `self.backend.process_circuits()`.
- `**kwargs (Any)` – Additional keyword arguments to be passed to `self.backend.process_circuits()`.

>Returns

`List[ResultHandle]` – A list of pytket `ResultHandle` objects representing the handles for the launched circuits.

`property n_circuit: int`

Returns the total number of circuits.

`rebuild(*args, **kwargs)`

Rebuild the internal data structure.

It is equivalent to `clear().build(*args, **kwargs)`.

Parameters

- `*args (Any)` – Arguments to be passed to `build()`.

- ****kwargs** (Any) – Keyword arguments to be passed to `build()`.

Returns

`TypeVar(TBuildClearMixin, bound= BuildClearMixin)` – self.

retrieve (*source*, ***kwargs*)

Retrieve distributions from the backend for the given source.

If the *source* is a list of pytket ResultHandle, the distributions are retrieved using `self.backend.get_results()` method. If the *source* is a list of pytket BackendResult, it is assumed that the results are already provided.

Parameters

- **source** (Union[List[ResultHandle], List[BackendResult]]) – A list of pytket ResultHandle or BackendResult objects representing the source of the distributions.
- ****kwargs** (Any) – Additional keyword arguments to be passed to `self.backend.get_results()` when retrieving results.

Returns

`BaseOverlapProtocol` – Returns self instance.

run (**args*, ***kwargs*)

Run the protocol and waiting for the results.

This method executes the following steps: 1. Launches the measurement circuits to obtain the circuit handles.
2. Retrieves the measurement distributions using the circuit handles.

Parameters

- ***args** (Any) – Arguments to be passed to `launch()`.
- ****kwargs** (Any) – Keyword arguments to be passed to `launch()`.

Returns

`TypeVar(TLaunchRetrieveMixin, bound= LaunchRetrieveMixin)` – self.

class ComputeUncomputeFactorizedOverlap (*backend*, *shots_per_circuit*)

Bases: `FactorizedOverlap`

Computes the ket and uncomputes the bra to construct the required linear combination on a single state register.

Computes the real or imaginary part of $\langle A | B \rangle$ where $|A\rangle = U_A U_{\text{ref}} |0\rangle$ and $|B\rangle = U_B U_{\text{ref}} |0\rangle$ in the manner shown in Figure 2 of <https://journals.aps.org/prxquantum/abstract/10.1103/PRXQuantum.4.030307>.

This protocol is not compatible with the “direct” operator averaging approach.

Parameters

- **backend** (Backend) – The backend to use for quantum computations.
- **shots_per_circuit** (int) – Number of shots to perform.

build (*parameters*, *bra_state*, *ket_state*, **operators*, *component*, *optimisation_level*=1)

Builds and compiles the necessary circuits for computing an overlap.

Parameters

- **parameters** (Union[SymbolDict, Dict]) – Parameter values for the circuits.
- **bra_state** (GeneralAnsatz) – Left-hand state.
- **ket_state** (GeneralAnsatz) – Right-hand state.

- **operators** (*QubitOperator*) – Operator kernel, default is the identity. Coefficients in the kernel are ignored; only the Pauli strings are used in circuit construction.
- **component** (*Union[str, NumberType]*) – Component of overlap to measure: `real`, `imag`, or `complex`.
- **optimisation_level** (*int*, default: 1) – Passed to the backend's `get_compiled_circuits()` method.

Returns

`BaseOverlapProtocol` – Self instance.

build_from(*parameters, computable, exclude=None*)

Builds the protocol based on given parameters and computable expression.

This method walks over the computable expression tree and calls the `build()` method for leaves of type `Overlap`, `OverlapReal`, and `OverlapImag`.

Parameters

- **parameters** (*Union[SymbolDict, Dict]*) – Parameters for building.
- **computable** (*Any*) – A root node of a computable expression tree, simply it is a computable expression.
- **exclude** (*Optional[Callable[[Any], bool]]*, default: `None`) – Optional callable function to exclude certain nodes from processing.

Returns

`BaseOverlapProtocol` – The modified instance after building.

classmethod build_protocols_from(*parameters, computable, exclude=None, *args, **kwargs*)

Build a list of protocols based on the given parameters and computable expression.

This method walks over the computable expression tree and collects all qubit operator kernels appearing in nodes of type:

- `Overlap`
- `OverlapReal`
- `OverlapImag`

After it has walked over the tree, it creates an instance of `HadamardTestOverlap` for each distinct bra, ket pair of states, and calls the `build()` method with the kernel operators associated with the states. It collects all created protocols into a `ProtocolList` object to be returned.

Parameters

- **parameters** (*Union[SymbolDict, Dict]*) – Values for the parameters in the computable expression.
- **computable** (*ComputableNode*) – A root node of a computable expression tree.
- **exclude** (*Optional[Callable[[ComputableNode], bool]]*, default: `None`) – Optional callable function to exclude certain nodes from processing.
- ***args** – Arguments passed to the constructor of `HadamardTestOverlap`.
- ****kwargs** – Keyword arguments passed to the constructor of `HadamardTestOverlap`.

Returns

`ProtocolList` – A list, containing all the newly instantiated and built protocols.

`circuits: Optional[List[Circuit]]`

`clear()`

Resets the internal state of the object by clearing all stored data.

This method clears all stored data including state hashes, operator hashes, measurement setups, parameters, and associated dataframes and circuits.

Returns

`BaseOverlapProtocol – self.`

`cost()`

Calculate a simple cost metric running the protocol.

The cost is calculated as the sum of the depth of each circuit multiplied by the number of shots.

Returns

`int – The cost value as an integer.`

`credits(syntax_checker=None, use_websocket=None)`

Evaluate an approximate cost for the measurement circuits built in Quantinuum credits.

Syntax checker will usually be automatically selected, but in some cases needs to be provided. If the measurement circuits are not built yet, the credits will be zero.

Note

This works only with QuantinuumBackend compatible backends.

Parameters

- `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 total cost in Quantinuum credits to run the circuits.`

`dataframe_circuit_shot()`

Create a pandas DataFrame with circuit, shot, and depth information.

Returns

`DataFrame – A pandas DataFrame containing the circuit, shot, and depth information.`

`evaluate_overlap(bra_state, ket_state, kernel={(): 1.0})`

Evaluates the overlap for given states and kernel.

This method can only be performed if, prior to calling this method, the protocol has been built for `component='complex'`, with the same input states and kernel (or other operators composed of the same Pauli strings), and the protocol has also been run.

Parameters

- `bra_state` (`GeneralAnsatz`) – Left-hand state.
- `ket_state` (`GeneralAnsatz`) – Right-hand state.
- `kernel` (`QubitOperator`, default: `{() : 1.0}`) – Operator kernel.

Returns

`complex` – Real part of the overlap.

evaluate_overlap_imag (*bra_state*, *ket_state*, *kernel*=`{(): 1.0}`)

Evaluates the imaginary part of the overlap for given states and kernel.

This method can only be performed if, prior to calling this method, the protocol has been built for `component='imag'` or `'complex'`, with the same input states and kernel (or other operators composed of the same Pauli strings), and the protocol has also been run.

Parameters

- **bra_state** (*GeneralAnsatz*) – Left-hand state.
- **ket_state** (*GeneralAnsatz*) – Right-hand state.
- **kernel** (*QubitOperator*, default: `{(): 1.0}`) – Operator kernel.

Returns

`float` – Real part of the overlap.

evaluate_overlap_real (*bra_state*, *ket_state*, *kernel*=`{(): 1.0}`)

Evaluates the real part of the overlap for given states and kernel.

This method can only be performed if, prior to calling this method, the protocol has been built for `component='real'` or `'complex'`, with the same input states and kernel (or other operators composed of the same Pauli strings), and the protocol has also been run.

Parameters

- **bra_state** (*GeneralAnsatz*) – Left-hand state.
- **ket_state** (*GeneralAnsatz*) – Right-hand state.
- **kernel** (*QubitOperator*, default: `{(): 1.0}`) – Operator kernel.

Returns

`float` – Real part of the overlap.

get_circuits ()

Returns the quantum circuits built for this protocol.

Returns

`List[Circuit]` – List of circuits.

get_circuitshots ()

Generate the circuit shot pairs.

Yields

Pair of circuit and the associated number of shots.

Return type

`Iterator[CircuitShots]`

get_evaluator (*allow_partial=False*)

Returns an evaluator function to evaluate quantum computables.

This method creates and returns a function (evaluator) that takes in a specific quantum computable and evaluates it based on its type.

Quantum computables the returned evaluator can handle:

- `Overlap`
- `OverlapReal`

- *OverlapImag*

Parameters

`allow_partial` (bool, default: False) – If False, evaluation will fail when an unsupported computable node is encountered. If True, unsupported nodes will be skipped.

Note

This evaluator works for quantum computables the protocol has been built from.

Raises

`NotImplementedError` – If an unsupported computable is encountered, and `allow_partial==False`.

Returns

`Callable[[Evaluatable], Union[Evaluatable, Any]]` – A function that can evaluate quantum computables. If a computable is supported by this protocol, it is computed; otherwise, it returns the computable itself.

`get_runner` (`qc, compile_symbolic=False, *args, **kwargs`)

Returns an end-to-end executor function for a quantum computable.

All nodes in the quantum computable must be evaluable by this protocol. That is, the leaf nodes need to be of one of the supported types:

- *Overlap*
- *OverlapReal*
- *OverlapImag*

Note

At every call of the returned function the internal state of the protocol changes.

Parameters

- `qc` (`ComputableNode`) – The quantum computable node to be measured and evaluated at every call of the returned function.
- `compile_symbolic` (bool, default: False) – If True, circuits are immediately built and compiled symbolically. Calling the returned function substitutes symbols, runs circuits, and evaluates the output. If False, circuits are built and compiled only when the returned function is called and parameters are provided. Note that circuits may be deeper when `compile_symbolic=True`, because circuit compilation cannot perform optimizations based on numerical parameter values. The advantage of this case is that compilation only takes place once.
- `args` (`Any`)
- `kwargs` (`Any`)

Returns

`Callable[[Union[SymbolDict, Dict]], Any]` – A function that takes the parameters, builds and measures the necessary circuits, and returns the evaluated result. If the result is not a `float`, it returns `math.nan`.

get_shots()

Returns the number shots to be used for each circuit.

Returns

`List[int]` – List of number of shots, the length of this list is the same as the number of circuits.

property is_built: bool

Boolean flag indicating if the instance has been built.

Returns

`True` if the `build()` method has been successfully invoked, otherwise `False`.

property is_numeric: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`False` if any built measurement circuit contains unsubstituted symbols, otherwise `True`.

property is_run: bool

Boolean flag indicating if the instance has been run.

Returns

`True` if the `run()` method has been successfully invoked, otherwise `False`.

property is_symbolic: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`True` if any built measurement circuit contains unsubstituted symbols, otherwise `False`.

launch(*args, **kwargs)

Launch the circuits to the backend and return the handles for the results.

This method processes all the circuits and returns a list of `ResultHandle` objects representing the handles for the results.

Parameters

- `*args (Any)` – Additional arguments to be passed to `self.backend.process_circuits()`.
- `**kwargs (Any)` – Additional keyword arguments to be passed to `self.backend.process_circuits()`.

Returns

`List[ResultHandle]` – A list of pytket `ResultHandle` objects representing the handles for the launched circuits.

property n_circuit: int

Returns the total number of circuits.

rebuild(*args, **kwargs)

Rebuild the internal data structure.

It is equivalent to `clear().build(*args, **kwargs)`.

Parameters

- `*args (Any)` – Arguments to be passed to `build()`.
- `**kwargs (Any)` – Keyword arguments to be passed to `build()`.

Returns

`TypeVar(TBuildClearMixin, bound= BuildClearMixin) – self.`

retrieve (source, **kwargs)

Retrieve distributions from the backend for the given source.

If the `source` is a list of pytket ResultHandle, the distributions are retrieved using `self.backend.get_results()` method. If the `source` is a list of pytket BackendResult, it is assumed that the results are already provided.

Parameters

- `source (Union[List[ResultHandle], List[BackendResult]])` – A list of pytket ResultHandle or BackendResult objects representing the source of the distributions.
- `**kwargs (Any)` – Additional keyword arguments to be passed to `self.backend.get_results()` when retrieving results.

Returns

`BaseOverlapProtocol` – Returns self instance.

run (*args, **kwargs)

Run the protocol and waiting for the results.

This method executes the following steps: 1. Launches the measurement circuits to obtain the circuit handles. 2. Retrieves the measurement distributions using the circuit handles.

Parameters

- `*args (Any)` – Arguments to be passed to `launch ()`.
- `**kwargs (Any)` – Keyword arguments to be passed to `launch ()`.

Returns

`TypeVar(TLaunchRetrieveMixin, bound= LaunchRetrieveMixin) – self.`

27.11.4 Statevector-Based Protocols

`class SparseStatevectorProtocol(backend, cache=None, caching_decorator=cached)`

Bases: `_BaseStatevectorProtocol, EvaluatorRunnerMixin`

Protocol for sparse statevector calculations using caching.

Uses an (optionally) externally provided cache-handling class and an external caching decorator to wrap its methods to be cached.

Parameters

- `backend (Backend)` – The backend to use for quantum computations.
- `cache (Optional[Cache], default: None)` – A cache-handling class object. If None, `ProtocolCache` is implicitly instantiated and used.
- `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. By default, a function decorator local to the class module is used.

property cache: Cache

Returns the cache-handling object.

`cache_hit_report()`

Returns a cache hit report in the pandas DataFrame format.

Return type

`Optional[DataFrame]`

`clear(keep_cache=False)`

Resets the internal state of the object by clearing all stored data.

Sets the backend to `None`, and clears the cache.

Parameters

`keep_cache (bool, default: False)` – Whether to keep old cache.

Return type

`None`

`copy()`

Returns a deep copy of the protocol.

Return type

`_BaseStatevectorProtocol`

`get_evaluator(parameters, allow_partial=True)`

Returns an evaluator function to evaluate quantum computables.

This method creates and returns a function (evaluator) that takes in a specific quantum computable and evaluates it based on its type. The returned evaluator handles various computable types, such as expectation values, their derivatives, overlaps, etc.

Quantum computables the returned evaluator can handle:

- `ExpectationValue`
- `ExpectationValueNonHermitian`
- `ExpectationValueDerivative`
- `ExpectationValueBraDerivativeReal`
- `ExpectationValueBraDerivativeImag`
- `ExpectationValueKetDerivativeReal`
- `ExpectationValueKetDerivativeImag`
- `MetricTensorReal`
- `Overlap`
- `OverlapImag`
- `OverlapReal`
- `OverlapSquared`

Parameters

- `parameters (SymbolDict)` – Symbols and their values to be used in the evaluation.
- `allow_partial (bool, default: True)` – If `False`, evaluation will fail when an unsupported computable node is encountered. If `True`, unsupported nodes will be skipped.

Raises

`NotImplementedError` – If an unsupported computable is encountered, and `allow_partial==False`.

Returns

`Callable[[Any], Any]` – A function that can evaluate quantum computables. If the Computable is supported by this protocol, it is computed; otherwise, it returns the Computable itself.

get_runner (qc)

Returns an end-to-end executor function for a quantum computable.

All nodes in the quantum computable must be evaluable by this protocol. Supported computable are:

- `ExpectationValue`
- `ExpectationValueNonHermitian`
- `ExpectationValueDerivative`
- `ExpectationValueBraDerivativeReal`
- `ExpectationValueBraDerivativeImag`
- `ExpectationValueKetDerivativeReal`
- `ExpectationValueKetDerivativeImag`
- `MetricTensorReal`
- `Overlap`
- `OverlapImag`
- `OverlapReal`
- `OverlapSquared`

Parameters

`qc (ComputableNode)` – The quantum computable node to be measured and evaluated at every call of the returned function.

Returns

`Callable[[SymbolDict], Any]` – A function that takes the parameters and returns the evaluated result. If the result is not a `float`, it returns `math.nan`.

class BackendStatevectorProtocol (backend)

Bases: `_BaseStatevectorProtocol`

Statevector protocol utilising backend functionality to enable statevector simulations.

Parameters

`backend (Backend)` – The backend to use for quantum computations.

copy ()

Returns a deep copy of the protocol.

Return type

`_BaseStatevectorProtocol`

get_evaluator (parameters, allow_partial=True)

Returns an evaluator function to evaluate quantum computables.

This method creates and returns a function (evaluator) that takes in a specific quantum computable and evaluates it based on its type. The returned evaluator handles various computable types, such as expectation values, their derivatives, overlaps, etc.

Quantum computables the returned evaluator can handle:

- *ExpectationValue*
- *ExpectationValueNonHermitian*

Parameters

- **parameters** (*SymbolDict*) – Symbols and their values to be used in the evaluation.
- **allow_partial** (*bool*, default: `True`) – If `False`, evaluation will fail when an unsupported computable node is encountered. If `True`, unsupported nodes will be skipped.

Raises

`NotImplementedError` – If an unsupported computable is encountered, and `allow_partial==False`.

Returns

`Callable[[Any], Any]` – A function that evaluates quantum computables. If the Computable is supported by this protocol, it is computed; otherwise, it returns the Computable itself.

`get_runner(qc)`

Returns an end-to-end executor function for a quantum computable.

All nodes in the quantum computable must be evaluable by this protocol. Supported computables are:

- *ExpectationValue*
- *ExpectationValueNonHermitian*

Parameters

`qc` (*ComputableNode*) – The quantum computable node to be measured and evaluated at every call of the returned function.

Returns

`Callable[[SymbolDict], Any]` – A function that takes the parameters and returns the evaluated result. If the result is not a `float`, it returns `math.nan`.

`class SymbolicProtocol(simplifier=None)`

Bases: `EvaluatorRunnerMixin`

Protocol for statevector calculations using Sympy symbolic evaluation.

Note

This protocol caches the symbolic states, therefore one evaluation might be slow, but subsequent evaluations cost less time.

Parameters

`simplifier` (`Callable[[Expr], Expr]`, default: `None`) – A sympy simplifier that will be applied on the results before substitution, default is `None`.

`clear()`

Resets the internal state of the object by clearing all stored data.

Clears runtime auxiliary variables in the protocol.

Return type

`None`

get_evaluator(parameters, allow_partial=True)

Returns an evaluator function to evaluate quantum computables.

This method creates and returns a function (evaluator) that takes in a specific quantum computable and evaluates it based on its type. The returned evaluator handles various computable types, such as expectation values, their derivatives, overlaps, etc.

Quantum computables the returned evaluator can handle:

- *ExpectationValue*
- *ExpectationValueNonHermitian*
- *ExpectationValueDerivative*
- *ExpectationValueBraDerivativeReal*
- *ExpectationValueBraDerivativeImag*
- *MetricTensorReal*

Parameters

- **parameters** (`Union[SymbolDict, Dict[Symbol, float]]`) – Symbols and their values to be used in the evaluation.
- **allow_partial** (`bool`, default: `True`) – If `False`, evaluation will fail when an unsupported computable node is encountered. If `True`, unsupported nodes will be skipped.

Raises

`NotImplementedError` – If an unsupported computable is encountered, and `allow_partial==False`.

Returns

`Callable[[Evaluable], Union[Evaluable, Any]]` – A function that can evaluates quantum computables. If the `Computable` is supported by this protocol, it is computed; otherwise, it returns the `Computable` itself.

get_runner(qc)

Returns an end-to-end executor function for a quantum computable.

All nodes in the quantum computable must be evaluable by this protocol. Supported computables are:

- *ExpectationValue*
- *ExpectationValueNonHermitian*
- *ExpectationValueDerivative*
- *ExpectationValueBraDerivativeReal*
- *ExpectationValueBraDerivativeImag*
- *MetricTensorReal*

Parameters

`qc` (`ComputableNode`) – The quantum computable node to be measured and evaluated at every call of the returned function.

Returns

`Callable[[SymbolDict], Any]` – A function that takes the parameters and returns the evaluated result. If the result is not a `float`, it returns `math.nan`.

27.11.5 Protocols for Derivatives

```
class HadamardTestDerivativeOverlap(backend, shots_per_circuit=10000)
```

Bases: ProtocolListItem, PartiallyPickleable, ComputableCompliantMixin

Calculate the derivative overlap (“Metric tensor”) of a parametrized reference state using an ancilla qubit.

Computes the real or imaginary part of the quantity $\langle \partial_i \psi | \partial_j \psi \rangle$. See <https://journals.aps.org/prx/abstract/10.1103/PhysRevX.7.021050> and section III.B of <https://journals.aps.org/pra/abstract/10.1103/PhysRevA.99.032331>.

Input state $|\psi(\theta)\rangle$ is “regularized” so that all parameters are replaced with dummy parameters such that each gate in the state preparation circuit depends on a single dummy parameter, and each parameter appears only once: $|\psi(\theta')\rangle = \prod_i U_i(\theta'_i)|0\rangle$. Using the chain rule, the derivative overlap is then given by $\langle \partial_i \psi | \partial_j \psi \rangle = \sum_{kl} J_{ki} J_{lj} \langle \partial'_k \psi | \partial'_l \psi \rangle$ where $J_{ki} = \partial \theta'_k / \partial \theta_i$ is the Jacobian, and $\langle \partial'_k \psi | \partial'_l \psi \rangle$ is the regularized derivative overlap, for which we construct circuits to compute.

Note

Imaginary part is not yet supported.

Parameters

- **backend** (Backend) – The backend to use for quantum computations.
- **shots_per_circuit** (int, default: 10000) – Number of shots for each circuit.

```
TOLERANCE = 1e-08
```

```
build(parameters, state, diff_symbols=None, component=NumberType.REAL, optimisation_level=1)
```

Build measurement circuits for a derivative overlap, or matrix of derivative overlaps (“Metric tensor”).

Parameters

- **parameters** (Union[SymbolDict, Dict]) – Numeric values for ansatz parameters.
- **state** (GeneralAnsatz) – Parametrized input state $|\psi(\theta)\rangle$ with which to calculate derivative overlap.
- **diff_symbols** (Union[Tuple[Symbol, Symbol], List[Tuple[Symbol, Symbol]], None], default: None) – Symbol pairs with respect to which the bra and ket derivatives are computed. If None, the entire metric tensor is computed.
- **component** (Union[str, NumberType], default: <NumberType.REAL: 'real'>) – Component of derivative overlap to measure: 'real' or 'imag'.
- **optimisation_level** (int, default: 1) – Passed to the backend’s `get_compiled_circuits()` method.

Note

Imaginary part is not yet supported.

Returns

`HadamardTestDerivativeOverlap` – Self instance.

`build_from`(parameters, computable, exclude=None)

Build the protocol based on given parameters and computable expression.

This method walks over the computable expression tree and calls the `build()` method for leaves of type `MetricTensorReal`.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Parameters for building.
- **computable** (`ComputableNode`) – A root node of a computable expression tree, simply it is a computable expression.
- **exclude** (`Optional[Callable[[ComputableNode], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.

Returns

`HadamardTestDerivativeOverlap` – Returns the modified instance after building.

`classmethod build_protocols_from`(parameters, computable, exclude=None, optimisation_level=1, *args, **kwargs)

Build a list of protocols based on the given parameters and computable expression.

This method walks over the computable expression tree and collects leaf nodes of type:

- `MetricTensorReal`

After it has walked over the tree, it creates an instance of this protocol for each distinct state and calls the `build()` method. It collects all created protocols into a `ProtocolList` object to be returned.

Parameters

- **parameters** (`Union[dict, SymbolDict]`) – Values for the parameters in the computable expression.
- **computable** (`ComputableNode`) – A root node of a computable expression tree.
- **exclude** (`Optional[Callable[[ComputableNode], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.
- **optimisation_level** (`int`, default: 1) – Passed to the backend's `get_compiled_circuits()` method.
- ***args** – Arguments passed to the constructor of the overlap squared protocol.
- ****kwargs** – Keyword arguments passed to the constructor of the overlap squared protocol.

Returns

`ProtocolList` – A list, containing all the newly instantiated and built protocols.

`clear()`

Resets the internal state of the object by clearing all stored data.

This method clears all stored data including state hashes, operator hashes, measurement setups, parameters, and associated dataframes and circuits.

Returns

`HadamardTestDerivativeOverlap` – self.

`cost()`

Calculate a simple cost metric running the protocol.

The cost is calculated as the sum of the depth of each circuit multiplied by the number of shots.

Returns

`int` – The cost value as an integer.

credits (`syntax_checker=None`, `use_websocket=None`)

Evaluate an approximate cost for the measurement circuits built in Quantinuum credits.

Syntax checker will usually be automatically selected, but in some cases needs to be provided. If the measurement circuits are not built yet, the credits will be zero.

 **Note**

This works only with `QuantinuumBackend` compatible backends.

Parameters

- `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 total cost in Quantinuum credits to run the circuits.

dataframe_circuit_shot()

Create a pandas `DataFrame` with circuit, shot, and depth information.

Returns

`DataFrame` – A pandas `DataFrame` containing the circuit, shot, and depth information.

distributions: Optional[List[EmpiricalDistribution]]

dump (file)

Save the object to a file using pickle.

Parameters

`file` (`Union[str, BinaryIO]`) – The file path or file object to write the pickled data to.

Return type

`None`

dumps ()

Returns the object pickled as a bytes object.

Returns

`bytes` – The object's state pickled as a bytes object.

evaluate_derivative_overlap(state=None, diff_symbols=None)

Evaluates the derivative overlap, or matrix of derivative overlaps (“Metric tensor”).

If input args are `code=None`, computes the derivative overlap for which this protocol was built.

`state`: Quantum state for which the derivative overlap is to be calculated. `diff_symbols`: The set of symbols with respect to which the derivatives are evaluated. If `None`, all

elements of the metric tensor are evaluated.

Returns

`Dict[Tuple[Symbol, Symbol], float]` – Dict of metric tensor elements indexed by the symbols with respect to which the bra and ket derivatives are performed.

Parameters

- **state** (`Optional[GeneralAnsatz]`, default: `None`)
- **diff_symbols** (`Optional[Set[Symbol]]`, default: `None`)

`get_circuits()`

Returns the quantum circuits built for this protocol.

Returns

`List[Circuit]` – List of circuits.

`get_circuitshots()`

Generate the circuit shot pairs.

Yields

Pair of circuit and the associated number of shots.

Return type

`Iterator[CircuitShots]`

`get_dataframe_derivative_overlap()`

Return pandas dataframe showing computational details.

Shows all ingredients in the calculation of the derivative overlap. Each row is a term in (the real or imaginary part of): $\langle \partial_i \psi | \partial_j \psi \rangle = \sum_{kl} J_{ki} J_{lj} \langle \partial'_k \psi | \partial'_l \psi \rangle$.

Column headings are:

- symbols: `tuple` of symbols i and j.
- regularized symbols: `tuple` of symbols k and l.
- prefactor: Product of Jacobians $J_{ki} J_{lj}$. With a minus sign if necessary for the imaginary part.
- circuit index: Index of measurement circuit in `get_circuits()` which computes the real/imag part of the regularized derivative overlap $\langle \partial'_k \psi | \partial'_l \psi \rangle$. If `None`, the regularized derivative overlap is 1.

Returns

`DataFrame` – Dataframe detailing the calculation.

`get_evaluator(allow_partial=False)`

Returns an evaluator function to evaluate quantum computables.

This method creates and returns a function (evaluator) that takes in a specific quantum computable and evaluates it based on its type.

Quantum computables the returned evaluator can handle:

- `MetricTensorReal`

Parameters

`allow_partial` (`bool`, default: `False`) – If `False`, evaluation will fail when an unsupported computable node is encountered. If `True`, unsupported nodes will be skipped.

Raises

`NotImplementedError` – If an unsupported computable is encountered, and `allow_partial==False`.

Note

This evaluator works for quantum computables the protocol has been built from.

Returns

`Callable[[Evaluatable], Union[Evaluatable, Any]]` – A function that can evaluates quantum computables. If a computable is supported by this protocol, it is computed; otherwise, it returns the computable itself.

`get_runner(qc)`

Returns an end-to-end executor function for a quantum computable.

Note

All nodes in the quantum computable must be evaluable by this protocol. That is, the leaf nodes need to be `MetricTensorReal`.

Note

At every call of the returned function the internal state of the protocol changes.

Parameters

`qc (ComputableNode)` – The quantum computable node to be measured and evaluated at every call of the returned function.

Returns

`Callable[[Union[SymbolDict, Dict]], Any]` – A function that takes the parameters, builds and measures the necessary circuits, and returns the evaluated result. If the result is not a `float`, it returns `math.nan`.

`get_shots()`

Returns the number shots to be used for each circuit.

Returns

`List[int]` – List of number of shots, the length of this list is the same as the number of circuits.

`property is_built: bool`

Boolean flag indicating if the instance has been built.

Returns

`True` if the `build()` method has been successfully invoked, otherwise `False`.

`property is_numeric: bool`

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`False` if any built measurement circuit contains unsubstituted symbols, otherwise `True`.

`property is_run: bool`

Boolean flag indicating if the instance has been run.

Returns

`True` if the `run()` method has been successfully invoked, otherwise `False`.

```
property is_symbolic: bool
```

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`True` if any built measurement circuit contains unsubstituted symbols, otherwise `False`.

```
launch(*args, **kwargs)
```

Launch the circuits to the backend and return the handles for the results.

This method processes all the circuits and returns a list of `ResultHandle` objects representing the handles for the results.

Parameters

- `*args` – Additional arguments to be passed to `self.backend.process_circuits()`.
- `**kwargs` – Additional keyword arguments to be passed to `self.backend.process_circuits()`.

Returns

`List[ResultHandle]` – A list of pytket `ResultHandle` objects representing the handles for the launched circuits.

```
classmethod load(file, *args, **kwargs)
```

Load a pickled object from a file.

Parameters

- `file (Union[str, BinaryIO])` – The file path or file object to load the pickled data from.
- `*args (Any)` – Additional arguments passed to the class constructor.
- `**kwargs (Any)` – Additional keyword arguments passed to the class constructor.

Returns

`TypeVar(T, bound= PartiallyPickleable)` – An instance of the class with its state loaded from the pickled data.

```
classmethod loads(pickled_data, *args, **kwargs)
```

Load a pickled object from a bytes object.

Parameters

- `pickled_data (bytes)` – The pickled data to load the object from.
- `*args (Any)` – Additional arguments passed to the class constructor.
- `**kwargs (Any)` – Additional keyword arguments passed to the class constructor.

Returns

`TypeVar(T, bound= PartiallyPickleable)` – An instance of the class with its state loaded from the pickled data.

```
property n_circuit: int
```

Returns the total number of circuits.

```
parameters: Optional[SymbolDict]
```

```
rebuild(*args, **kwargs)
```

Rebuild the internal data structure.

It is equivalent to `clear().build(*args, **kwargs)`.

Parameters

- ***args** ([Any](#)) – Arguments to be passed to `build()`.
- ****kwargs** ([Any](#)) – Keyword arguments to be passed to `build()`.

Returns`TypeVar(TBuildClearMixin, bound= BuildClearMixin) – self.`**retrieve** (*source*, **args*, ***kwargs*)

Retrieve distributions from the backend for the given source.

If the *source* is a list of pytket ResultHandle, the distributions are retrieved using `self.backend.get_results()` method. If the *source* is a list of pytket BackendResult, it is assumed that the results are already provided.

Parameters

- **source** ([Union\[List\[ResultHandle\], List\[BackendResult\]\]](#)) – A list of pytket ResultHandle or BackendResult objects representing the source of the distributions.
- ****kwargs** – Additional keyword arguments to be passed to `self.backend.get_results()` when retrieving results.

Returns`HadamardTestDerivativeOverlap` – Returns self instance.**run** (**args*, ***kwargs*)

Run the protocol and waiting for the results.

This method executes the following steps: 1. Launches the measurement circuits to obtain the circuit handles.
2. Retrieves the measurement distributions using the circuit handles.

Parameters

- ***args** ([Any](#)) – Arguments to be passed to `launch()`.
- ****kwargs** ([Any](#)) – Keyword arguments to be passed to `launch()`.

Returns`TypeVar(TLaunchRetrieveMixin, bound= LaunchRetrieveMixin) – self.`**class HadamardTestDerivative** (*backend*, *shots_per_circuit*=8000, *direct*=True)

Bases: `ProtocolListItem`, `ComputableCompliantMixin`

Calculates a partial derivative of an expectation value using Hadamard test.

Implements the procedure and it's variant described in [arXiv:1701.01450](#).

If *direct*=True, both ancilla and all other qubits are measured (original method). If it is instead set to False, an alternative circuit is built, and only ancilla qubit is measured.

Parameters

- **backend** (`Backend`) – The backend to use for quantum computations.
- **shots_per_circuit** (`int`, default: 8000) – Number of shots for each circuit.
- **direct** (`bool`, default: True) – Switching between direct- or indirect-type circuits.

build (*parameters*, *state*, *operator*, *symbols*, *complex_part*, *optimisation_level*=1)

Build the necessary circuits and measurement data for evaluating derivatives.

Parameters

- **parameters** ([Union\[SymbolDict, Dict\]](#)) – A dictionary or `SymbolDict` containing the parameter values for the circuits.

- **state** (`GeneralAnsatz`) – Parametrized input state with which to calculate derivative.
- **operator** (`QubitOperator`) – Operator kernel.
- **symbols** (`Set[Symbol]`) – Symbols with respect to which the derivatives are computed. If `None`, derivatives are computed with respect to all symbols.
- **complex_part** (`Union[ComplexPart, str]`) – Component of derivative to measure, real or imaginary. Accepts strings '`real`' and '`imag`'.
- **optimisation_level** (`int`, default: `1`) – Passed to the backend's `get_compiled_circuits()` method.

Returns

`HadamardTestDerivative` – Self instance.

build_from(*parameters*, *computable*, *exclude*=`None`)

Build the protocol based on given parameters and computable expression.

This method walks over the computable expression tree and calls build for leaves of type:

- `ExpectationValueDerivative`
- `ExpectationValueBraDerivativeReal`
- `ExpectationValueBraDerivativeImag`
- `ExpectationValueKetDerivativeReal`
- `ExpectationValueKetDerivativeImag`

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Parameters for building.
- **computable** (`ComputableNode`) – A root node of a computable expression tree, simply it is a computable expression.
- **exclude** (`Optional[Callable[[ComputableNode], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.

Returns

`HadamardTestDerivative` – Returns the modified instance after building.

classmethod build_protocols_from(*parameters*, *computable*, *exclude*=`None`, *optimisation_level*=`1`, **args*, ***kwargs*)

Build a list of protocols based on the given parameters and computable expression.

This method walks over the computable expression tree and collects leaf nodes of type:

- `ExpectationValueDerivative`
- `ExpectationValueBraDerivativeReal`
- `ExpectationValueBraDerivativeImag`
- `ExpectationValueKetDerivativeReal`
- `ExpectationValueKetDerivativeImag`

After it has walked over the tree, it creates an instance of this protocol for each distinct state, component pair and calls the `build()` method. It collects all created protocols into a `ProtocolList` object to be returned.

Parameters

- **parameters** (`Union[dict, SymbolDict]`) – Values for the parameters in the computable expression.
- **computable** (`ComputableNode`) – A root node of a computable expression tree.
- **exclude** (`Optional[Callable[[ComputableNode], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.
- **optimisation_level** (`int`, default: 1) – Passed to the backend's `get_compiled_circuits()` method.
- ***args** – Arguments passed to the constructor of the overlap squared protocol.
- ****kwargs** – Keyword arguments passed to the constructor of the overlap squared protocol.

Returns

`ProtocolList` – A list, containing all the newly instantiated and built protocols.

clear()

Resets the internal state of the object by clearing all stored data.

This method clears all stored data including state hashes, operator hashes, measurement setups, parameters, and associated dataframes and circuits.

Returns

`HadamardTestDerivative` – `self`.

cost()

Calculate a simple cost metric running the protocol.

The cost is calculated as the sum of the depth of each circuit multiplied by the number of shots.

Returns

`int` – The cost value as an integer.

credits (`syntax_checker=None, use_websocket=None`)

Evaluate an approximate cost for the measurement circuits built in Quantinuum credits.

Syntax checker will usually be automatically selected, but in some cases needs to be provided. If the measurement circuits are not built yet, the credits will be zero.

Note

This works only with QuantinuumBackend compatible backends.

Parameters

- **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 total cost in Quantinuum credits to run the circuits.

dataframe_circuit_shot()

Create a pandas `DataFrame` with circuit, shot, and depth information.

Returns

`DataFrame` – A pandas `DataFrame` containing the circuit, shot, and depth information.

`evaluate_dbra`(*state, kernel, symbols, complex_part*)

Evaluates partial bra derivatives of expectation value of a Hermitian operator.

This method can only be performed if, prior to calling this method, the protocol has been built for the state, kernel (or other operators composed of the same Pauli strings), symbols and the corresponding complex part and the protocol has also been run.

Parameters

- **state** (*GeneralAnsatz*) – The quantum state for which the derivative of the expectation value of the kernel is to be calculated.
- **kernel** (*QubitOperator*) – The Hermitian operator for which the derivative of the expectation value is being computed.
- **symbols** (*Optional[Set[Symbol]]*) – The set of symbols with respect to which the derivative will be evaluated. If *None*, the derivative will be evaluated with respect to all symbols found in the expressions.
- **complex_part** (*Union[ComplexPart, str]*) – Component of the dbra expression to evaluate, real or imaginary, assuming the kernel is Hermitian. Accepts strings '*real*' and '*imag*'.

Returns

Dict[Symbol, float] – A dictionary mapping each symbol to its corresponding derivative.

`evaluate_dket`(*state, kernel, symbols, complex_part*)

Evaluates partial ket derivatives of expectation value of a Hermitian operator.

If no symbols are provided, it will evaluate the derivative with respect to all symbols found in the state.

This method can only be performed if, prior to calling this method, the protocol has been built for the state, kernel (or other operators composed of the same Pauli strings), symbols and the corresponding complex part and the protocol has also been run.

Parameters

- **state** (*GeneralAnsatz*) – The quantum state for which the derivative of the expectation value of the kernel is to be calculated.
- **kernel** (*QubitOperator*) – The Hermitian operator for which the derivative of the expectation value is being computed.
- **symbols** (*Optional[Set[Symbol]]*) – The set of symbols with respect to which the derivative will be evaluated. If *None*, the derivative will be evaluated with respect to all symbols found in the expressions.
- **complex_part** (*Union[ComplexPart, str]*) – Component of the dket expression to evaluate, real or imaginary, assuming the kernel is Hermitian. Accepts strings '*real*' and '*imag*'.

Returns

Dict[Symbol, float] – A dictionary mapping each symbol to its corresponding derivative.

`evaluate_gradient`(*state, kernel, symbols=None*)

Evaluates gradient of expectation value of a Hermitian operator.

Since the kernel is Hermitian, the gradient is calculated as $2 * \text{Re}(\langle d\Psi|H|\Psi\rangle)$.

This method can only be performed if, prior to calling this method, the protocol has been built for the state, kernel (or other operators composed of the same Pauli strings), symbols and *ComplexPart.REAL* and the protocol has also been run.

Parameters

- **state** (*GeneralAnsatz*) – The quantum state for which the derivative of the expectation value of the kernel is to be calculated.
- **kernel** (*QubitOperator*) – The Hermitian operator for which the derivative of the expectation value is being computed.
- **symbols** (*Optional[Set[Symbol]]*, default: `None`) – The set of symbols with respect to which the derivative will be evaluated. If `None`, the derivative will be evaluated with respect to all symbols found in the expressions.

Returns

`Dict[Symbol, float]` – A dictionary mapping each symbol to its corresponding derivative.

`get_circuits()`

Returns the quantum circuits built for this protocol.

Returns

`List[Circuit]` – List of circuits.

`get_circuitshots()`

Generate the circuit shot pairs.

Yields

Pair of circuit and the associated number of shots.

Return type

`Iterator[CircuitShots]`

`get_evaluator(allow_partial=False)`

Returns an evaluator function to evaluate quantum computables.

This method creates and returns a function (evaluator) that takes in a specific quantum computable and evaluates it based on its type.

Quantum computables the returned evaluator can handle:

- *ExpectationValueDerivative*
- *ExpectationValueBraDerivativeReal*
- *ExpectationValueBraDerivativeImag*
- *ExpectationValueKetDerivativeReal*
- *ExpectationValueKetDerivativeImag*

Parameters

`allow_partial` (`bool`, default: `False`) – If `False`, evaluation will fail when an unsupported computable node is encountered. If `True`, unsupported nodes will be skipped.

Note

This evaluator works for quantum computables the protocol has been built from.

Raises

`NotImplementedError` – If an unsupported computable is encountered, and `allow_partial==False`.

Returns

`Callable[[Evaluatable], Union[Evaluatable, Any]]` – A function that can evaluate quantum computables. If a computable is supported by this protocol, it is computed; otherwise, it returns the computable itself.

get_runner (`qc, compile_symbolic=False, *args, **kwargs`)

Returns an end-to-end executor function for a quantum computable.

All nodes in the quantum computable must be evaluable by this protocol. That is, the leaf nodes need to be of one of the supported types:

- `ExpectationValueDerivative`
- `ExpectationValueBraDerivativeReal`
- `ExpectationValueBraDerivativeImag`
- `ExpectationValueKetDerivativeReal`
- `ExpectationValueKetDerivativeImag`

 **Note**

At every call of the returned function the internal state of the protocol changes.

Parameters

- `qc` (`ComputableNode`) – The quantum computable node to be measured and evaluated at every call of the returned function.
- `compile_symbolic` (`bool`, default: `False`) – If `True`, circuits are immediately built and compiled symbolically. Calling the returned function substitutes symbols, runs circuits, and evaluates the output. If `False`, circuits are built and compiled only when the returned function is called and parameters are provided. Note that circuits may be deeper when `compile_symbolic=True`, because circuit compilation cannot perform optimizations based on numerical parameter values. The advantage of this case is that compilation only takes place once.
- `args` (`Any`)
- `kwargs` (`Any`)

Returns

`Callable[[Union[SymbolDict, Dict]], float]` – A function that takes the parameters, builds and measures the necessary circuits, and returns the evaluated result. If the result is not a `float`, it returns `math.nan`.

get_shots()

Returns the number shots to be used for each circuit.

Returns

`List[int]` – List of number of shots, the length of this list is the same as the number of circuits.

property is_built: bool

Boolean flag indicating if the instance has been built.

Returns

`True` if the `build()` method has been successfully invoked, otherwise `False`.

property is_numeric: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`False` if any built measurement circuit contains unsubstituted symbols, otherwise `True`.

property is_run: bool

Boolean flag indicating if the instance has been run.

Returns

`True` if the `run()` method has been successfully invoked, otherwise `False`.

property is_symbolic: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`True` if any built measurement circuit contains unsubstituted symbols, otherwise `False`.

launch(*args, **kwargs)

Launch the circuits to the backend and return the handles for the results.

This method processes all the circuits and returns a list of `ResultHandle` objects representing the handles for the results.

Parameters

- ***args** – Additional arguments to be passed to `self.backend.process_circuits()`.
- ****kwargs** – Additional keyword arguments to be passed to `self.backend.process_circuits()`.

Returns

`List[ResultHandle]` – A list of pytket `ResultHandle` objects representing the handles for the launched circuits.

property n_circuit: int

Returns the total number of circuits.

rebuild(*args, **kwargs)

Rebuild the internal data structure.

It is equivalent to `clear().build(*args, **kwargs)`.

Parameters

- ***args (Any)** – Arguments to be passed to `build()`.
- ****kwargs (Any)** – Keyword arguments to be passed to `build()`.

Returns

`TypeVar(TBuildClearMixin, bound=BuildClearMixin) – self.`

retrieve(source, *args, **kwargs)

Retrieve distributions from the backend for the given source.

If the `source` is a list of pytket `ResultHandle`, the distributions are retrieved using `self.backend.get_results()` method. If the `source` is a list of pytket `BackendResult`, it is assumed that the results are already provided.

Parameters

- **source (Union[List[ResultHandle], List[BackendResult]])** – A list of pytket `ResultHandle` or `BackendResult` objects representing the source of the distributions.

- ****kwargs** – Additional keyword arguments to be passed to `self.backend.get_results()` when retrieving results.

Returns

`HadamardTestDerivative` – Returns self instance.

run (*args, **kwargs)

Run the protocol and waiting for the results.

This method executes the following steps: 1. Launches the measurement circuits to obtain the circuit handles. 2. Retrieves the measurement distributions using the circuit handles.

Parameters

- ***args** (`Any`) – Arguments to be passed to `launch()`.
- ****kwargs** (`Any`) – Keyword arguments to be passed to `launch()`.

Returns

`TypeVar(TLaunchRetrieveMixin, bound=LaunchRetrieveMixin)` – self.

class PhaseShift (backend, shots_per_circuit=8000, pauli_partition_strategy=PauliPartitionStrat.CommutingSets)

Bases: `ProtocolListItem`, `ComputableCompliantMixin`

Calculates total gradient of an expectation value using gate parameter shift.

Implements the parameter-shift rule described in [arXiv:1811.11184](https://arxiv.org/abs/1811.11184).

Parameters

- **backend** (`Backend`) – The backend to use for quantum computations.
- **shots_per_circuit** (`int`, default: 8000) – Number of shots for each circuit.
- **pauli_partition_strategy** (`PauliPartitionStrat`, default: `PauliPartitionStrat.CommutingSets`) – Strategy to partition Pauli operators. Default is `CommutingSets`.

build (parameters, state, kernel, symbols=None, optimisation_level=1)

Build the necessary circuits and measurement data for evaluating derivatives.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – A dictionary or `SymbolDict` containing the parameter values for the circuits.
- **state** (`GeneralAnsatz`) – Parametrized input state with which to calculate derivative.
- **kernel** (`QubitOperator`) – Operator kernel.
- **symbols** (`Optional[Set[Symbol]]`, default: `None`) – Symbols with respect to which the derivatives are computed. If `None`, derivatives are computed with respect to all symbols.
- **optimisation_level** (`int`, default: 1) – Passed to the backend's `get_compiled_circuits()` method.

Returns

`PhaseShift` – Self instance.

build_from (parameters, computable, exclude=None)

Build the protocol based on given parameters and computable expression.

This method walks over the computable expression tree and calls build for leaves of type:

- `ExpectationValueDerivative`

- *ExpectationValueBraDerivativeReal*
- *ExpectationValueKetDerivativeReal*

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – Parameters for building.
- **computable** (`ComputableNode`) – A root node of a computable expression tree, simply it is a computable expression.
- **exclude** (`Optional[Callable[[ComputableNode], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.

Returns

`PhaseShift` – Returns the modified instance after building.

```
classmethod build_protocols_from(parameters, computable, exclude=None, optimisation_level=1,
                                  *args, **kwargs)
```

Build a list of protocols based on the given parameters and computable expression.

This method walks over the computable expression tree and collects leaf nodes of type:

- *ExpectationValueDerivative*
- *ExpectationValueBraDerivativeReal*
- *ExpectationValueKetDerivativeReal*

After it has walked over the tree, it creates an instance of this protocol for each distinct state and calls the `build()` method. It collects all created protocols into a `ProtocolList` object to be returned.

Parameters

- **parameters** (`Union[dict, SymbolDict]`) – Values for the parameters in the computable expression.
- **computable** (`ComputableNode`) – A root node of a computable expression tree.
- **exclude** (`Optional[Callable[[ComputableNode], bool]]`, default: `None`) – Optional callable function to exclude certain nodes from processing.
- **optimisation_level** (`int`, default: 1) – Passed to the backend's `get_compiled_circuits()` method.
- ***args** – Arguments passed to the constructor of the overlap squared protocol.
- ****kwargs** – Keyword arguments passed to the constructor of the overlap squared protocol.

Returns

`ProtocolList` – A list, containing all the newly instantiated and built protocols.

```
clear()
```

Resets the internal state of the object by clearing all stored data.

This method clears all stored data including state hashes, operator hashes, measurement setups, parameters, and associated dataframes and circuits.

Returns

`PhaseShift` – self.

```
cost()
```

Calculate a simple cost metric running the protocol.

The cost is calculated as the sum of the depth of each circuit multiplied by the number of shots.

Returns

`int` – The cost value as an integer.

credits (`syntax_checker=None`, `use_websocket=None`)

Evaluate an approximate cost for the measurement circuits built in Quantinuum credits.

Syntax checker will usually be automatically selected, but in some cases needs to be provided. If the measurement circuits are not built yet, the credits will be zero.

 **Note**

This works only with `QuantinuumBackend` compatible backends.

Parameters

- **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 total cost in Quantinuum credits to run the circuits.

dataframe_circuit_shot()

Create a pandas `DataFrame` with circuit, shot, and depth information.

Returns

`DataFrame` – A pandas `DataFrame` containing the circuit, shot, and depth information.

evaluate_dbra (`state`, `kernel`, `symbols`)

Evaluates partial bra derivatives of expectation value of a Hermitian operator.

This method can only be performed if, prior to calling this method, the protocol has been built for the state, kernel (or other operators composed of the same Pauli strings) and symbols and the protocol has also been run.

Parameters

- **state** (`GeneralAnsatz`) – The quantum state for which the derivative of the expectation value of the kernel is to be calculated.
- **kernel** (`QubitOperator`) – The Hermitian operator for which the derivative of the expectation value is being computed.
- **symbols** (`Optional[Set[Symbol]]`) – The set of symbols with respect to which the derivative will be evaluated. If `None`, the derivative will be evaluated with respect to all symbols found in the expressions.

Returns

`Dict[Symbol, float]` – A dictionary mapping each symbol to its corresponding derivative.

evaluate_dket (`state`, `kernel`, `symbols`)

Evaluates partial ket derivatives of expectation value of a Hermitian operator.

This method can only be performed if, prior to calling this method, the protocol has been built for the state, kernel (or other operators composed of the same Pauli strings) and symbols and the protocol has also been run.

Parameters

- **state** (*GeneralAnsatz*) – The quantum state for which the derivative of the expectation value of the kernel is to be calculated.
- **kernel** (*QubitOperator*) – The Hermitian operator for which the derivative of the expectation value is being computed.
- **symbols** (*Optional[Set[Symbol]]*) – The set of symbols with respect to which the derivative will be evaluated. If *None*, the derivative will be evaluated with respect to all symbols found in the expressions.

Returns`Dict[Symbol, float]` – A dictionary mapping each symbol to its corresponding derivative.**`evaluate_gradient(state, kernel, symbols=None)`**

Evaluates gradient of the expectation value of a Hermitian operator.

This method can only be performed if, prior to calling this method, the protocol has been built for the state, kernel (or other operators composed of the same Pauli strings) and symbols and the protocol has also been run.

Parameters

- **state** (*GeneralAnsatz*) – The quantum state for which the derivative of the expectation value of the kernel is to be calculated.
- **kernel** (*QubitOperator*) – The Hermitian operator for which the derivative of the expectation value is being computed.
- **symbols** (*Optional[Set[Symbol]]*, default: *None*) – The set of symbols with respect to which the derivative will be evaluated. If *None*, the derivative will be evaluated with respect to all symbols found in the expressions.

Returns`Dict[Symbol, float]` – A dictionary mapping each symbol to its corresponding derivative.**`get_circuits()`**

Returns the quantum circuits built for this protocol.

Returns`List[Circuit]` – List of circuits.**`get_circuitshots()`**

Generate the circuit shot pairs.

Yields

Pair of circuit and the associated number of shots.

Return type`Iterator[CircuitShots]`**`get_evaluator(allow_partial=False)`**

Returns an evaluator function to evaluate quantum computables.

This method creates and returns a function (evaluator) that takes in a specific quantum computable and evaluates it based on its type.

Quantum computables the returned evaluator can handle:

- *ExpectationValueDerivative*
- *ExpectationValueBraDerivativeReal*
- *ExpectationValueKetDerivativeReal*

Parameters

`allow_partial` (`bool`, default: `False`) – If `False`, evaluation will fail when an unsupported computable node is encountered. If `True`, unsupported nodes will be skipped.

Raises

`NotImplementedError` – If an unsupported computable is encountered, and `allow_partial==False`.

Note

This evaluator works for quantum computables the protocol has been built from.

Returns

`Callable[[Evaluable], Union[Evaluable, Any]]` – A function that can evaluate quantum computables. If a computable is supported by this protocol, it is computed; otherwise, it returns the computable itself.

`get_runner` (`qc`, `compile_symbolic=False`, `*args`, `**kwargs`)

Returns an end-to-end executor function for a quantum computable.

All nodes in the quantum computable must be evaluable by this protocol. That is, the leaf nodes need to be of one of the supported types:

- `ExpectationValueDerivative`
- `ExpectationValueBraDerivativeReal`
- `ExpectationValueKetDerivativeReal`

Note

At every call of the returned function the internal state of the protocol changes.

Parameters

- `qc` (`ComputableNode`) – The quantum computable node to be measured and evaluated at every call of the returned function.
- `compile_symbolic` (`bool`, default: `False`) – If `True`, circuits are immediately built and compiled symbolically. Calling the returned function substitutes symbols, runs circuits, and evaluates the output. If `False`, circuits are built and compiled only when the returned function is called and parameters are provided. Note that circuits may be deeper when `compile_symbolic=True`, because circuit compilation cannot perform optimizations based on numerical parameter values. The advantage of this case is that compilation only takes place once.
- `args` (`Any`)
- `kwargs` (`Any`)

Returns

`Callable[[Union[SymbolDict, Dict]], float]` – A function that takes the parameters, builds and measures the necessary circuits, and returns the evaluated result. If the result is not a `float`, it returns `math.nan`.

get_shots()

Returns the number shots to be used for each circuit.

Returns

`List[int]` – List of number of shots, the length of this list is the same as the number of circuits.

property is_built: bool

Boolean flag indicating if the instance has been built.

Returns

`True` if the `build()` method has been successfully invoked, otherwise `False`.

property is_numeric: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`False` if any built measurement circuit contains unsubstituted symbols, otherwise `True`.

property is_run: bool

Boolean flag indicating if the instance has been run.

Returns

`True` if the `run()` method has been successfully invoked, otherwise `False`.

property is_symbolic: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`True` if any built measurement circuit contains unsubstituted symbols, otherwise `False`.

launch(*args, **kwargs)

Launch the circuits to the backend and return the handles for the results.

This method processes all the circuits and returns a list of `ResultHandle` objects representing the handles for the results.

Parameters

- `*args (Any)` – Additional arguments to be passed to `self.backend.process_circuits()`.
- `**kwargs (Any)` – Additional keyword arguments to be passed to `self.backend.process_circuits()`.

Returns

`List[ResultHandle]` – A list of pytket `ResultHandle` objects representing the handles for the launched circuits.

property n_circuit: int

Returns the total number of circuits.

rebuild(*args, **kwargs)

Rebuild the internal data structure.

It is equivalent to `clear().build(*args, **kwargs)`.

Parameters

- `*args (Any)` – Arguments to be passed to `build()`.
- `**kwargs (Any)` – Keyword arguments to be passed to `build()`.

Returns

`TypeVar(TBuildClearMixin, bound= BuildClearMixin) – self.`

retrieve (source, **kwargs)

Retrieve distributions from the backend for the given source.

If the `source` is a list of pytket ResultHandle, the distributions are retrieved using `self.backend.get_results()` method. If the `source` is a list of pytket BackendResult, it is assumed that the results are already provided.

Parameters

- `source (Union[List[ResultHandle], List[BackendResult]])` – A list of pytket ResultHandle or BackendResult objects representing the source of the distributions.
- `**kwargs (Any)` – Additional keyword arguments to be passed to `self.backend.get_results()` when retrieving results.

Returns

`PhaseShift` – Returns self instance.

run (*args, **kwargs)

Run the protocol and waiting for the results.

This method executes the following steps: 1. Launches the measurement circuits to obtain the circuit handles. 2. Retrieves the measurement distributions using the circuit handles.

Parameters

- `*args (Any)` – Arguments to be passed to `launch ()`.
- `**kwargs (Any)` – Keyword arguments to be passed to `launch ()`.

Returns

`TypeVar(TLaunchRetrieveMixin, bound= LaunchRetrieveMixin) – self.`

27.11.6 Protocols for Phase-Estimation

```
class CanonicalPhaseEstimation(backend, n_rounds, n_shots=10, compiler_passes=None, seed=None,
                               optimisation_level=2)
```

Bases: DeterministicQPECircuitProtocol

Protocol for the generation of canonical Quantum Phase Estimation circuits.

Parameters

- `backend (Backend)` – pytket Backend object.
- `n_rounds (int)` – Number of qubits in the readout register.
- `n_shots (int, default: 10)` – Number of shots.
- `compiler_passes (Optional[BasePass], default: None)` – Compiler passes to be used.
- `seed (Optional[int], default: None)` – Random seed.
- `optimisation_level (int, default: 2)` – Level of circuit optimization.

copy ()

Returns a deep copy of the protocol.

Return type

Protocol

```
class IterativePhaseEstimationSingleCircuit (backend, n_rounds, n_shots=10, compiler_passes=None,
                                             seed=None, optimisation_level=2)
```

Bases: DeterministicQPECircuitProtocol

Protocol for the generation of “one-circuit” iterative Quantum Phase Estimation circuits.

“One-circuit” here refers to the method of circuit generation, wherein the entire iterative phase estimation procedure is returned as a single circuit with extensive classical control. The successive rounds of phase estimation are concatenated into a single circuit, with the “correction” Z rotation generated by 2^N rotation gates, each classically controlled on the outcome of the preceding rounds of phase estimation.

Parameters

- **backend** (Backend) – pytket Backend object.
- **n_rounds** (int) – Number of qubits in the readout register.
- **n_shots** (int, default: 10) – Number of shots.
- **compiler_passes** (Optional[BasePass], default: None) – Compiler passes to be used.
- **seed** (Optional[int], default: None) – Random seed.
- **optimisation_level** (int, default: 2) – Level of circuit optimization.

copy()

Returns a deep copy of the protocol.

Return type

Protocol

```
class MeasurementPluralityPhaseEstimator
```

Bases: ShotBasedPhaseEstimator

Converts raw experimental results from a shot-based simulation to a phase, by taking the most frequent outcome.

```
class LinearInterpolatorPhaseEstimator
```

Bases: ShotBasedPhaseEstimator

Converts raw experimental results from a shot-based simulation to a phase, by generating a probability distribution function over the results.

The probability distribution function will be generated through periodic linear interpolation on the experimental results. Protocol execution will return an expected value of the phase by querying the PDF at every possible measurement outcome, and selecting the outcome with the highest probability.

```
class IterativePhaseEstimation (backend, n_shots=1, compilation_level=CompilationLevel.COMPILED,
                                 optimisation_level=1)
```

Bases: BaseIterativePhaseEstimationCircuit

Perform iterative QPE with a circuit with k and β to obtain the measurement outcome m .

Parameters

- **backend** (Backend) – pytket backend.
- **n_shots** (int, default: 1) – Number of shots (default: 1).
- **compilation_level** (CompilationLevel, default: CompilationLevel.COMPILED) – Compilation level.
- **optimisation_level** (int, default: 1) – Optimization level.

```
property backend: Backend | None
    Backend object to be internally used.

property beta_iqpe: float
    Circuit parameter  $\beta$ .

build(state, evolution_operator_exponents, eoe_totally_commuting=None, ctrlu_strat=None, passes=None)
    Build a function to generate Iterative QPE circuit using the Lie-Trotter product formula.
```

Parameters

- **state** (*GeneralAnsatz*) – Ansatz for the state preparation (non-symbolic).
- **evolution_operator_exponents** (*QubitOperatorList*) – List of Pauli strings to be trotterized.
- **eoe_totally_commuting** (*Optional[QubitOperatorList]*, default: `None`) – Totally commuting set of Pauli strings to be added to the end of CTRL-U.
- **ctrlu_strat** (*Optional[CtrluStrat]*, default: `None`) – CTRL-U compilation strategy.
- **passes** (*Optional[BasePass]*, default: `None`) – Compiler pass to be applied for Ctrl-U circuit.

Returns`BaseIterativePhaseEstimation – self`

```
build_from_circuit(state, get_ctrlu)
    Build the protocol using the lower-level inputs (circuits).
```

Parameters

- **state** (*Circuit*) – State preparation circuit.
- **get_ctrlu** (*Callable[[int], Circuit]*) – Function to return a controlled unitary circuit.

```
clear()
```

Resets the internal state of the object by clearing all stored data.

This method clears all stored data including state hashes, operator hashes, measurement setups, parameters, and associated dataframes and circuits.

Returns`BaseIterativePhaseEstimationCircuit – self.`

```
clear_cache()
```

Clear the circuit cache.

```
property compilation_level: CompilationLevel
```

Compilation level.

```
cost()
```

Calculate a simple cost metric running the protocol.

The cost is calculated as the sum of the depth of each circuit multiplied by the number of shots.

Returns`int – The cost value as an integer.`

credits (*syntax_checker=None, use_websocket=None*)

Evaluate an approximate cost for the measurement circuits built in Quantinuum credits.

Syntax checker will usually be automatically selected, but in some cases needs to be provided. If the measurement circuits are not built yet, the credits will be zero.

 **Note**

This works only with QuantinuumBackend compatible backends.

Parameters

- **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 total cost in Quantinuum credits to run the circuits.

dataframe_circuit_shot()

Create a pandas DataFrame with circuit, shot, and depth information.

Returns

`DataFrame` – A pandas DataFrame containing the circuit, shot, and depth information.

get_circuits()

Returns the quantum circuits built for this protocol.

Returns

`List[Circuit]` – List of circuits.

get_circuitshots()

Generate the circuit shot pairs.

Yields

Pair of circuit and the associated number of shots.

Return type

`Iterator[CircuitShots]`

get_distribuition()

Get the distribution based on the measurement outcome.

Returns

`Mapping[Tuple[int, ...], float]` – Distribution.

property get_iqpe_circuit: Callable[[int, float], Circuit]

Interface to the function to generate the IQPE circuit.

Returns

function to build an IQPE circuit.

get_measurement_outcome()

Get the measurement outcome as a bit string.

Returns

`list[int]` – List of int in {0, 1} to be used by iterative QPE algorithms.

get_shots()

Returns the number shots to be used for each circuit.

Returns

`List[int]` – List of number of shots, the length of this list is the same as the number of circuits.

property is_built: bool

Boolean flag indicating if the instance has been built.

Returns

`True` if the `build()` method has been successfully invoked, otherwise `False`.

property is_numeric: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`False` if any built measurement circuit contains unsubstituted symbols, otherwise `True`.

property is_run: bool

Boolean flag indicating if the instance has been run.

Returns

`True` if the `run()` method has been successfully invoked, otherwise `False`.

property is_symbolic: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`True` if any built measurement circuit contains unsubstituted symbols, otherwise `False`.

property k_iqpe: int

Circuit parameter k .

launch(kwargs)**

Launch the circuits to the backend and return the handles for the results.

This method processes all the circuits and returns a list of `ResultHandle` objects representing the handles for the results.

Parameters

- `*args` – Additional arguments to be passed to `self.backend.process_circuits()`.
- `**kwargs` (`Mapping[str, Any]`) – Additional keyword arguments to be passed to `self.backend.process_circuits()`.

Returns

`List[ResultHandle]` – A list of pytket `ResultHandle` objects representing the handles for the launched circuits.

property n_circuit: int

Returns the total number of circuits.

property n_shots: int

Number of shots.

property optimisation_level: int

Optimization level.

rebuild(*args, **kwargs)

Rebuild the internal data structure.

It is equivalent to `clear().build(*args, **kwargs)`.

Parameters

- ***args** (`Any`) – Arguments to be passed to `build()`.
- ****kwargs** (`Any`) – Keyword arguments to be passed to `build()`.

Returns

`TypeVar(TBuildClearMixin, bound= BuildClearMixin) – self.`

retrieve(source, **kwargs)

Retrieve distributions from the backend for the given source.

If the `source` is a list of pytket ResultHandle, the distributions are retrieved using `self.backend.get_results()` method. If the `source` is a list of pytket BackendResult, it is assumed that the results are already provided.

Parameters

- **source** (`Union[List[ResultHandle], List[BackendResult]]`) – A list of pytket ResultHandle or BackendResult objects representing the source of the distributions.
- ****kwargs** (`Mapping[str, Any]`) – Additional keyword arguments to be passed to `self.backend.get_results()` when retrieving results.

Returns

`BaseIterativePhaseEstimation – Returns self instance.`

run(*args, **kwargs)

Run the protocol and waiting for the results.

This method executes the following steps: 1. Launches the measurement circuits to obtain the circuit handles.
2. Retrieves the measurement distributions using the circuit handles.

Parameters

- ***args** (`Any`) – Arguments to be passed to `launch()`.
- ****kwargs** (`Any`) – Keyword arguments to be passed to `launch()`.

Returns

`TypeVar(TLaunchRetrieveMixin, bound= LaunchRetrieveMixin) – self.`

update_k_and_beta(k, beta)

Update the circuit parameters k and beta.

Parameters

- **k** (`int`) – The number of repeats of the controlled unitary.
- **beta** (`float`) – Pre-measurement rotation angle of ancilla.

Return type

`BaseIterativePhaseEstimationCircuit`

```
class IterativePhaseEstimationQuantinuum(backend, n_shots=1,
                                         compilation_level=CompilationLevelQuantinuum.COMPILED,
                                         optimisation_level=1)
```

Bases: `BaseIterativePhaseEstimationCircuit`

General interface to the iterative QPE circuit with k and β to obtain the measurement outcome m .

Parameters

- **backend** (`Backend`) – pytket `backend`.
- **n_shots** (`Union[int, Callable[[int], int]]`, default: 1) – Number of shots (default: 1).
- **compilation_level** (`CompilationLevelQuantinum`,
`CompilationLevelQuantinum.COMPILED`) – Compilation level.
default:
- **optimisation_level** (`int`, default: 1) – Optimization level.

Notes

If `n_shots` is given as a function, it takes k to dynamically set the number of shots.

property backend: Backend | None

Backend object to be internally used.

property beta_iqpe: float

Circuit parameter β .

**build(state, evolution_operator_exponents, encoding_method, encoding_options=None,
eoe_totally_commuting=None, terms_map=None, paulis_map=None, time_split=None,
ctrlu_strat=CtrluStrat.PAULI_EXP_BOX)**

Build the controlled unitary function.

Parameters

- **state** (`GeneralAnsatz`) – Ansatz for the state preparation (non-symbolic).
- **evolution_operator_exponents** (`QubitOperatorList`) – List of Pauli strings to be trotterized.
- **encoding_method** (`CircuitEncoderQuantinum`) – Encoding method of the logical qubit implementation.
- **encoding_options** (`Union[PlainOptions, IcebergOptions, None]`, default: `None`) – Encoding options of the logical qubit implementation.
- **eoe_totally_commuting** (`Optional[QubitOperatorList]`, default: `None`) – Totally commuting set of Pauli strings.
- **terms_map** (`Optional[Mapping[QubitPauliString, QubitPauliString]]`, default: `None`) – Pauli term mapping used when the Trotterization.
- **paulis_map** (`Optional[Mapping[QubitPauliString, QubitPauliString]]`, default: `None`) – Pauli term mapping for the optimization with the dummy qubit.
- **time_split** (`Optional[list[float]]`, default: `None`) – Delta t splitting for the gate error suppression.
- **ctrlu_strat** (`CtrluStrat`, default: `<CtrluStrat.PAULI_EXP_BOX: 0>`) – Ctrl-U compilation strategy.

Return type

`IterativePhaseEstimationQuantinum`

**build_from_circuit(state, get_ctrlu, encoding_method, encoding_options=None,
get_ctrlu_totally_commuting=None)**

Build the protocol using the lower-level inputs (circuits).

Parameters

- **state** (`Circuit`) – State preparation circuit.
- **get_ctrlu** (`Callable[[int], Circuit]`) – Function to return a controlled unitary circuit.
- **encoding_method** (`CircuitEncoderQuantinuum`) – Encoding method of the logical qubit implementation.
- **encoding_options** (`Union[PlainOptions, IcebergOptions, None]`, default: `None`)
 - Encoding options of the logical qubit implementation.
- **get_ctrlu_totally_commuting** (`Optional[Callable[[int], Circuit]]`, default: `None`) – Function to return a controlled unitary circuit (totally commuting set)

Return type

`IterativePhaseEstimationQuantinuum`

Notes

`get_ctrlu_totally_commuting` is the IcebergCode only.

`clear()`

Resets the internal state of the object by clearing all stored data.

This method clears all stored data including state hashes, operator hashes, measurement setups, parameters, and associated dataframes and circuits.

Returns

`BaseIterativePhaseEstimationCircuit – self.`

`clear_cache()`

Clear the circuit cache.

`property compilation_level: CompilationLevel`

Compilation level.

`cost()`

Calculate a simple cost metric running the protocol.

The cost is calculated as the sum of the depth of each circuit multiplied by the number of shots.

Returns

`int – The cost value as an integer.`

`credits(syntax_checker=None, use_websocket=None)`

Evaluate an approximate cost for the measurement circuits built in Quantinuum credits.

Syntax checker will usually be automatically selected, but in some cases needs to be provided. If the measurement circuits are not built yet, the credits will be zero.

Note

This works only with `QuantinuumBackend` compatible backends.

Parameters

- **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 total cost in Quantinuum credits to run the circuits.

dataframe_circuit_shot()

Create a pandas `DataFrame` with circuit, shot, and depth information.

Returns

`DataFrame` – A pandas `DataFrame` containing the circuit, shot, and depth information.

get_circuits()

Returns the quantum circuits built for this protocol.

Returns

`List[Circuit]` – List of circuits.

get_circuitshots()

Generate the circuit shot pairs.

Yields

Pair of circuit and the associated number of shots.

Return type

`Iterator[CircuitShots]`

get_distribuition()

Get the distribution based on the measurement outcome.

Returns

`Mapping[Tuple[int, ...], float]` – Distribution.

property get_iqpe_circuit: Callable[[int, float], Circuit]

Interface to the function to generate the IQPE circuit.

Returns

function to build an IQPE circuit.

get_measurement_outcome()

Get the measurement outcome as a bit string.

Returns

`list[int]` – List of int in {0, 1} to be used by iterative QPE algorithms.

get_shots()

Returns the number shots to be used for each circuit.

Returns

`List[int]` – List of number of shots, the length of this list is the same as the number of circuits.

property is_built: bool

Boolean flag indicating if the instance has been built.

Returns

`True` if the `build()` method has been successfully invoked, otherwise `False`.

property is_numeric: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`False` if any built measurement circuit contains unsubstituted symbols, otherwise `True`.

property is_run: bool

Boolean flag indicating if the instance has been run.

Returns

`True` if the `run()` method has been successfully invoked, otherwise `False`.

property is_symbolic: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`True` if any built measurement circuit contains unsubstituted symbols, otherwise `False`.

property k_ippe: int

Circuit parameter k .

launch(kwargs)**

Launch the circuits to the backend and return the handles for the results.

This method processes all the circuits and returns a list of `ResultHandle` objects representing the handles for the results.

Parameters

- `*args` – Additional arguments to be passed to `self.backend.process_circuits()`.
- `**kwargs` (`Mapping[str, Any]`) – Additional keyword arguments to be passed to `self.backend.process_circuits()`.

Returns

`List[ResultHandle]` – A list of pytket `ResultHandle` objects representing the handles for the launched circuits.

property n_circuit: int

Returns the total number of circuits.

property n_shots: int

Number of shots.

property optimisation_level: int

Optimization level.

rebuild(*args, **kwargs)

Rebuild the internal data structure.

It is equivalent to `clear().build(*args, **kwargs)`.

Parameters

- `*args` (`Any`) – Arguments to be passed to `build()`.
- `**kwargs` (`Any`) – Keyword arguments to be passed to `build()`.

Returns

`TypeVar(TBuildClearMixin, bound=BuildClearMixin)` – `self`.

retrieve(source, **kwargs)

Retrieve distributions from the backend for the given source.

If the `source` is a list of pytket `ResultHandle`, the distributions are retrieved using `self.backend.get_results()` method. If the `source` is a list of pytket `BackendResult`, it is assumed that the results are already provided.

Parameters

- **source** (`Union[List[ResultHandle], List[BackendResult]]`) – A list of pytket `ResultHandle` or `BackendResult` objects representing the source of the distributions.
- ****kwargs** (`Mapping[str, Any]`) – Additional keyword arguments to be passed to `self.backend.get_results()` when retrieving results.

Returns`IterativePhaseEstimationQuantinuum` – Returns self instance.**run** (*`args`, **`kwargs`)

Run the protocol and waiting for the results.

This method executes the following steps: 1. Launches the measurement circuits to obtain the circuit handles.
2. Retrieves the measurement distributions using the circuit handles.

Parameters

- ***args** (`Any`) – Arguments to be passed to `launch()`.
- ****kwargs** (`Any`) – Keyword arguments to be passed to `launch()`.

Returns`TypeVar(TLaunchRetrieveMixin, bound=LaunchRetrieveMixin)` – self.**update_k_and_beta** (`k, beta`)

Update the circuit parameters k and beta.

Parameters

- **k** (`int`) – The number of repeats of the controlled unitary.
- **beta** (`float`) – Pre-measurement rotation angle of ancilla.

Return type`BaseIterativePhaseEstimationCircuit`**class IterativePhaseEstimationStatevector** (`state_backend=None, unitary_backend=None, n_shots=1`)Bases: `BaseIterativePhaseEstimation`

Iterative phase estimation using the likelihood calculated with statevector simulation.

This class is designed for prototyping iterative QPE algorithms, rather than performing the production runs. The `n_shots` is recognized as a number of samples taken from the exact likelihood.

The exact likelihood is internally constructed by diagonalizing the unitary and the state population is calculated using the state preparation circuit. The `get_distribution()` returns exact distribution, rather than that from the measurement outcome.

Parameters

- **state_backend** (`Optional[Backend]`, default: `None`) – pytket backend supporting `get_state()`.
- **unitary_backend** (`Optional[Backend]`, default: `None`) – pytket backend supporting `get_unitary()`.
- **n_shots** (`int`, default: `1`) – Number of shots (default: `1`).

Note

Currently, `state_backend=None` and `unitary_backend=None` are required. External backends are not supported as for now, but the built-in functions of `pytket.Circuit` are used instead.

build(*state*, *evolution_operator_exponents*, *eoe_totally_commuting*=None)

Build the internal objects needed for the exact likelihood.

It calculate the eigenvalues and their populations in the initial state by the exact diagonalization of the Hamiltonian.

Parameters

- **state** (*GeneralAnsatz*) – Ansatz for state preparation (non-symbolic).
- **evolution_operator_exponents** (*QubitOperatorList*) – List of Pauli strings to be trotterized.
- **eoe_totally_commuting** (*Optional[QubitOperatorList]*, default: `None`) – Totally commuting set of Pauli strings.

Returns

IterativePhaseEstimationStatevector – Self.

build_from_circuit(*state*, *get_ctrlu*)

Build the protocol using the lower-level inputs (circuits).

Parameters

- **state** (*Circuit*) – State preparation circuit.
- **get_ctrlu** (*Callable[[int], Circuit]*) – Function to return a CTRL-U circuit.

Return type

IterativePhaseEstimationStatevector

clear()

Resets the internal state of the object by clearing all stored data.

This method clears all stored data including state hashes, operator hashes, measurement setups, parameters, and associated dataframes and circuits.

Returns

IterativePhaseEstimationStatevector – self.

property eigenvalues: ndarray

Eigenvalues to be used for generating the likelihood.

Returns

Eigenvalues of the unitary $U = e^{-iHt}$.

get_distribuiton()

Get the distribution based on the measurement outcome.

Returns

Mapping[Tuple[int, ...], float] – Distribution.

get_distribution()

Distribution of the measurement outcome to be used by the iterative phase estimation algorithms.

Returns

Mapping[Tuple[int, ...], float] – Probability to measure “0” from the ancilla qubit.

Notes

It returns the exact distribution, rather than that calculated from the measurement outcome.

get_measurement_outcome()

Measurement outcome to be used by the iterative phase estimation algorithms.

Returns

`list[int]` – List of 0 or 1 as a measurement outcome of the IQPE circuit.

property is_built: bool

Boolean flag indicating if the instance has been built.

Returns

True if the `build()` method has been successfully invoked, otherwise `False`.

property is_run

Boolean flag indicating if the instance has been run.

Returns

True if the `run()` method has been successfully invoked, otherwise `False`.

launch()

Calculate the mock measurement outcome for the current k and β .

Return type

`int`

property populations: ndarray

Eigenstate populations to be used for generating the likelihood.

Returns

Eigenstate Populations.

rebuild(*args, **kwargs)

Rebuild the internal data structure.

It is equivalent to `clear().build(*args, **kwargs)`.

Parameters

- ***args** (`Any`) – Arguments to be passed to `build()`.
- ****kwargs** (`Any`) – Keyword arguments to be passed to `build()`.

Returns

`TypeVar(TBuildClearMixin, bound= BuildClearMixin)` – self.

retrieve(handle)

Retrieve the measurement outcome to get ready for invoking `get_measurement_outcome()`.

Parameters

`handle (int)` – The ID returned from `launch()` method.

Returns

Self instance.

run(*args, **kwargs)

Run the protocol and waiting for the results.

This method executes the following steps: 1. Launches the measurement circuits to obtain the circuit handles.
2. Retrieves the measurement distributions using the circuit handles.

Parameters

- ***args** (`Any`) – Arguments to be passed to `launch()`.
- ****kwargs** (`Any`) – Keyword arguments to be passed to `launch()`.

Returns

TypeVar(TLaunchRetrieveMixin, bound= LaunchRetrieveMixin) – self.

update_k_and_beta(*k*, *beta*)

Update the circuit parameters *k* and *beta*.

Parameters

- ***k*** (`int`) – The number of repeats of the controlled unitary.
- ***beta*** (`float`) – Pre-measurement rotation angle of ancilla.

Return type

IterativePhaseEstimationStatevector

27.11.7 Other protocols

class ProjectiveMeasurements(*backend*, *shots_per_circuit*)

Bases: LaunchRetrieveMixin, BuildClearMixin, GenerateCircuitShotMixin, PartiallyPickleable

This protocol measures the probability amplitudes of the computational basis states.

Examples

```
>>> from pytket.extensions.qiskit import AerBackend
>>> backend = AerBackend()
>>> circuit = Circuit(3).H(0).H(1).H(2)
>>> protocol = ProjectiveMeasurements(backend, 10000).build({}, ↴
    ↪CircuitAnsatz(circuit)).run(seed=0)
>>> protocol.get_dominant_basis_states(2)
[((1, 0, 0), 1278), ((0, 0, 0), 1275)]
>>> protocol.get_zero_state_probability()
0.1275
>>> protocol.get_zero_state_uncertainty()
0.0033353223232545307
>>> protocol.get_dataframe_basis_states(8)
   Basis State  Probability  Uncertainty  Count
0      100     0.1278    0.003339    1278
1      000     0.1275    0.003335    1275
2      011     0.1266    0.003325    1266
3      110     0.1262    0.003321    1262
4      101     0.1254    0.003312    1254
5      111     0.1241    0.003297    1241
6      001     0.1223    0.003276    1223
7      010     0.1201    0.003251    1201
```

Parameters

- ***backend*** (Backend) – The quantum backend to be used for measurements.
- ***shots_per_circuit*** (`int`) – The number of measurement shots.

build(*parameters*, *state*, *optimisation_level*=1)

Builds the necessary circuits and measurement data for the state.

Parameters

- **parameters** (`Union[SymbolDict, Dict]`) – A dictionary or `SymbolDict` containing the parameter values for the circuits.
- **state** (`GeneralAnsatz`) – The quantum state for which circuit is to be built.
- **optimisation_level** (`int`, default: 1) – Passed to the backend's `get_compiled_circuits()` method.

Returns

`ProjectiveMeasurements` – Self instance after building the necessary circuits and measurement data.

circuit: `Optional[Circuit]`

clear()

Resets the internal state of the object by clearing all stored data.

This method clears all stored data including state hashes, operator hashes, measurement setups, parameters, and associated dataframes and circuits.

Returns

`ProjectiveMeasurements` – self.

compile_circuits (`optimisation_level=1`)

Compiles circuits.

Parameters

- **optimisation_level** (`int`, default: 1) – Passed to the backend's `get_compiled_circuits()` method.

Returns

`ProjectiveMeasurements` – Self instance after compiling circuits.

cost()

Calculate a simple cost metric running the protocol.

The cost is calculated as the sum of the depth of each circuit multiplied by the number of shots.

Returns

`int` – The cost value as an integer.

counts: `Optional[Counter]`

credits (`syntax_checker=None, use_websocket=None`)

Evaluate an approximate cost for the measurement circuits built in Quantinuum credits.

Syntax checker will usually be automatically selected, but in some cases needs to be provided. If the measurement circuits are not built yet, the credits will be zero.

 **Note**

This works only with QuantinuumBackend compatible backends.

Parameters

- **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 total cost in Quantinuum credits to run the circuits.

`dataframe_circuit_shot()`

Create a pandas DataFrame with circuit, shot, and depth information.

Returns

`DataFrame` – A pandas DataFrame containing the circuit, shot, and depth information.

`dump(file)`

Save the object to a file using pickle.

Parameters

`file(Union[str, BinaryIO])` – The file path or file object to write the pickled data to.

Return type

`None`

`dumps()`

Returns the object pickled as a bytes object.

Returns

`bytes` – The object's state pickled as a bytes object.

`get_circuits()`

Returns the quantum circuits built for this protocol.

Returns

`List[Circuit]` – List of circuits.

`get_circuitshots()`

Generate the circuit shot pairs.

Yields

Pair of circuit and the associated number of shots.

Return type

`Iterator[CircuitShots]`

`get_dataframe_basis_states(max_rows)`

Get a DataFrame of basis states with probabilities and uncertainties.

Parameters

`max_rows(int)` – The maximum number of rows to include in the DataFrame.

Returns

A pandas DataFrame containing basis states, probabilities, uncertainties, and counts.

`get_distribution()`

Get the measurement distribution.

Returns

`Counter` – The measurement distribution.

`get_dominant_basis_states(n)`

Get the n most dominant basis states and their counts.

Parameters

`n(int)` – The number of dominant states to retrieve.

Returns

`List[Tuple[Tuple[int, ...], int]]` – A list of tuples containing basis states and their corresponding counts.

get_phaseless_qubit_state(n)

Builds a qubit state with the n most common basis states.

The coefficients are set as the sqrt(probability), and the state will be normalized.

 **Note**

This will not account for any phase the coefficients had originally.

Parameters

`n (int)` – The number of most common basis states to include in the state.

Returns

`QubitState` – The constructed qubit state.

get_shots()

Returns the number shots to be used for each circuit.

Returns

`List[int]` – List of number of shots, the length of this list is the same as the number of circuits.

get_zero_state_probability()

Get the probability of measuring the zero state.

Returns

`float` – The probability of measuring the zero state.

get_zero_state_uncertainty()

Get the uncertainty of the probability of measuring the zero state.

Returns

`float` – The uncertainty of the probability of measuring the zero state.

property is_built: bool

Boolean flag indicating if the instance has been built.

property is_numeric: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`False` if any built measurement circuit contains unsubstituted symbols, otherwise `True`.

property is_run: bool

Boolean flag indicating if the instance has been run.

Returns

`True` if the `run()` method has been successfully invoked, otherwise `False`.

property is_symbolic: bool

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`True` if any built measurement circuit contains unsubstituted symbols, otherwise `False`.

launch(*args, **kwargs)

Launch the circuits to the backend and return the handles for the results.

This method processes all the circuits and returns a list of `ResultHandle` objects representing the handles for the results.

Parameters

- ***args** – Additional arguments to be passed to `self.backend.process_circuits()`.
- ****kwargs** – Additional keyword arguments to be passed to `self.backend.process_circuits()`.

Returns

`List[ResultHandle]` – A list of pytket `ResultHandle` objects representing the handles for the launched circuits.

classmethod load(file, *args, **kwargs)

Load a pickled object from a file.

Parameters

- **file** (`Union[str, BinaryIO]`) – The file path or file object to load the pickled data from.
- ***args** (`Any`) – Additional arguments passed to the class constructor.
- ****kwargs** (`Any`) – Additional keyword arguments passed to the class constructor.

Returns

`TypeVar(T, bound= PartiallyPickleable)` – An instance of the class with its state loaded from the pickled data.

classmethod loads(pickled_data, *args, **kwargs)

Load a pickled object from a bytes object.

Parameters

- **pickled_data** (`bytes`) – The pickled data to load the object from.
- ***args** (`Any`) – Additional arguments passed to the class constructor.
- ****kwargs** (`Any`) – Additional keyword arguments passed to the class constructor.

Returns

`TypeVar(T, bound= PartiallyPickleable)` – An instance of the class with its state loaded from the pickled data.

property n_circuit: int

Returns the total number of circuits.

rebuild(*args, **kwargs)

Rebuild the internal data structure.

It is equivalent to `clear().build(*args, **kwargs)`.

Parameters

- ***args** (`Any`) – Arguments to be passed to `build()`.
- ****kwargs** (`Any`) – Keyword arguments to be passed to `build()`.

Returns

`TypeVar(TBuildClearMixin, bound= BuildClearMixin)` – self.

`results: Optional[List[BackendResult]]`

`retrieve(source, *args, **kwargs)`

Retrieve distributions from the backend for the given source.

If the `source` is a list of pytket ResultHandle, the distributions are retrieved using `self.backend.get_results()` method. If the `source` is a list of pytket BackendResult, it is assumed that the results are already provided.

Parameters

- `source` (`Union[List[ResultHandle], List[BackendResult]]`) – A list of pytket ResultHandle or BackendResult objects representing the source of the distributions.
- `**kwargs` – Additional keyword arguments to be passed to `self.backend.get_results()` when retrieving results.

Returns

`ProjectiveMeasurements` – Returns self instance.

`run(*args, **kwargs)`

Run the protocol and waiting for the results.

This method executes the following steps: 1. Launches the measurement circuits to obtain the circuit handles.
2. Retrieves the measurement distributions using the circuit handles.

Parameters

- `*args` (`Any`) – Arguments to be passed to `launch()`.
- `**kwargs` (`Any`) – Keyword arguments to be passed to `launch()`.

Returns

`TypeVar(TLaunchRetrieveMixin, bound=LaunchRetrieveMixin)` – self.

class ProtocolList

Bases: `LaunchRetrieveMixin, GenerateCircuitShotMixin`

A class for containing multiple shot-based protocols.

This class combines multiple built protocols together to collectively manage their launch and retrieve workflow, and provide combined evaluator functions.

Note

The collection will only accept protocols that have been built (`is_built == True`).

Usage:

- Instantiate backend, states and operator objects:

```
>>> from inquanto.protocols import ProtocolList, PauliAveraging,_
    ~HadamardTestOverlap
>>> from pytket.extensions.qiskit import AerBackend
>>> backend = AerBackend()
```

```
>>> from pytket.circuit import Circuit
>>> from inquanto.ansatzes import CircuitAnsatz
>>> state1 = CircuitAnsatz(Circuit(4).X(0).X(1))
>>> state2 = CircuitAnsatz(Circuit(4).X(2).X(3))
```

```
>>> from inquanto.operators import QubitOperator
>>> op = QubitOperator("Y0 X1 X2 X3")
```

- Create an instance of ProtocolList

```
>>> protocols = ProtocolList()
```

- Create a protocol (assuming BuildClearMixin is the base protocol)

```
>>> protocol_pa = PauliAveraging(backend)
>>> protocol_ho = HadamardTestOverlap(backend, 1000)
```

- Add the protocol to the protocol list

```
>>> protocols.append(protocol_pa.build({}, state1, op))
>>> protocols.append(protocol_ho.build({}, state1, state2, op, component=
    ↪"complex"))
```

- Run the protocol list, which will run each protocol in the list.

```
>>> _ = protocols.run(seed=0)
```

- Therefore the protocols in the list will have run status

```
>>> protocol_pa.is_run
True
>>> protocol_ho.is_run
True
```

- Total circuit analysis:

```
>>> protocols.dataframe_circuit_shot()
   Qubits Depth Depth2q DepthCX  Shots
0       4      2       0       0   8000
1       5      8       4       4   1000
2       5      8       4       4   1000
Sum     -     -     -     -  10000
```

`append(protocol)`

Appends a built protocol to the collection.

Parameters

`protocol` (BuildClearMixin) – The protocol to be appended.

Raises

`ValueError` – If the protocol has not been built yet.

Return type

`None`

cost()

Calculate a simple cost metric running the protocol.

The cost is calculated as the sum of the depth of each circuit multiplied by the number of shots.

Returns

`int` – The cost value as an integer.

credits(*syntax_checker=None, use_websocket=None*)

Evaluate an approximate cost for the measurement circuits built in Quantinuum credits.

Syntax checker will usually be automatically selected, but in some cases needs to be provided. If the measurement circuits are not built yet, the credits will be zero.

Note

This works only with QuantinuumBackend compatible backends.

Parameters

- `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 total cost in Quantinuum credits to run the circuits.

dataframe_circuit_shot()

Create a pandas `DataFrame` with circuit, shot, and depth information.

Returns

`DataFrame` – A pandas `DataFrame` containing the circuit, shot, and depth information.

dataframe_protocol_circuit()

Create a pandas `DataFrame` with protocol, circuit and shot information.

Returns

`DataFrame` – A pandas `DataFrame` containing the protocol, circuit and shot information.

get_circuits()

Returns the quantum circuits built for this protocol.

Returns

`List[Circuit]` – List of circuits.

get_circuitshots()

Generate the circuit shot pairs.

Yields

Pair of circuit and the associated number of shots.

Return type

`Iterator[CircuitShots]`

get_evaluator(*allow_partial=False*)

Returns an evaluator function to evaluate quantum computables.

This method creates and returns a function (evaluator) that takes in a quantum computable and evaluates it. The returned function iterates over the evaluators of constituent protocols. The input computable is evaluated by the first compatible evaluator.

Note

This evaluator works for quantum computables the protocols in the list have been built from.

Parameters

`allow_partial (bool, default: False)` – If `False`, evaluation will fail when an unsupported computable node is encountered. If `True`, unsupported nodes will be skipped.

Returns

`Callable[[Evaluable], Union[Evaluable, Any]]` – A function that can evaluate quantum computables.

`get_shots()`

Returns the number shots to be used for each circuit.

Returns

`List[int]` – List of number of shots, the length of this list is the same as the number of circuits.

`property is_numeric: bool`

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`False` if any built measurement circuit contains unsubstituted symbols, otherwise `True`.

`property is_run: bool`

Boolean flag indicating if the instance has been run.

Returns

`True` if the `run()` method has been successfully invoked, otherwise `False`.

`property is_symbolic: bool`

Boolean flag indicating if the measurement circuits contain free symbols.

Returns

`True` if any built measurement circuit contains unsubstituted symbols, otherwise `False`.

`launch(*args, **kwargs)`

Launch the circuits to the backend and return the handles for the results.

This method processes all the circuits and returns a list of `ResultHandle` objects representing the handles for the results.

Parameters

- `*args (Any)` – Additional arguments to be passed to `self.backend.process_circuits()`.
- `**kwargs (Any)` – Additional keyword arguments to be passed to `self.backend.process_circuits()`.

Returns

`List[ResultHandle]` – A list of pytket `ResultHandle` objects representing the handles for the launched circuits.

```
property n_circuit: int
```

Returns the total number of circuits.

```
retrieve (source, **kwargs)
```

Retrieve distributions from the backend for the given source.

If the `source` is a list of pytket ResultHandle, the distributions are retrieved using `self.backend.get_results()` method. If the `source` is a list of pytket BackendResult, it is assumed that the results are already provided.

Parameters

- **source** (`Union[List[ResultHandle], List[BackendResult]]`) – A list of pytket ResultHandle or BackendResult objects representing the source of the distributions.
- ****kwargs** (`Any`) – Additional keyword arguments to be passed to `self.backend.get_results()` when retrieving results.

Returns

`ProtocolList` – Returns self instance.

```
run (*args, **kwargs)
```

Run the protocol and waiting for the results.

This method executes the following steps: 1. Launches the measurement circuits to obtain the circuit handles.
2. Retrieves the measurement distributions using the circuit handles.

Parameters

- ***args** (`Any`) – Arguments to be passed to `launch()`.
- ****kwargs** (`Any`) – Keyword arguments to be passed to `launch()`.

Returns

`TypeVar(TLaunchRetrieveMixin, bound=LaunchRetrieveMixin)` – self.

27.11.8 Error mitigation

```
class PMSV(stabilisers, atol=1e-10)
```

Bases: NoiseMitigation

Appends the physical symmetries of a system into the pauli-strings measured by a quantum circuit.

Implements the Partition Measurement Symmetry Verification 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 `QubitOperatorString` objects.

Note

If measurement reduction is used in the protocol with strategy `PauliPartitionStrat.NonConflicting`, PMSV will overwrite the circuits with strategy `PauliPartitionStrat.CommutingSets`.

Parameters

- **stabilisers** (`List[QubitOperator]`) – List of state stabilizers as `QubitOperator` objects with only a single Pauli string in them.

- **atol** (`float`, default: `1e-10`) – Absolute tolerance for checking phase normalization of stabilizers.

post (*backend_results*, *measurement_setup*)

PMSV post method to modify backend results after measurements.

Parameters

- **backend_results** (`List[BackendResult]`) – The backend results to be modified.
- **measurement_setup** (`MeasurementSetup`) – The measurement setup used for the measurements.

Returns

`Tuple[List[BackendResult], MeasurementSetup]` – The modified backend results.

pre (*state_circuit*, *measurement_circuits*, *measurement_setup*)

PMSV pre method to modify measurement setup before measurements.

Parameters

- **state_circuit** (`Circuit`) – State preparation circuit.
- **measurement_circuits** (`List[Circuit]`) – The measurement circuits for the measurements.
- **measurement_setup** (`MeasurementSetup`) – The measurement setup to be modified.

Returns

`Tuple[Circuit, List[Circuit], MeasurementSetup]` – The modified measurement setup.

class SPAM (*backend*, *correlations=None*)

Bases: NoiseMitigation

Interface to tket's State Preparation And Measurement (SPAM) correction utility.

See [pytket SPAM API reference](#)

Parameters

- **backend** (`Backend`) – Backend to run calibration on.
- **correlations** (`Optional[List[List[Any]]]`, default: `None`) – The `qubit_subsets` parameter to be forwarded to `SpamCorrecter` in pytket.

calibrate (*calibration_shots=50*, `**kwargs`)

Generates the SPAM transition matrix by running tomography circuits.

Parameters

- **calibration_shots** (`int`, default: 50) – Number of calibration shots.
- **kwargs** (`Any`)

Returns

`SPAM` – Self instance.

calibrate_process (*calibration_shots=50*, `**kwargs`)

Generates the SPAM transition matrix results by running tomography circuits.

Parameters

- **calibration_shots** (`int`, default: 50) – Number of calibration shots.

- **kwargs** ([Any](#))

Returns

`list[ResultHandle]` – Self instance.

calibrate_retrieve(handles)

Retrieves the SPAM transition matrix results.

Parameters

`handles` (`list[ResultHandle]`) – List of results from the Backend generated by `calibrate_process()`.

Returns

`SPAM` – Self instance.

post(backend_results, measurement_setup)

SPAM post method to modify backend results after measurements.

Parameters

- `backend_results` (`List[BackendResult]`) – The backend results to be modified.
- `measurement_setup` (`MeasurementSetup`) – The measurement setup used for the measurements.

Returns

`Tuple[List[BackendResult], MeasurementSetup]` – The modified backend results.

pre(state_circuit, measurement_circuits, measurement_setup)

SPAM pre method to modify measurement setup before measurements.

 **Note**

The method will compile circuits via a backend with optimisation level 1.

Parameters

- `state_circuit` (`Circuit`) – State preparation circuit.
- `measurement_circuits` (`List[Circuit]`) – The measurement circuits for the measurements.
- `measurement_setup` (`MeasurementSetup`) – The measurement setup to be modified.

Returns

`Tuple[Circuit, List[Circuit], MeasurementSetup]` – The modified measurement setup.

class CombinedMitigation(mitms1, mitms2)

Bases: NoiseMitigation

A NoiseMitigation subclass to combine two NoiseMitigation objects.

This method allows you to compose two NoiseMitigation instances in such a way that the pre and post methods of the instances are called in sequence. The pre method of the second instance is called with the result of the pre method of the first instance, and the post method of the first instance is called with the result of the post method of the second instance.

That is `(ms1 @ ms2).pre(...)` is equivalent to `ms2.pre(ms1.pre(...))` and `(ms1 @ ms2).post(...)` is equivalent to `ms1.post(ms2.post(...))`.

Parameters

- **mitms1** (NoiseMitigation) – First NoiseMitigation object.
- **mitms2** (NoiseMitigation) – Second NoiseMitigation object.

post (*backend_results*, *measurement_setup*)

The post method to modify backend results after measurements.

This returns `ms1.post(ms2.post(backend_results, measurement_setup))`.

Parameters

- **backend_results** (`List[BackendResult]`) – The backend results to be modified.
- **measurement_setup** (MeasurementSetup) – The measurement setup used for the measurements.

Returns

`Tuple[List[BackendResult], MeasurementSetup]` – The modified backend results.

pre (*state_circuit*, *measurement_circuits*, *measurement_setup*)

The pre method to modify measurement setup before measurements.

This returns `ms2.pre(ms1.pre(circuits, measurement_setup))`.

Parameters

- **state_circuit** (Circuit) – State preparation circuit.
- **measurement_circuits** (`List[Circuit]`) – The measurement circuits for the measurements.
- **measurement_setup** (MeasurementSetup) – The measurement setup to be modified.

Returns

`Tuple[Circuit, List[Circuit], MeasurementSetup]` – The modified measurement setup.

27.11.9 Utility classes

```
class ProtocolCache(level=CacheLevels.LIFETIME, max_mem_size=None, size_unit=CacheSizeUnit.MB,
key_generator=None)
```

Bases: `Cache`

Specialises Cache to include protocol-specific functionality.

Parameters

- **level** (`CacheLevels`, default: `CacheLevels.LIFETIME`) – Cache level (`CacheLevels.NONE`: nothing is cached, or `CacheLevels.LIFETIME`: cache is not cleared during lifetime of the object, using it (unless memory used exceeds `max_mem_size` parameter value).
- **max_mem_size** (`Optional[int]`, default: `None`) – Maximum memory size that cache is allowed to occupy. Not checked if set to `None`.
- **size_unit** (`CacheSizeUnit`, default: `CacheSizeUnit.MB`) – Memory size unit (used to process input and output).
- **key_generator** (`Optional[Callable[..., Hashable]]`, default: `None`) – Function to generate a key from the input objects.

Raises

`ValueError` – When `max_mem_size` parameter has negative value.

property cache: Dict[Any, Any]
 Returns the internal dict, storing cache.

property check_mem: bool
 Returns whether the size of memory occupied by cache is to be checked at runtime.

Notes

Currently the `Cache` class does not check for memory size itself (except for the report). Such checks are supposed to be done by a handler (e.g. a caching decorator).

clear()

Clears cache.

Return type

`None`

hashkey(*args, **kwargs)

Creates a hashable key from input arguments.

Overwrites same method from base class. Doesn't use `key_generator` method that can be optionally passed as an argument to the base class constructor.

Parameters

- `args (Any)` – Arguments, to be made hashable.
- `kwargs (Any)` – Keyword arguments, which values are to be made hashable.

Returns

`Tuple[Any, ...]` – A list of hashable objects, to be used as a cache dict key.

Raises

`RuntimeError` – If created key is not hashable.

property level: CacheLevels

Returns the cache level.

property max_mem_size: int | None

Returns the maximum allowed memory size for cache.

property mem_size: float

Calculates size of memory occupied by cache.

Currently uses implementation given `here` and converts result (in bytes) to the set memory size units.

property num_entries: int

Returns the number of entries currently stored in cache.

report()

Returns a pandas `DataFrame` object containing a report of the current state of cache.

Return type

`DataFrame`

property size_unit: CacheSizeUnit

Returns the memory size unit as used by the class instance.

```
class PlainOptions(n_plus_states: int = 1)
```

Bases: `NamedTuple`

Plain code (physical qubit implementation) options for the `IterativePhaseEstimationQuantinuum` class.

Parameters

- ***n_plus_states*** – Number of $|+\rangle$ states in the initial state.

n_plus_states*: *int

Alias for field number 0

```
class IcebergOptions(n_plus_states: int = 1, syndrome_interval: int = -1, sx_insertion: bool = False,  
                          conditional_exit: bool = False)
```

Bases: `NamedTuple`

Iceberg code options for the `IterativePhaseEstimationQuantinuum` class.

Parameters

- ***n_plus_states*** – Number of $|+\rangle$ states in the initial state.
- ***syndrome_interval*** – Syndrome measurement interval in the number of CTRL-U operations.
- ***sx_insertion*** – X stabilizer insertion for the dynamical decoupling.
- ***conditional_exit*** – Conditional exit (QuantinuumBackend only).

Notes

If the `conditional_exit` is used,

```
options = {  
    'compiler_options': {  
        'conditional_branching': True,  
        'merge_classical_branches': True,  
        'conditional_sq_gates': True  
    }  
}
```

must be passed to the `QuantinuumBackend.process_circuit()` function.

conditional_exit*: *bool

Alias for field number 3

n_plus_states*: *int

Alias for field number 0

sx_insertion*: *bool

Alias for field number 2

syndrome_interval*: *int

Alias for field number 1

```
class CircuitEncoderQuantinuum(value, names=None, *, module=None, qualname=None, type=None, start=1,  
                          boundary=None)
```

Bases: `Enum`

Encoder options for the `IterativePhaseEstimationQuantinuum` class.

PLAIN

No logical qubit encoding to run physical qubit experiments.

ICEBERG

Apply $[[k+2, k, 2]]$ error detection code (dotted Iceberg code).

ICEBERG = 1**PLAIN = 0**

```
class CompilationLevel(value, names=None, *, module=None, qualname=None, type=None, start=1,  
    boundary=None)
```

Bases: `int, Enum`

Compilation level used by `IterativePhaseEstimation`.

`COMPILED` is needed for a job to be run on a backend. The other options are used for analyzing the circuit.

LOGICAL

Flag to return the raw logical circuit.

COMPILED

Flag to return the compiled and optimized circuit for a given backend.

COMPILED = 0**LOGICAL = -1****as_integer_ratio()**

Return integer ratio.

Return a pair of integers, whose ratio is exactly equal to the original int and with a positive denominator.

```
>>> (10).as_integer_ratio()  
(10, 1)  
>>> (-10).as_integer_ratio()  
(-10, 1)  
>>> (0).as_integer_ratio()  
(0, 1)
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)  
'0b1101'  
>>> (13).bit_count()  
3
```

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)  
'0b100101'  
>>> (37).bit_length()  
6
```

conjugate()

Returns self, the complex conjugate of any int.

denominator

the denominator of a rational number in lowest terms

from_bytes (*byteorder='big'*, *, *signed=False*)

Return the integer represented by the given array of bytes.

bytes

Holds the array of bytes to convert. The argument must either support the buffer protocol or be an iterable object producing bytes. Bytes and bytearray are examples of built-in objects that support the buffer protocol.

byteorder

The byte order used to represent the integer. If byteorder is ‘big’, the most significant byte is at the beginning of the byte array. If byteorder is ‘little’, the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value. Default is to use ‘big’.

signed

Indicates whether two’s complement is used to represent the integer.

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

real

the real part of a complex number

to_bytes (*length=1*, *byteorder='big'*, *, *signed=False*)

Return an array of bytes representing an integer.

length

Length of bytes object to use. An OverflowError is raised if the integer is not representable with the given number of bytes. Default is length 1.

byteorder

The byte order used to represent the integer. If byteorder is ‘big’, the most significant byte is at the beginning of the byte array. If byteorder is ‘little’, the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value. Default is to use ‘big’.

signed

Determines whether two’s complement is used to represent the integer. If signed is False and a negative integer is given, an OverflowError is raised.

class CtrlUStrat (*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `int`, `Enum`

Enum of the strategy of CTRL-U compilation.

PAULI_EXP_BOX

Use PauliExpBox using CTRL-P(x) = ZP(-x/2)IP(x/2).

Pauli_GADGET_RZZ

Use Pauli gadgets but the CX Rz CX is replaced with Rzz.

PAULI_EXP_BOX = 0

PAULI_GADGET_RZZ = 1

as_integer_ratio()

Return integer ratio.

Return a pair of integers, whose ratio is exactly equal to the original int and with a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

conjugate()

Returns self, the complex conjugate of any int.

denominator

the denominator of a rational number in lowest terms

from_bytes (byteorder='big', *, signed=False)

Return the integer represented by the given array of bytes.

bytes

Holds the array of bytes to convert. The argument must either support the buffer protocol or be an iterable object producing bytes. Bytes and bytearray are examples of built-in objects that support the buffer protocol.

byteorder

The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value. Default is to use 'big'.

signed

Indicates whether two's complement is used to represent the integer.

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

real

the real part of a complex number

to_bytes (*length=1*, *byteorder='big'*, *, *signed=False*)

Return an array of bytes representing an integer.

length

Length of bytes object to use. An OverflowError is raised if the integer is not representable with the given number of bytes. Default is length 1.

byteorder

The byte order used to represent the integer. If byteorder is ‘big’, the most significant byte is at the beginning of the byte array. If byteorder is ‘little’, the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder` as the byte order value. Default is to use ‘big’.

signed

Determines whether two’s complement is used to represent the integer. If signed is False and a negative integer is given, an OverflowError is raised.

27.12 inquanto.spaces

This module provides classes for representing various types of Hilbert spaces; chiefly useful in the generation of operators on those spaces.

class FermionSpace (*n_spin_orb*, *point_group=None*, *orb_irreps=None*)

Bases: OccupationSpace

A class providing utilities associated with a fermionic occupation space.

This mostly consists of tools to generate specific operators acting on and states within a given fermionic Hilbert space.

Parameters

- ***n_spin_orb*** (`int`) – The number of spin orbitals in the system.
- ***point_group*** (`Union[PointGroup, str]`, default: `None`) – The point group symmetry of the system.
- ***orb_irreps*** (`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.

    This generates a boolean list, where the elements are True if the spin-orbital is acted upon by the operator or an operator in the list, otherwise False.

    Parameters
        operators (Union[FermionOperator, List[FermionOperator]]) – An operator or list of operators.

    Returns
        List[bool] – 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 FermionOperatorList 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 in fermion_state.

    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.

    Returns
        FermionOperatorList – Generalised double excitation operators stored independently.

construct_generalised_double_ucc_operators()
    Generate a FermionOperatorList of generalised anti-Hermitian double excitation operators.

    Each UCC excitation consists of an “excitation - de-excitation”,  $(a_p^\dagger a_q - a_q^\dagger a_p)$ , where p, q are generalised indices, virtual-virtual, occupied-occupied, and occupied-virtual excitations are all allowed.
```

Notes

These excitations do not distinguish occupied and virtual spaces, therefore no reference state is required here. Note that spin crossover excitations (that change number of alphas/betas) are not allowed. See [acs.jctc.8b01004](#) for further details.

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 *FermionOperatorList* 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

These excitations do not distinguish occupied and virtual spaces, therefore no reference state is required here. Note that spin crossover excitations (that change number of alphas/betas) are not allowed. See [acs.jctc.8b01004](#) for further details.

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 *FermionOperatorList* of generalised anti-Hermitian single excitation operators.

Each UCC excitation consists of an “excitation - de-excitation”, $(a_p^\dagger a_q - a_q^\dagger a_p)$, where p, q are generalised indices, virtual-virtual, occupied-occupied, and occupied-virtual excitations are all allowed.

Notes

These excitations do not distinguish occupied and virtual spaces, therefore no reference state is required here. Note that spin crossover excitations (that change number of alphas/betas) are not allowed. See [acs.jetc.8b01004](#) for further details.

Returns

FermionOperatorList – Generalised single UCC operators stored independently

`construct_n_body_spinless_pdm_operators(rank)`

Return an array of `FermionOperator` objects for measurement of spin-traced n -body pre-density matrix.

PDM is what is called the pre-density matrix in Orca and returned by `pyscf.fci.rdm.make_dm1234`. In this function, `pdm2[p, q, r, s]` is $\sum_{\sigma\tau} a_{p\sigma}^\dagger a_{q\sigma} a_{r\tau}^\dagger a_{s\tau}$ `pdm3[p, q, r, s, t, u]` is $\sum_{\sigma\tau\mu} a_{p\sigma}^\dagger a_{q\sigma} a_{r\tau}^\dagger a_{s\tau} a_{t\mu}^\dagger a_{u\mu}$. `pdm4[p, q, r, s, t, u, v, w]` is $\sum_{\sigma\tau\mu\nu} a_{p\sigma}^\dagger a_{q\sigma} a_{r\tau}^\dagger a_{s\tau} a_{t\mu}^\dagger a_{u\mu} a_{v\nu}^\dagger a_{w\nu}$. where σ, τ, μ, ν are spin indices. The 1-PDM is the same as the 1-RDM.

Parameters

- `rank (int)` – Rank n of the n -PDM.

Returns

A $2n$ -dimensional matrix of `FermionOperator` objects.

`construct_n_body_spinless_rdm_operators(rank, ordering='p^r^sq')`

Return an array of `FermionOperator` objects for measurement of spin-traced n -body reduced density matrix.

Parameters

- `rank (int)` – Rank n of the n -RDM.
- `ordering (str, default: "p^r^sq")` – Ordering of the spatial orbital indices given by either "`p^r^sq`" or "`p^q^sr`". This choice is expressed in terms of the indices of the 2-RDM because that is the lowest rank of RDM for which these orderings are non-equivalent. The returned array element of the e.g. 2-RDM accessed with indices `[p, q, r, s]` corresponds to operator terms of either $\sum_{\sigma\tau} a_{p\sigma}^\dagger a_{r\tau}^\dagger a_{s\tau} a_{q\sigma}$ with `ordering="p^r^sq"`, or $\sum_{\sigma\tau} a_{p\sigma}^\dagger a_{q\tau}^\dagger a_{s\tau} a_{r\sigma}$ with `ordering="p^q^sr"`, where σ, τ are spin indices.

The returned array element of the e.g. 3-RDM accessed with indices `[p, q, r, s, t, u]` corresponds to operator terms of either $\sum_{\sigma\tau\mu} a_{p\sigma}^\dagger a_{q\tau}^\dagger a_{r\mu}^\dagger a_{u\tau} a_{s\sigma} a_{q\mu}$ with `ordering="p^r^sq"`, or $\sum_{\sigma\tau\mu} a_{p\sigma}^\dagger a_{q\tau}^\dagger a_{r\mu}^\dagger a_{u\tau} a_{t\sigma} a_{s\mu}$ with `ordering="p^q^sr"`, where σ, τ, μ are spin indices.

Returns

`ndarray[Any, dtype[FermionOperator]]` – A $2n$ -dimensional matrix of `FermionOperator` objects.

`construct_number_alpha_operator()`

Construct a `FermionOperator` which represents a number operator acting on all alpha spin orbitals.

Returns

FermionOperator – The number operator over alpha spin-orbitals only.

`construct_number_beta_operator()`

Construct a `FermionOperator` which represents a number operator acting on all beta spin orbitals.

Returns

FermionOperator – The number operator over beta spin-orbitals only.

construct_number_operator()

Creates and returns a `FermionOperator` representation of the number operator.

The `FermionOperator` acts on the entire set of spin orbitals.

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 a one-body spatial integral array.

The indices are transformed to spin-orbital notation, with spin-minor ordering (i.e. alternating alpha and beta spins). The one-body operator is given by $\hat{h} = \sum_{pq} h_{pq} a_p^\dagger a_q$, where h_{pq} are the one-body integrals.

Parameters

- `one_body_spatial` (`ndarray[Any, dtype[float]]`) – Integrals in chemist's notation.
- `spins` (`Tuple[int, int]`) – The spin configuration, e.g. `(0, 0) = (SPIN_UP, SPIN_UP)`.
- `spatial_mask` (default: `None`) – 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 data type.

Parameters

- `spins` (`Tuple[int, int]`) – The corresponding spins as (s_i, s_j) .
- `spatial_mask` (`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 annihilation operators, respectively.

Returns

`ndarray[Any, dtype[FermionOperator]]` – The one body spatial RDM operator as a rank-2 array of `FermionOperator` objects.

static construct_operator_from_string(input)

Construct an operator from a string.

Parameters

`input` (`str`) – String representation of a `FermionOperator`.

Returns

`FermionOperator` – An instance of `FermionOperator` corresponding to the string provided.

construct_orbital_number_operators()

Construct a `list` of `FermionOperator` objects, 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]` – The individual orbital number operators.

```
static construct_scalar_operator(value)
```

Construct a FermionOperator consisting of only the identity term multiplied by a scalar coefficient.

Parameters

value (*float*) – Coefficient multiplying 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 anti-Hermitian single excitation operators based on a reference *fermion_state*.

Each UCC excitation consists of an “excitation - de-excitation”, $(a_p^\dagger a_q - a_q^\dagger a_p)$, 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 FermionOperatorList of spin-adapted singlet double excitation operators based on a reference *fermion_state*.

Each excitation is chemist ordered and spin-traced, $a_{p_0}^\dagger a_{q_0} a_{r_0}^\dagger a_{s_0} + a_{p_1}^\dagger a_{q_1} a_{r_1}^\dagger a_{s_1} + a_{p_0}^\dagger a_{q_0} a_{r_1}^\dagger a_{s_1} + a_{p_1}^\dagger a_{q_1} a_{r_1}^\dagger a_{s_1}$, 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 FermionOperatorList of anti-Hermitian singlet 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 in *fermion_state*.

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-traced, $a_{p_0}^\dagger a_{q_0}^\dagger a_{r_0} a_{s_0} + a_{p_1}^\dagger a_{q_1}^\dagger a_{r_1} a_{s_1}$, 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-traced, $a_{p_0}^\dagger a_{q_0} + a_{p_1}^\dagger a_{q_1}$, 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 *FermionOperatorList* of generalised anti-Hermitian single singlet spin-adapted excitation operators.

Each UCC excitation consists of an “excitation - de-excitation”, $(a_p^\dagger a_q - a_q^\dagger a_p)$, where both p and q run over occupied and virtual orbitals.

Notes

These excitations do not distinguish occupied and virtual spaces, therefore no reference state is required here. Note that spin crossover excitations (that change number of alphas/betas) are not allowed. See [acs.jctc.8b01004](#) for further details.

Returns

FermionOperatorList – The singlet generalised UCC single excitations.

construct_singlet_single_excitation_operators (fermion_state)

Generate a *FermionOperatorList* of spin-adapted singlet single excitation operators based on a reference *fermion_state*.

Each excitation is chemist ordered and spin-traced, $a_{p_0}^\dagger a_{q_0} + a_{p_1}^\dagger a_{q_1}$, 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 FermionOperatorList of anti-Hermitian singlet single excitation operators based on a reference *fermion_state*.

Each UCC excitation consists of an “excitation - de-excitation”, $(a_p^\dagger a_q - a_q^\dagger a_p)$, where *p* is virtual and *q* is occupied in *fermion_state*.

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 \hat{S}^2 operator.

Returns

FermionOperator – The \hat{S}^2 operator.

construct_sz_operator()

This function creates and returns a FermionOperator representation of the \hat{S}_z operator.

Returns

FermionOperator – The \hat{S}_z 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-traced, $a_{p_0}^\dagger a_{q_0} - a_{p_1}^\dagger a_{q_1}$, $a_{p_0}^\dagger a_{q_1}$, or $a_{p_1}^\dagger a_{q_0}$, 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 FermionOperatorList of generalised anti-Hermitian single excitation operators, spin adapted to a triplet.

Each UCC excitation consists of an “excitation - de-excitation”, $(a_p^\dagger a_q - a_q^\dagger a_p)$, where *p*, *q* are generalised indices.

 **Notes**

These excitations do not distinguish occupied and virtual spaces, therefore no reference state is required here. Note that spin crossover excitations (that change number of alphas/betas) are not allowed. See [acs.jctc.8b01004](#) for further details.

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 a two-body spatial integral array.

$\sum_{ijkl} (ij|kl) a_{i,s_i}^\dagger a_{k,s_k}^\dagger a_{l,s_l} a_{j,s_j}$ where $s_i, s_j, s_k, s_l \in \{0, 1\}$ are the elements of *spins*.

Notes

The integrals are in chemist's notation, i.e. `two_body_spatial[i, j, k, l] = (ij|kl)`.

Parameters

- `two_body_spatial` (`ndarray[Any, dtype[float]]`) – Integrals in chemist's 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 a two-body spatial integral array.

That is, $\sum_{ijkl} (ij|kl) a_{i,s_i}^\dagger a_{k,s_k}^\dagger a_{l,s_l} a_{j,s_j}$ where $s_i, s_j, s_k, s_l \in \{0, 1\}$ are the elements of spins.

Notes

The integrals are in chemist's notation, that is `two_body_tensor[i, k, l, j] = (ij|kl)`. The difference from the `two_body_spatial` argument to `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 chemist's 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 `FermionOperator` objects 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 annihilation operators, respectively.

Returns

`ndarray[Any, dtype[FermionOperator]]` – The two body spatial RDM operators.

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]]` – 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 `FermionOperator` instances to be contracted.

Returns

`Tuple[FermionSpace, FermionState, FermionOperator]` – The contracted `FermionSpace`, contracted `FermionState`, and 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]` – Positional indices of `mask` elements with value `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 deprecated legacy method and will be removed in subsequent releases.

 **Notes**

If some value is `None`, then it is counted.

Parameters

- **fermion_state** (`FermionState`, default: `None`) – A `FermionState` object.
- **state_value** (`int`, default: `1`) – `1/0` signifying occupied/unoccupied, respectively.
- **column** (`int`, default: `None`) – The column index in the quantum number table.
- **column_value** (`int`, default: `None`) – The value to be counted.
- **state_mask** (`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 `FermionState` instance representing a fermionic state vector.

Returns

`FermionSpace` – An instance of `FermionSpace` with C_1 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

`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – Complete set of masks.

Examples

```
>>> space = FermionSpace(8)
>>> space.generate_cyclic_masks(window=4, shift=2)
array([[ True,  True,  True,  True, False, False, False, False],
       [False, False,  True,  True,  True, False, False, False],
       [False, False, False,  True,  True,  True,  True, False],
       [ True,  True, False, False, False,  True,  True,  True]])
```

generate_cyclic_window_mask(*window*, *shift*=0)

Generate a mask with a window and shift.

The `window` defines how many elements in the result are `True`, with the `shift` determining the start index of the window. See below for example.

Parameters

- **window** (`int`) – Number of spin orbitals within the window.
- **shift** (default: 0) – The position in the occupation number vector to begin the window.

Returns

`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – Booleans defining the mask.

Examples

```
>>> space = FermionSpace(8)
>>> space.generate_cyclic_window_mask(window=4, shift=2)
array([False, False,  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 minus one.

Parameters

- **n_fermion** (`int`, default: 0) – Number of Fermions to include in `FermionState`.
- **multiplicity** (`int`, default: `None`) – Multiplicity of the generated `FermionState`.

Returns

`FermionState` – The generated fermionic state.

generate_occupation_state_from_list(*occupation_state_list=None*)

Convert spin-orbital occupation numbers to a FermionState.

Parameters

occupation_state_list (`List[int]`, default: `None`) – Represents spin-orbital occupations. Default value is `None`, which returns the all zeros FermionState (i.e. the vacuum state).

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]`) – n entries with values of 0, 1, or 2, where n is the number of spatial orbitals. Each entry indicates 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

`FermionOperator` 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

`FermionOperator` objects corresponding to spin-adapted, triplet single excitation operators.

Return type

`Generator[FermionOperator, None, None]`

get_orb_irreps_dataframe()

Return a pandas `DataFrame` listing orbitals and their respective irreducible representations.

Assumes spin minor ordering of spin orbitals i.e. [0a, 0b, 1a, 1b...].

Returns

`DataFrame` – Dataframe detailing orbital irreps.

index(*orb, spin*)

Encode spatial orbital index and spin index as 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.

property `n_spin_orb: int`

Return the number of spin-orbitals.

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}
```

orb_irreps()

Get orbital irreducible representations.

Returns

`List[str]` – Orbital irreps.

point_group(as_string=True)

Return the point group symmetry of the space.

Parameters

as_string (`bool`, default: `True`) – If `True`, returns name of point group as string. If `False`, returns an InQuanto `PointGroup` object.

Returns

`Union[str, PointGroup]` – Point group symmetry.

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.

Return type

None

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 .
```

quantum_label(*i*)

Generates a label for a given spin-orbital formed as a concatenation of its 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] – 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 a 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`.

 **Warning**

This method is deprecated and will be removed in InQuanto v5.0.0

Parameters

- `fermion_state (FermionState, default: None)` – A FermionState object.
- `state_value (default: 1)` – State value.
- `column (int, default: None)` – The column in the underlying table to select from, 0 for spin, 1 for orbital.
- `column_value (int, default: None)` – The value of interest for the given column.
- `state_mask (List[bool], default: None)` – A boolean mask 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)`

Constructs symmetry operators corresponding to point group, spin- and particle-number parity \mathbb{Z}_2 symmetries.

Danger

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]]` – The \mathbb{Z}_2 symmetry operators in a format determined by the `return_factorized` argument.

```
symmetry_operators_z2_in_sector(fermion_state, spin_ordering='abab', point_group=True,
                                 return_factorized=False)
```

Constructs \mathbb{Z}_2 symmetry operators which stabilize a provided state.

Symmetry operators returned correspond to the point group, spin parity and particle number parity \mathbb{Z}_2 symmetries, as per `symmetry_operators_z2()`.

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.

Danger

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]]` – The \mathbb{Z}_2 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.

This generates a boolean `list`, where the elements are `True` if the spin-orbital is acted upon by the operator or an operator in the list, otherwise `False`.

Parameters

`operators` (`Union[FermionOperator, List[FermionOperator]]`) – An operator or `list` of operators.

Returns

`List[bool]` – 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 chemist notation.
- **spins** (Tuple[int, int]) – The spins, e.g. (0, 0) = (SPIN_UP, SPIN_UP).
- **kpoints** (Tuple[int, int]) – The k-points, e.g. (0, 1) = (k_i, k_j).
- **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

$$\sum_{ijkl} (ij|kl) a_{i,s_i,k_i}^\dagger a_{k,s_k,k_k}^\dagger a_{l,s_l,k_l} a_{j,s_j,k_j}$$
 (27.2)

where *s_i, s_j, s_k, s_l* are spin indices and *k_i, k_j, k_k, k_l* are k-points.

Notes

The integrals are in chemist's notation, i.e. *two_body_spatial*[i, j, k, l] = (ij|kl).

Parameters

- **two_body_spatial** (ndarray[*Any*, dtype[*float*]]) – Integrals in chemist's 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** (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) – Active spatial orbital indices.
- **active_kps** (Optional[List[int]], default: None) – Active k-points.

Returns

`Tuple[FermionSpaceBrillouin, List[bool]]` – 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 the provided fermionic state.

 **Notes**

This currently works only with FermionOperator.

Parameters

- **fermion_state** (*FermionState*) – A state providing reference occupation numbers.
- **contraction_mask** (List[bool]) – A mask where entries correspond to spin-orbitals. 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]` – Positional indices of *mask* elements with value `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 deprecated legacy method and will be removed in subsequent releases.

 **Notes**

If some value is `None`, then it is counted.

Parameters

- **fermion_state** (*FermionState*, default: `None`) – A *FermionState* object.
- **state_value** (`int`, default: `1`) – 1/0 signifying occupied/unoccupied, respectively.
- **column** (`int`, default: `None`) – The column index in the quantum number table.
- **column_value** (`int`, default: `None`) – The value to be counted.
- **state_mask** (`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** (`int`, default: `None`) – Multiplicity of the state per k-point.

Returns

FermionState – The generated *FermionState*.

generate_occupation_state_from_list (*fermion_state_list=None*)

Convert occupations to an occupation state.

Parameters

fermion_state_list (`Optional[List[int]]`, default: `None`) – Occupation numbers.

Returns

FermionState – The generated fermion state.

generate_occupation_state_from_spatial_occupation (*occupation*)

Generates a *FermionState* from spatial orbital occupations per k-point.

Parameters

occupation (`List[List[int]]`) – *list* instances that contain occupation values of 0, 1, or 2. The 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

Number of k-points.

property n_spin_orb: int

Return the number of spin-orbitals.

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.

Return type*None***i 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   .
```

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] – Quantum numbers for k-points, spatial orbitals, and 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`.

 **Warning**

This method is deprecated and will be removed in InQuanto v5.0.0

Parameters

- `fermion_state (FermionState, default: None)` – A FermionState object.
- `state_value (default: 1)` – State value.
- `column (int, default: None)` – The column in the underlying table to select from, 0 for spin, 1 for orbital.
- `column_value (int, default: None)` – The value of interest for the given column.
- `state_mask (List[bool], default: None)` – A boolean mask 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

Represents a 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.

This generates a boolean list, where the elements are `True` if the spin-orbital is acted upon by the operator or an operator in the list, otherwise `False`.

Parameters

`operators` (`Union[FermionOperator, List[FermionOperator]]`) – An operator or list of operators.

Returns

`List[bool]` – 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** (`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_co, bound=generic, covariant=True)]]`) – Integrals in chemist's notation for the supercell.
- **spins** (`Tuple[int, int]`) – The spins, e.g. $(0, 0) = (\text{SPIN_UP}, \text{SPIN_UP})$.
- **spatial_mask** (default: `None`) – Mask filtering the generations of terms in the operator.

Returns

`FermionOperator` – The one-body operator.

`construct_permutation_operator(perm)`

Construct a fermionic permutation operator from a permutation.

The iterable of length two iterables must contain the two indices to be swapped.

Parameters

- **perm** (`Union[ndarray[Any, dtype[int]], List[int]]`) – Permutation array.

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

$$\sum_{ijkl} (ij|kl) a_{i,s_i}^\dagger a_{k,s_k}^\dagger a_{l,s_l} a_{j,s_j}. \quad (27.3)$$

Where s_i, s_j, s_k, s_l are the spin indices.

Notes

The integrals are in chemist's notation, that is `two_body_spatial[i, j, k, l] = (ij|kl)`.

Parameters

- `two_body_spatial` (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]`) – Integrals in chemist's notation for the supercell.
- `spins` – The corresponding spins as s_i, s_j, s_k, s_l .
- `spatial_mask` (default: `None`) – 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 the provided fermionic state.

Notes

This currently works only with `FermionOperator`.

Parameters

- `fermion_state` (`FermionState`) – A state providing reference occupation numbers.

- **contraction_mask** (`List[bool]`) – A mask where entries correspond to spin-orbitals. 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]` – Positional indices of `mask` elements with value `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 deprecated legacy method and will be removed in subsequent releases.

 **Notes**

If some value is `None`, then it is counted.

Parameters

- **fermion_state** (`FermionState`, default: `None`) – A `FermionState` object.
- **state_value** (`int`, default: `1`) – `1/0` signifying occupied/unoccupied, respectively.
- **column** (`int`, default: `None`) – The column index in the quantum number table.
- **column_value** (`int`, default: `None`) – The value to be counted.
- **state_mask** (`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 numpy `ndarray` 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.

The `window` defines how many elements in the result are `True`, with the `shift` determining the start index of the window. See below for example.

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 numpy `ndarray` 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])
```

`generate_fock_state_from_list`(*fock_state_list*=*None*)

Create a `FermionState` object corresponding to `fock_state_list`.

Parameters

- **fock_state_list** (`List[int]`, default: `None`) – A 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]`) – Occupation numbers $\in \{0, 1, 2\}$ for each spatial orbital.

Returns

`FermionState` – The generated fermionic state.

generate_fock_state_from_spatial_occupation(*occupation*)

Generates a FermionState from spatial orbital occupations of each cell in the supercell.

Parameters

- **occupation** (`List[List[int]]`) – Occupations, each a `list` with entries $\in \{0, 1, 2\}$ and length equal to 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 region.
- **multiplicity** (`int`, default: `None`) – Multiplicity of the state per region.

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]]`) – 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.

property n_spin_orb: int

Return the number of spin-orbitals.

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.

Parameters

- **swaps** (`List[Tuple[int, int]]`) – The 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(permuation)

Convert a permutation to a permutation matrix.

Parameters

- **permuation** (`Union[List[int], ndarray[Any, dtype[int]]]`) – 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.

Return type

`None`

❶ 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   .
```

`quantum_label(i)`

Get the quantum label corresponding to the space element indexed by the argument `i`.

Parameters

`i (int)` – Index of the space element.

Returns

`str` – The r-point, spatial orbital and spin quantum numbers as a formatted string.

`quantum_number(i)`

Compute the quantum numbers corresponding to the space element indexed by the argument `i`.

Parameters

`i (int)` – Index of the space element.

Returns

`Tuple[int, int, int]` – The regional block index, spatial orbital index and spin quantum number of the space element with index `i`.

`quantum_number_orb(i)`

Compute the orbital quantum number corresponding to the space element indexed by the argument `i`.

Parameters

`i (int)` – Index of the space element.

Returns

`int` – The orbital quantum number of the space element with index `i`.

`quantum_number_rp(i)`

Compute the regional block quantum number corresponding to the space element indexed by the argument `i`.

Parameters

`i (int)` – Index of the space element.

Returns

`int` – The regional block quantum number of the `i`th space element.

`quantum_number_spin(i)`

Compute the spin quantum number corresponding to the space element indexed by the argument `i`.

Parameters

- `i` (`int`) – Index of the space element.

Returns

- `int` – The spin quantum number of the space element with index `i`.

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`.

 **Warning**

This method is deprecated and will be removed in InQuanto v5.0.0

Parameters

- `fermion_state` (`FermionState`, default: `None`) – A `FermionState` object.
- `state_value` (default: `1`) – State value.
- `column` (`int`, default: `None`) – The column in the underlying table to select from, 0 for spin, 1 for orbital.
- `column_value` (`int`, default: `None`) – The value of interest for the given column.
- `state_mask` (`List[bool]`, default: `None`) – A boolean mask 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 ndarray 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 ndarray.

translate_operator(*operator*, *translation*=*None*)

Constructs a new operator that is translated by the cell translation function.

The default cell translation is $R \rightarrow R + 1$, i.e. it shifts to the next indices. In 1D this is a simple translation along the axis.

Parameters

- **operator** (`FermionOperator`) – Fermion operator.
- **translation** (default: `None`) – Cell translation function.

Returns

`FermionOperator` – The translated fermion operator.

class ParaFermionSpace(*n_spin_orb*, *point_group*=*None*, *orb_irreps*=*None*)

Bases: `QubitSpace`

Represents a parafermionic Hilbert space, for the purposes of simulating fermions.

This class is specifically intended for the use of simulating fermionic systems, rather than a general abstract parafermionic space. As such, it is constructed via the specification of the number of spin-orbitals in the simulated fermionic system. Optionally, information regarding the fermionic symmetries may be provided.

Unlike conventional mapping techniques, fermionic anticommutation relations are not encoded into the operators generated by this class. This may be useful in circumstances where the fermionic anticommutation relations are “already included” in other parts of a simulation – for instance, in the generation of chemical ansatz states.

Parameters

- **n_spin_orb** (`int`) – Number of spin orbitals.
- **point_group** (`Union[PointGroup, str]`, default: `None`) – Point group of the molecule of interest.
- **orb_irreps** (`List[str]`, default: `None`) – Point group irreducible representations of the orbitals.

construct_double_qubit_excitation_operators()

Generate 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

Jordan-Wigner encoding of Hamiltonian is assumed: state of qubit i represents occupation of spin orbital i .

Returns

QubitOperatorList – Every unique double qubit excitation.

construct_imag_pauli_exponent_operators()

Constructs a *QubitOperatorList* containing operators with even Pauli-Y count and an imaginary unit coefficient.

Each created *QubitOperator* in the *QubitOperatorList* is associated with a fresh coefficient symbol.

Returns

A *QubitOperatorList* containing *QubitOperator* objects with even Pauli-Y count and an imaginary unit coefficient.

static construct_operator_from_string(*input*)

Construct a *QubitOperator* from the string provided as input.

See *from_string()* for further details.

Parameters

input (*str*) – String representation of *QubitOperator*.

Returns

QubitOperator – The generated qubit operator.

construct_real_pauli_exponent_operators()

Constructs a *QubitOperatorList* containing operators with odd Pauli-Y count and a unit coefficient.

Each created *QubitOperator* in the *QubitOperatorList* is associated with a fresh coefficient symbol.

Returns

QubitOperatorList – A *QubitOperatorList* containing *QubitOperator* objects with an odd number of Pauli-Y operators and a unit coefficient.

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 , $T_{ik} = \frac{1}{2}([X_i Y_k] - [Y_i X_k])$. Fresh parameter symbols are generated for each operator.

 **Warning**

Jordan-Wigner encoding of Hamiltonian is assumed: state of qubit i represents occupation of spin orbital i .

Returns

QubitOperatorList – All unique single qubit excitation operators.

```
static count_spin(qstate)
```

Counts total spin of a QubitState assuming a Jordan-Wigner mapped state.

The value returned should equal $\langle \hat{S}_z \rangle$. 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.

```
property n_spin_orb: int
```

Returns the number of spin-orbitals.

```
property paulis: Set[str]
```

Returns a set of python strings {"I", "X", "Y", "Z"}.

```
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, formatted at the orbital quantum number followed by spin label (a or b).

```
quantum_number(i)
```

Converts the index to a tuple of quantum numbers.

Parameters

i (**int**) – Index of a given basis state.

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 of a given basis state.

Returns

`int` – Spatial orbital quantum number.

quantum_number_spin (i)

Get the spin quantum number from an index.

Parameters

`i (int)` – Index of a given basis state.

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 \mathbb{Z}_2 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 \mathbb{Z}_2 symmetries of.

Returns

`List[SymmetryOperatorPauli]` – Pauli representations of the \mathbb{Z}_2 symmetry operators.

static symmetry_operators_z2_in_sector (operator, state)

Constructs \mathbb{Z}_2 symmetry operators which stabilize a provided state.

An independent generating set of symmetry operators is found by finding the kernel of the symplectic form of the operator (arXiv 1701.08213). Operators which stabilize a desired symmetry sector are then returned.

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 \mathbb{Z}_2 symmetries of.
- `state (QubitState)` – A state with which the returned symmetry operators will have an eigenvalue of +1.

Returns

`List[SymmetryOperatorPauli]` – Pauli representations of the \mathbb{Z}_2 symmetry operators which stabilize the desired symmetry sectors.

```
class QubitSpace(n_qubits)
```

Bases: `object`

Represents a Hilbert space comprised of N qubits, described in a basis of Pauli vectors.

Parameters

`n_qubits` (`int`) – The number of qubits.

```
construct_imag_pauli_exponent_operators()
```

Constructs a `QubitOperatorList` containing operators with even Pauli- Y count and an imaginary unit coefficient.

Each created `QubitOperator` in the `QubitOperatorList` is associated with a fresh coefficient symbol.

Returns

A `QubitOperatorList` containing `QubitOperator` objects with even Pauli- Y count and an imaginary unit coefficient.

```
static construct_operator_from_string(input)
```

Construct a `QubitOperator` from the string provided as input.

See `from_string()` for further details.

Parameters

`input` (`str`) – String representation of `QubitOperator`.

Returns

`QubitOperator` – The generated qubit operator.

```
construct_real_pauli_exponent_operators()
```

Constructs a `QubitOperatorList` containing operators with odd Pauli- Y count and a unit coefficient.

Each created `QubitOperator` in the `QubitOperatorList` is associated with a fresh coefficient symbol.

Returns

`QubitOperatorList` – A `QubitOperatorList` containing `QubitOperator` objects with an odd number of Pauli- Y operators and a unit coefficient.

```
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" }.

```
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 \mathbb{Z}_2 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 \mathbb{Z}_2 symmetries of.

Returns

`List[SymmetryOperatorPauli]` – Pauli representations of the \mathbb{Z}_2 symmetry operators.

static symmetry_operators_z2_in_sector (operator, state)

Constructs \mathbb{Z}_2 symmetry operators which stabilize a provided state.

An independent generating set of symmetry operators is found by finding the kernel of the symplectic form of the operator (arXiv 1701.08213). Operators which stabilize a desired symmetry sector are then returned.

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 \mathbb{Z}_2 symmetries of.
- `state (QubitState)` – A state with which the returned symmetry operators will have an eigenvalue of +1.

Returns

`List[SymmetryOperatorPauli]` – Pauli representations of the \mathbb{Z}_2 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.

27.13 inquanto.states

InQuanto representation of quantum states.

class FermionState (data=None, coeff=1.0)

Bases: `State`

Represents a state in a fermionic Fock space as a linear combination of basis states.

The 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 `FermionStateString` and a coefficient, a `tuple` of `tuple` elements, each containing a coefficient and a `FermionStateString` object, or a `dict`, with `FermionStateString` objects as keys and coefficients as values. For a single basis state, it can also be initialised with a `list` of integers, wherein a default set of modes (indexed from 0 upwards) will be assumed. See below for examples.

Parameters

- `data (Union[List[int], StateString, Tuple[Tuple[Union[int, float, complex, Expr], StateString]], Dict[StateString, Union[int, float, complex, Expr]]], default: None)` – Information regarding the state to be created. A variety of input formats are parsed - see examples below for details.
- `coeff (Union[int, float, complex, Expr], default: 1.0)` – An optional additional coefficient, used when a single basis state is being created. See below for example.

Examples

```
>>> fs0 = FermionState([1, 1, 0, 0], 1)
>>> print(fs0)
(1, {0: 1, 1: 1, 2: 0, 3: 0})
>>> fss = FermionStateString({0:1, 1:1, 2:0, 3:0})
>>> fs1 = FermionState(fss, 1)
>>> print(fs1)
(1, {0: 1, 1: 1, 2: 0, 3: 0})
>>> fss0 = FermionStateString({0:1, 1:1, 2:0, 5:0})
>>> fss1 = FermionStateString({0:1, 1:0, 2:1, 5:0})
>>> fs2 = FermionState((0.9, fss0), (0.1, fss1)))
>>> print(fs2)
(0.9, {0: 1, 1: 1, 2: 0, 5: 0}), (0.1, {0: 1, 1: 0, 2: 1, 5: 0})
>>> fss0 = FermionStateString({0:1, 1:1, 2:0, 5:0})
>>> fss1 = FermionStateString({0:1, 1:0, 2:1, 5:0})
>>> fs3 = FermionState({fss0: 0.9, fss1: 0.1})
>>> print(fs3)
(0.9, {0: 1, 1: 1, 2: 0, 5: 0}), (0.1, {0: 1, 1: 0, 2: 1, 5: 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` – 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 against which the norm of the difference is checked.

Return type

`bool`

`property basis_states: List[StateStringT]`

Returns the basis states with nonzero coefficient as a `list`.

clone()

Performs shallow copy of the object.

Return type

`TypeVar(SYMBOLICTYPE, bound= Symbolic)`

property coefficients: List[int | float | complex | Expr]

Returns the scalar coefficients of individual basis vectors as a `list`.

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 `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 ansatzes and digital quantum simulation. Otherwise, it may remove terms relevant to computation. Each expression is of the form $f(a_1, a_2, \dots, a_n)$ for some symbols a_i . $|f(a_1, a_2, \dots, a_n)|$ 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(state_vector, modes=None, reverse=False)

Returns an instance representing the provided `ndarray` state vector.

This method will take a state vector as a `ndarray` column vector and convert it to a `State` object. Modes - and the bit position in the state index to which each corresponds - can be defined with the `modes` parameter. In the absence of this parameter, a default register of the appropriate size is implicit, with the index being assumed to correspond to an ILO-BE ordering of the default register.

In keeping with numpy convention, 1-D arrays and 2-D arrays with shape $(N, 1)$ are treated as column vectors. The length N of the vector must be either 0 or a power of 2. Differently shaped arrays are not accepted.

Parameters

- **state_vector** (`ndarray`) – The state vector representing the state to be created.
- **modes** ([Optional\[Iterable\[TypeVar\(ModeT\)\]\]](#), default: `None`) – An ordering of modes used for defining indices, from most significant to least significant. Default behaviour as above.
- **reverse** (default: `False`) – If `True`, reverse bit order of state vector indices (e.g. for changing endianness).

Returns

An instance of this class representing the provided `ndarray` state vector.

Raises

`ValueError` – If the input state vector has invalid shape.

classmethod from_sparray(state_vector, modes=None, reverse=False)

Returns an instance representing the provided `scipy sparse array` `state_vector`.

This method will take a state vector as a `scipy sparse array` and convert it to a `State` object. Modes - and the bit position in the state index to which each corresponds - can be defined with the `modes` parameter. In the absence of this parameter, a default register of the appropriate size is implicit, with the index being assumed to correspond to an ILO-BE ordering of the default register.

The shape of the array must be $(N, 1)$ where N is either 0 or a power of 2. Differently shaped arrays are not accepted.

Parameters

- **state_vector** (`spmatrix`) – The state vector representing the state to be created.
- **modes** ([Optional\[Iterable\[TypeVar\(ModeT\)\]\]](#), default: `None`) – An ordering of modes used for defining indices, from most significant to least significant. Default behaviour as above.
- **reverse** (default: `False`) – If `True`, reverse bit order of state vector indices (e.g. for changing endianness).

Returns

An instance of this class representing `state_vector`.

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

`TypeVar(StateT, bound= State)` – Child class object.

get_numeric_representation()

Placeholder, unimplemented method.

get_symbolic_representation()

Placeholder, unimplemented method.

is_all_coeff_complex()

Check if all coefficients have complex values.

Returns

`bool` – `False` if a non-complex value occurs before any free symbols in the `dict` values, or `True` if no non-complex values occur.

 **Warning**

Returns `None` if a free symbol occurs before any non-complex values in the coefficients.

is_all_coeff_imag()

Check if all coefficients have purely imaginary values.

Returns

`bool` – `False` if a non-complex value occurs before any free symbols in the `dict` values, or `True` if no non-complex values occur.

 **Warning**

Returns `None` if a free symbol occurs before any non-imaginary values in the coefficients.

is_all_coeff_real()

Check if all coefficients have real values.

Returns

`bool` – `False` if a non-real value occurs before any free symbols in the `dict` values, or `True` if no non-real values occur.

 **Warning**

Returns `None` if a free symbol occurs before any non-real values in the `dict` coefficients.

`is_all_coeff_symbolic()`

Check if all coefficients contain free symbols.

Returns

`bool` – Whether all coefficients contain free symbols.

`is_any_coeff_complex()`

Check if any coefficients have complex values.

Returns

`bool` – True if a complex value occurs before any free symbols in the `dict` values, or `False` if no complex values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any complex values in the coefficients.

`is_any_coeff_imag()`

Check if any coefficients have imaginary values.

Returns

`bool` – True if an imaginary value occurs before any free symbols in the `dict` values, or `False` if no imaginary values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any imaginary values in the coefficients.

`is_any_coeff_real()`

Check if any coefficients have real values.

Returns

`bool` – True if a real value occurs before any free symbols in the `dict` values, or `False` if no real values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any real values in the coefficients.

`is_any_coeff_symbolic()`

Check if any coefficients contain free symbols.

Returns

`bool` – Whether any coefficients contain free symbols.

`is_basis_state()`

Checks this object represents a scalar multiple of a single basis state.

Return type

`bool`

`is_normalized(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_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]]`

```
classmethod key_from_str(key_str)
    Converts a string to an instance of this class's basis state type.

    Parameters
        key_str (str)

    Return type
        TypeVar(StateStringT, bound= StateString)

classmethod load_h5(name)
    Loads a state object from .h5 file.

    Parameters
        name (Union[str, Group]) – Name of .h5 file to be loaded.

    Returns
        TypeVar(StateT, bound= State) – Loaded integral operator object.

make_hashable()
    Return a hashable representation of the object.

    Returns
        str – A string representation of this instance.

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 – This instance.

property n_symbols: int
    Returns the number of free symbols in the object.

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.

property num_modes: int
    Returns the number of modes.
```

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 *FermionState*.

Terms are treated and mapped independently.

Parameters

- **mapping** (*Optional[QubitMapping]*, default: None) – Mapping class. Default mapping procedure is the Jordan-Wigner transformation.
- **qubits** (*Union[list[Qubit], int, None]*, 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

QubitState – Mapped *QubitState*.

remove_global_phase()

Returns a copy with a global phase applied such that the first element has a real coefficient.

This is an alias for *set_global_phase()* - see this method for greater control over the phase to be applied.

Returns

LinearDictCombiner – A copy of the object with a global phase applied such that the first element has a real coefficient.

reversed_order()

Reverses the order of terms and returns it as a new object.

Return type

LinearDictCombiner

save_h5 (name)

Dumps state object to .h5 file.

Parameters

name (*Union[str, Group]*) – Destination filename of .h5 file.

Return type

None

set_global_phase (phase=0.0)

Returns a copy with a global phase applied such that the first element has the desired phase.

Parameters

- **phase** (*Union[int, float, complex, Expr]*, default: 0.0) – The phase to yield on the first element, in half-turns (i.e. multiples of pi).
- **phase.** (*A symbolic expression can be assigned to*)

Returns

LinearDictCombiner – A copy of the object with the desired global phase applied.

simpify(*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: [StateStringT](#)

Returns the stored configuration as a `StateString`, if only a single configuration is stored.

Raises

`RuntimeError` – if more or less than one configuration is stored

Returns

The single configuration occupation vector.

split()

Generates `State` objects containing only single configurations, preserving coefficients.

Return type

`Generator[TypeVar(StateT, bound= State), None, None]`

string_class

alias of `FermionStateString`

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]]`) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

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` – Sympification fails.

property terms: List[Any]

Returns dictionary keys.

to_ndarray(modes=None, dtype=complex, reverse=False)

Returns a state vector representation of this state.

Modes - and the bit position in the state index to which each corresponds - can be defined with the `modes` parameter. In the absence of this parameter, a default register of the appropriate size is implicit, with the index being assumed to correspond to an ILO-BE ordering of the default register. See below for examples.

Parameters

- `modes` (`Optional[List[TypeVar(ModeT)]]`, default: `None`) – An ordering of modes used for defining indices, from most significant to least significant. Default behaviour as above.
- `dtype` (default: `complex`) – The numpy `dtype` to be used for the returned vector.
- `reverse` (`bool`, default: `False`) – If `True`, reverse bit order of state vector indices (e.g. for changing endianness).

Returns

`ndarray` – A dense numpy `ndarray` representation of the state.

Examples

```
>>> qss0 = StateString({0:1,1:1,2:0,3:0})
>>> qss1 = StateString({0:1,1:0,2:1,3:0})
>>> qs = State({qss0: 0.9, qss1: 0.1})
>>> state_vector = qs.to_ndarray()
>>> print(state_vector)
[[0. +0.j]
 [0. +0.j]]
>>> state_vector = qs.to_ndarray([0,2,3,1])
>>> 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]
[0.9+0.j]
[0. +0.j]
[0. +0.j]
[0.1+0.j]
[0. +0.j]
[0. +0.j]
[0. +0.j]]
```

`to_sparray`(*modes=None*, *matrix_type=scipy.sparse.csr_array*, *dtype=complex*, *reverse=False*)

Returns a scipy sparse array state vector representation of this state.

Modes - and the bit position in the state index to which each corresponds - can be defined with the `modes` parameter. In the absence of this parameter, a default register of the appropriate size is implicit, with the index being assumed to correspond to an ILO-BE ordering of the default register. See [to_ndarray\(\)](#) for examples.

Parameters

- `modes` (`Optional[List[TypeVar(ModeT)]]`, default: `None`) – An ordering of modes used for defining indices, from most significant to least significant. Default behaviour as above.
- `matrix_type` (default: `scipy.sparse.csr_array`) – The type of the sparse matrix to be generated.
- `dtype` (default: `complex`) – The numpy `dtype` to be used for the returned vector.
- `reverse` (`bool`, default: `False`) – If `True`, reverse bit order of state vector indices (e.g. for changing endianness).

Returns

A scipy sparse array representation of the state.

`unsympify`(*precision=15*, *partial=False*)

Unsympifies dictionary values.

Replaces symbolic expressions with their corresponding numeric values.

Parameters

- `precision` (`int`, default: 15) – The number of decimal digits of precision used for evaluation.
- `partial` (`bool`, default: `False`) – Set to `True` to allow partial unsympification where terms containing free symbols are present. By default, free symbols in any coefficient will cause an exception.

Returns

`LinearDictCombiner` – Updated instance of `LinearDictCombiner`.

Raises

`TypeError` – Unsympification fails.

vdot (other)

Calculates the inner product with another `State` object.

The other `State` is treated as the ket, this one is treated as the bra - i.e. the coefficient

Parameters

`other` (`TypeVar(StateT, bound= State)`) – Another `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.

i Examples

```
>>> print(LinearDictCombiner.zero())
()
```

class FermionStateString (data)

Bases: `StateString[int]`

Represents a single basis state of a fermionic Fock space.

Fermionic modes are represented as integers, which serve as a numerical label for each mode/spin-orbital. This class does not contain capacity for storing a scalar coefficient associated with the basis state (i.e. it represents a ray, not a vector) – see `FermionState` for an arbitrary state in the fermionic state space. Note that instances of this class are immutable after creation.

Parameters

`data` (`Union[Dict[TypeVar(ModeT), Union[int, bool]], List[Union[int, bool]], Tuple[Union[int, bool]]]`) – The basis state to be represented. Generally this will be in the form of a dictionary mapping mode indices to occupation numbers, as per the example below. For backwards compatibility, this can additionally be instantiated using a list of integer occupation numbers, for which a default set of modes will be assumed.

Raises

`ValueError` – If provided occupation numbers are not 0 or 1.

i Examples

```
>>> fss = FermionStateString({0:1, 1:1, 2:0, 5:0})
>>> print(fss)
{0: 1, 1: 1, 2: 0, 5: 0}
```

property all_modes: FrozenSet

Returns all mode objects in the space of which this `StateString` is a basis state.

classmethod from_index (state_index, modes=None, reverse=False)

Creates an instance from the index of a state vector corresponding to a single basis state.

Modes - and the bit position in the state index to which each corresponds - can be defined with the `modes` parameter. In the absence of this parameter, a default register of the appropriate size is implicit, with the index being assumed to correspond to an ILO-BE ordering of the default register. See below for examples.

Parameters

- **state_index** (`int`) – The index of a state vector corresponding to the basis state to which the created object will correspond.
- **modes** (`Optional[Iterable[TypeVar(ModeT)]]`, default: `None`) – A collection of modes, the state values of which are represented by the state vector, in the appropriate ordering.
- **reverse** (default: `False`) – If `True`, reverse the bits of the index.

Returns

`TypeVar(StateStringT, bound= StateString)` – An instance of this class corresponding to the basis state to which the provided index corresponds.

Examples

```
>>> index = 12
>>> statestring = StateString.from_index(index)
>>> print(statestring)
{0: 1, 1: 1, 2: 0, 3: 0}
>>> statestring = StateString.from_index(index, modes=[3, 0, 1, 2])
>>> print(statestring)
{3: 1, 0: 1, 1: 0, 2: 0}
```

`classmethod from_list_int(data)`

Construct a `StateString` from a `list` of integers or booleans.

The integers within the `list` should correspond to the states of individual modes. Modes will be assumed to be indexed in the `[0, len(data))` range in ascending order.

Parameters

`data (List[Union[int, bool]])` – State values to assign for the default mode set.

Returns

`TypeVar(StateStringT, bound= StateString)` – An instance of this class representing the basis state with provided state values, using a default set of modes.

`classmethod from_string(data)`

Generate a class instance from a string.

This method will make a best effort to interpret a string as an instance of this class. It is provided largely for backwards compatibility. Caution is advised.

Parameters

`data (str)` – A string from which to attempt to generate an instance of this class.

Returns

`TypeVar(StateStringT, bound= StateString)` – A class instance generated from the input string.

`property hamming_weight: int`

Returns the Hamming weight of the basis state.

`classmethod load_h5(name)`

Loads a state string object from .h5 file.

Parameters

`name (Union[str, Group])` – Name of .h5 file to be loaded.

Returns

`TypeVar(StateStringT, bound= StateString)` – Loaded integral operator object.

mode_class

alias of `int`

property num_modes: int

Returns the number of modes in the space of which this `StateString` is a basis state.

occupations_ordered()

Returns a `list` of integer or boolean occupation numbers corresponding to modes ordered in increasing lexicographic order.

Return type

`List[Union[int, bool]]`

qubit_encode(mapping=None, qubits=None)

Performs qubit encoding (mapping), using provided mapping function, of the current `FermionStateString`.

Terms are treated and mapped independently.

Parameters

- **mapping** (`Optional[QubitMapping]`, default: `None`) – Mapping class. Default mapping procedure is the Jordan-Wigner transformation.
- **qubits** (`Union[list[Qubit], int, None]`, 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

`QubitStateString` – Mapped `QubitStateString`.

save_h5(name)

Dumps state string object to .h5 file.

Parameters

`name (Union[str, Group])` – Destination filename of .h5 file.

Return type

`None`

to_index(modes=None, reverse=False)

Returns the index of the state vector to which this basis state corresponds.

The ordering of modes when defining indexing into the state vector can be controlled with the `modes` parameter. By default, `modes` will be ordered following an ILO-BE convention - see examples.

Parameters

- **modes** (`Optional[Iterable[TypeVar(ModeT)]]`, default: `None`) – An ordering of modes used for defining state vector indices, from most significant to least significant.
- **reverse** (default: `False`) – If `True`, reverse the order of bits in the state vector index.

Returns

`int` – A state vector index corresponding to this basis state.

Raises

`ValueError` – If provided modes are not the same as those of the state.

❶ Examples

```
>>> statestring = StateString({0:1,1:1,2:0,3:0})
>>> index = statestring.to_index()
>>> print(index)
12
>>> index = statestring.to_index([0,2,3,1])
>>> print(index)
9
```

class QubitState(*data=None*, *coeff=1.0*)

Bases: *State*, *Representable*

Represents the state of a register of qubits as a linear combination of computational basis states.

The state is stored as a dictionary, with the keys being *QubitStateString* objects and values being the corresponding configuration coefficients. The *QubitState* may be initialised from a *QubitStateString* and a coefficient, a tuple of tuple elements, each containing a coefficient and a *QubitStateString* object, or a dict, with *QubitStateString* objects as keys and coefficients as values. For a single basis state, it can also be initialised with a list of integers, wherein a default set of qubits (indexed from 0 upwards) will be assumed. See below for examples.

Parameters

- **data** (`Union[List[int], StateString, Tuple[Tuple[Union[int, float, complex, Expr], StateString]], Dict[StateString, Union[int, float, complex, Expr]]]`, default: `None`) – Information regarding the state to be created. A variety of input formats are parsed - see examples below for details.
- **coeff** (`Union[int, float, complex, Expr]`, default: `1.0`) – An optional additional coefficient, used when a single basis state is being created. See below for example.

❶ Examples

```
>>> qs0 = QubitState([1, 1, 0, 0], 1)
>>> print(qs0)
(1, {q[0]: 1, q[1]: 1, q[2]: 0, q[3]: 0})
>>> qss = QubitStateString({Qubit(0):1,Qubit(1):1,Qubit(2):0,Qubit(5):0})
>>> qs1 = QubitState(qss, 1)
>>> print(qs1)
(1, {q[0]: 1, q[1]: 1, q[2]: 0, q[5]: 0})
>>> qss0 = QubitStateString({Qubit(0):1,Qubit(1):1,Qubit(2):0,Qubit(5):0})
>>> qss1 = QubitStateString({Qubit(0):1,Qubit(1):0,Qubit(2):1,Qubit(5):0})
>>> qs2 = QubitState((0.9, qss0), (0.1, qss1))
>>> print(qs2)
(0.9, {q[0]: 1, q[1]: 1, q[2]: 0, q[5]: 0}), (0.1, {q[0]: 1, q[1]: 0, q[2]: 1, ↴q[5]: 0})
>>> qss0 = QubitStateString({Qubit(0):1,Qubit(1):1,Qubit(2):0,Qubit(5):0})
>>> qss1 = QubitStateString({Qubit(0):1,Qubit(1):0,Qubit(2):1,Qubit(5):0})
>>> qs3 = QubitState({qss0: 0.9, qss1: 0.1})
>>> print(qs3)
(0.9, {q[0]: 1, q[1]: 1, q[2]: 0, q[5]: 0}), (0.1, {q[0]: 1, q[1]: 0, q[2]: 1, ↴q[5]: 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` – Comparison of two values can't be done due to types mismatch.

Return type

`bool`

`approx_equal_to_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 against which the norm of the difference is checked.

Return type

`bool`

`property basis_states: List[StateStringT]`

Returns the basis states with nonzero coefficient as a `list`.

`clone()`

Performs shallow copy of the object.

Return type

`TypeVar(SYMBOLICTYPE, bound=Symbolic)`

`property coefficients: List[int | float | complex | Expr]`

Returns the scalar coefficients of individual basis vectors as a `list`.

`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 `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 ansatzes and digital quantum simulation. Otherwise, it may remove terms relevant to computation. Each expression is of the form $f(a_1, a_2, \dots, a_n)$ for some symbols a_i . $|f(a_1, a_2, \dots, a_n)|$ 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(state_vector, modes=None, reverse=False)`

Returns an instance representing the provided ndarray state vector.

This method will take a state vector as a `ndarray` column vector and convert it to a `State` object. Modes - and the bit position in the state index to which each corresponds - can be defined with the `modes` parameter. In the absence of this parameter, a default register of the appropriate size is implicit, with the index being assumed to correspond to an ILO-BE ordering of the default register.

In keeping with numpy convention, 1-D arrays and 2-D arrays with shape $(N, 1)$ are treated as column vectors. The length N of the vector must be either 0 or a power of 2. Differently shaped arrays are not accepted.

Parameters

- **state_vector** (ndarray) – The state vector representing the state to be created.
- **modes** (`Optional[Iterable[TypeVar(ModeT)]]`, default: `None`) – An ordering of modes used for defining indices, from most significant to least significant. Default behaviour as above.
- **reverse** (default: `False`) – If `True`, reverse bit order of state vector indices (e.g. for changing endianness).

Returns

An instance of this class representing the provided `ndarray` state vector.

Raises

`ValueError` – If the input state vector has invalid shape.

`classmethod from_sparray(state_vector, modes=None, reverse=False)`

Returns an instance representing the provided scipy sparse array `state_vector`.

This method will take a state vector as a scipy sparse array and convert it to a `State` object. Modes - and the bit position in the state index to which each corresponds - can be defined with the `modes` parameter. In the absence of this parameter, a default register of the appropriate size is implicit, with the index being assumed to correspond to an ILO-BE ordering of the default register.

The shape of the array must be $(N, 1)$ where N is either 0 or a power of 2. Differently shaped arrays are not accepted.

Parameters

- **state_vector** (spmatrix) – The state vector representing the state to be created.
- **modes** (`Optional[Iterable[TypeVar(ModeT)]]`, default: `None`) – An ordering of modes used for defining indices, from most significant to least significant. Default behaviour as above.
- **reverse** (default: `False`) – If `True`, reverse bit order of state vector indices (e.g. for changing endianness).

Returns

An instance of this class representing `state_vector`.

`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

`TypeVar(StateT, bound= State)` – Child class object.

`get_numeric_representation(symbol_map=None, dtype=complex, backend=None)`

Returns a state vector representation of this state, with an optional symbol substitution map.

This proceeds by substituting symbols for numeric values according to the provided map, before converting to a `ndarray` using the `to_ndarray()` method. Qubits are ordered according to an ILO-BE convention. See the documentation of the `to_ndarray()` method for further details.

Danger

This method requires exponential resources.

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 applied before constructing 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 the numpy dtype to which the return array should be converted.
- **backend** (`None`, default: `None`) – Unused compatibility argument.

Returns

`Union[_SupportsArray[dtype[Any]], _NestedSequence[_SupportsArray[dtype[Any]]], bool, int, float, complex, str, bytes, _NestedSequence[Union[bool, int, float, complex, str, bytes]]]` – A vector representing the object.

get_symbolic_representation(`symbol_map=None`)

Constructs a sympy symbolic statevector representation.

Qubits will be ordered according to an ILO-BE convention. See `to_index()` for further details.

Danger

This method requires exponential resources.

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, to replace symbolic coefficients with numerical ones.

Returns

`Expr` – A symbolic statevector representation.

is_all_coeff_complex()

Check if all coefficients have complex values.

Returns

`bool` – `False` if a non-complex value occurs before any free symbols in the `dict` values, or `True` if no non-complex values occur.

Warning

Returns `None` if a free symbol occurs before any non-complex values in the coefficients.

`is_all_coeff_imag()`

Check if all coefficients have purely imaginary values.

Returns

`bool` – `False` if a non-complex value occurs before any free symbols in the `dict` values, or `True` if no non-complex values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any non-imaginary values in the coefficients.

`is_all_coeff_real()`

Check if all coefficients have real values.

Returns

`bool` – `False` if a non-real value occurs before any free symbols in the `dict` values, or `True` if no non-real values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any non-real values in the `dict` coefficients.

`is_all_coeff_symbolic()`

Check if all coefficients contain free symbols.

Returns

`bool` – Whether all coefficients contain free symbols.

`is_any_coeff_complex()`

Check if any coefficients have complex values.

Returns

`bool` – `True` if a complex value occurs before any free symbols in the `dict` values, or `False` if no complex values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any complex values in the coefficients.

`is_any_coeff_imag()`

Check if any coefficients have imaginary values.

Returns

`bool` – `True` if an imaginary value occurs before any free symbols in the `dict` values, or `False` if no imaginary values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any imaginary values in the coefficients.

`is_any_coeff_real()`

Check if any coefficients have real values.

Returns

`bool` – True if a real value occurs before any free symbols in the `dict` values, or `False` if no real values occur.

 **Warning**

Returns `None` if a free symbol occurs before any real values in the coefficients.

`is_any_coeff_symbolic()`

Check if any coefficients contain free symbols.

Returns

`bool` – Whether any coefficients contain free symbols.

`is_basis_state()`

Checks this object represents a scalar multiple of a single basis state.

Return type

`bool`

`is_normalized(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_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]]`

classmethod key_from_str(*key_str*)

Converts a string to an instance of this class's basis state type.

Parameters

key_str (`str`)

Return type

`TypeVar(StateStringT, bound= StateString)`

classmethod load_h5(*name*)

Loads a state object from .h5 file.

Parameters

name (`Union[str, Group]`) – Name of .h5 file to be loaded.

Returns

`TypeVar(StateT, bound= State)` – Loaded integral operator object.

make_hashable()

Return a hashable representation of the object.

Returns

`str` – A string representation of this instance.

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` – This instance.

property n_qubits: int

Returns number of qubits in state space.

property n_symbols: int

Returns the number of free symbols in the object.

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.

property num_modes: int

Returns the number of modes.

property num_qubits: int

Returns number of qubits in state space.

print_table()

Print dictionary formatted as a table.

Return type

`NoReturn`

remove_global_phase()

Returns a copy with a global phase applied such that the first element has a real coefficient.

This is an alias for `set_global_phase()` - see this method for greater control over the phase to be applied.

Returns

`LinearDictCombiner` – A copy of the object with a global phase applied such that the first element has a real coefficient.

reversed_order()

Reverses the order of terms and returns it as a new object.

Return type

`LinearDictCombiner`

save_h5 (name)

Dumps state object to .h5 file.

Parameters

name (`Union[str, Group]`) – Destination filename of .h5 file.

Return type`None`**`set_global_phase` (*phase*=*0.0*)**

Returns a copy with a global phase applied such that the first element has the desired phase.

Parameters

- **`phase`** (`Union[int, float, complex, Expr]`, default: `0.0`) – The phase to yield on the first element, in half-turns (i.e. multiples of π).
- **`phase.`** (A symbolic expression can be assigned to)

Returns`LinearDictCombiner` – A copy of the object with the desired global phase applied.**`simplify` (**args*, ***kargs*)**

Simplifies expressions stored in dictionary values.

Parameters

- **`args`** (`Any`) – Args to be passed to `sympy.simplify()`.
- **`kargs`** (`Any`) – Kwargs to be passed to `sympy.simplify()`.

Returns`LinearDictCombiner` – Updated instance of `LinearDictCombiner`.**`property single_term: StateStringT`**

Returns the stored configuration as a `StateString`, if only a single configuration is stored.

Raises`RuntimeError` – if more or less than one configuration is stored**Returns**

The single configuration occupation vector.

`split()`

Generates `State` objects containing only single configurations, preserving coefficients.

Return type`Generator[TypeVar(StateT, bound= State), None, None]`**`property state_symbols`**

Returns the free symbols within the coefficients of the state.

`string_class`

alias of `QubitStateString`

`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]`) – A mapping for substitution of free symbols.

Returns`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

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 `sympify()`.
- **kwargs** (`Any`) – Kwargs to be passed to `sympify()`.

Returns

`LinearDictCombiner` – Updated instance of `LinearDictCombiner`.

Raises

`RuntimeError` – Sympification fails.

property terms: List [Any]

Returns dictionary keys.

to_ndarray (modes=None, dtype=complex, reverse=False)

Returns a state vector representation of this state.

Modes - and the bit position in the state index to which each corresponds - can be defined with the `modes` parameter. In the absence of this parameter, a default register of the appropriate size is implicit, with the index being assumed to correspond to an ILO-BE ordering of the default register. See below for examples.

Parameters

- **modes** (`Optional[List[TypeVar(ModeT)]]`, default: `None`) – An ordering of modes used for defining indices, from most significant to least significant. Default behaviour as above.
- **dtype** (default: `complex`) – The numpy `dtype` to be used for the returned vector.
- **reverse** (`bool`, default: `False`) – If `True`, reverse bit order of state vector indices (e.g. for changing endianness).

Returns

`ndarray` – A dense numpy `ndarray` representation of the state.

Examples

```
>>> qss0 = StateString({0:1,1:1,2:0,3:0})
>>> qss1 = StateString({0:1,1:0,2:1,3:0})
>>> qs = State({qss0: 0.9, qss1: 0.1})
>>> state_vector = qs.to_ndarray()
>>> print(state_vector)
[[0. +0.j]
 [0. +0.j]
 [0. +0.j]]
```

```
[0. +0.j]
[0.1+0.j]
[0. +0.j]
[0.9+0.j]
[0. +0.j]
[0. +0.j]
[0. +0.j]
>>> state_vector = qs.to_ndarray([0,2,3,1])
>>> print(state_vector)
[[0. +0.j]
 [0. +0.j]]
```

`to_sparray`(*modes=None*, *matrix_type=scipy.sparse.csr_array*, *dtype=complex*, *reverse=False*)

Returns a scipy sparse array state vector representation of this state.

Modes - and the bit position in the state index to which each corresponds - can be defined with the `modes` parameter. In the absence of this parameter, a default register of the appropriate size is implicit, with the index being assumed to correspond to an ILO-BE ordering of the default register. See [to_ndarray\(\)](#) for examples.

Parameters

- **`modes`** (`Optional[List[TypeVar(ModeT)]]`, default: `None`) – An ordering of modes used for defining indices, from most significant to least significant. Default behaviour as above.
- **`matrix_type`** (default: `scipy.sparse.csr_array`) – The type of the sparse matrix to be generated.
- **`dtype`** (default: `complex`) – The numpy `dtype` to be used for the returned vector.
- **`reverse`** (`bool`, default: `False`) – If `True`, reverse bit order of state vector indices (e.g. for changing endianness).

Returns

A scipy sparse array representation of the state.

unsympify (*precision*=15, *partial*=False)

Unsympifies dictionary values.

Replaces symbolic expressions with their corresponding numeric values.

Parameters

- **precision** (`int`, default: 15) – The number of decimal digits of precision used for evaluation.
- **partial** (`bool`, default: `False`) – Set to True to allow partial unsympification where terms containing free symbols are present. By default, free symbols in any coefficient will cause an exception.

Returns

`LinearDictCombiner` – Updated instance of `LinearDictCombiner`.

Raises

`TypeError` – Unsympification fails.

vdot (*other*)

Calculates the inner product with another `State` object.

The other `State` is treated as the ket, this one is treated as the bra - i.e. the coefficient

Parameters

- **other** (`TypeVar(StateT, bound= State)`) – Another `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())
()
```

class QubitStateString(*data*)

Bases: `StateString`

Represents a computational basis state of a register of qubits.

Qubits are represented by `Qubit` objects. This class does not contain capacity for storing a scalar coefficient associated with the basis state (i.e. it represents a ray, not a vector) – see `QubitState` for an arbitrary state in the qubit state space. Note that instances of this class are immutable after creation.

Parameters

- **data** (`Union[Dict[TypeVar(ModeT), Union[int, bool]], List[Union[int, bool]], Tuple[Union[int, bool]]]`) – The basis state to be represented. Generally this will be in the form of a dictionary mapping qubits to 0 or 1 states, as per the example below. For backwards compatibility, this can additionally be instantiated using a list of integer occupation numbers, for which a default set of modes will be assumed.

Raises

`ValueError` – If provided state values are not 0 or 1.

Examples

```
>>> qss = QubitStateString({Qubit(0):1, Qubit(1):1, Qubit(2):0, Qubit(5):0})
>>> print(qss)
{q[0]: 1, q[1]: 1, q[2]: 0, q[5]: 0}
```

property all_modes: FrozenSet

Returns all mode objects in the space of which this *StateString* is a basis state.

classmethod from_index(state_index, modes=None, reverse=False)

Creates an instance from the index of a state vector corresponding to a single basis state.

Modes - and the bit position in the state index to which each corresponds - can be defined with the *modes* parameter. In the absence of this parameter, a default register of the appropriate size is implicit, with the index being assumed to correspond to an ILO-BE ordering of the default register. See below for examples.

Parameters

- **state_index (int)** – The index of a state vector corresponding to the basis state to which the created object will correspond.
- **modes (Optional[Iterable[TypeVar(ModeT)]]), default: None** – A collection of modes, the state values of which are represented by the state vector, in the appropriate ordering.
- **reverse (default: False)** – If True, reverse the bits of the index.

Returns

TypeVar(StateStringT, bound= StateString) – An instance of this class corresponding to the basis state to which the provided index corresponds.

Examples

```
>>> index = 12
>>> statestring = StateString.from_index(index)
>>> print(statestring)
{0: 1, 1: 1, 2: 0, 3: 0}
>>> statestring = StateString.from_index(index, modes=[3, 0, 1, 2])
>>> print(statestring)
{3: 1, 0: 1, 1: 0, 2: 0}
```

classmethod from_list_int(data)

Construct a *StateString* from a *list* of integers or booleans.

The integers within the *list* should correspond to the states of individual modes. Modes will be assumed to be indexed in the [0, *len*(*data*)) range in ascending order.

Parameters

data (List[Union[int, bool]]) – State values to assign for the default mode set.

Returns

TypeVar(StateStringT, bound= StateString) – An instance of this class representing the basis state with provided state values, using a default set of modes.

classmethod from_string(data)

Constructs a class instance from a string.

This method will make a best effort to interpret a string as an instance of this class. It is provided largely for backwards compatibility. Caution is advised.

Parameters

- **input_string** – String in the format `coeff1 [(coeff1_1, term1_1), ..., (coeff1_n, term1_n)], ..., coeffn [(coeffn_1, termn_1), ...]`.
- **data (str)**

Returns

`QubitStateString` – Child class object.

property hamming_weight: int

Returns the Hamming weight of the basis state.

classmethod load_h5(name)

Loads a state string object from .h5 file.

Parameters

`name (Union[str, Group])` – Name of .h5 file to be loaded.

Returns

`TypeVar(StateStringT, bound= StateString)` – Loaded integral operator object.

mode_class

alias of `Qubit`

property num_modes: int

Returns the number of modes in the space of which this `StateString` is a basis state.

occupations_ordered()

Returns a `list` of integer or boolean occupation numbers corresponding to modes ordered in increasing lexicographic order.

Return type

`List[Union[int, bool]]`

save_h5(name)

Dumps state string object to .h5 file.

Parameters

`name (Union[str, Group])` – Destination filename of .h5 file.

Return type

`None`

to_index(modes=None, reverse=False)

Returns the index of the state vector to which this basis state corresponds.

The ordering of modes when defining indexing into the state vector can be controlled with the `modes` parameter. By default, `modes` will be ordered following an ILO-BE convention - see examples.

Parameters

- **modes (Optional[Iterable[TypeVar(ModeT)]]), default: None** – An ordering of modes used for defining state vector indices, from most significant to least significant.
- **reverse (default: False)** – If `True`, reverse the order of bits in the state vector index.

Returns

`int` – A state vector index corresponding to this basis state.

Raises

`ValueError` – If provided modes are not the same as those of the state.

Examples

```
>>> statestring = StateString({0:1,1:1,2:0,3:0})
>>> index = statestring.to_index()
>>> print(index)
12
>>> index = statestring.to_index([0,2,3,1])
>>> print(index)
9
```

class State(data=None, coeff=1.0)

Bases: `LinearDictCombiner`, `Representable`, `Generic[StateStringT]`

A generic class for representing a state vector within some arbitrary Hilbert space.

This class is generic with regard to the nature of the underlying modes. Typically, one should use subclasses which represent basis vectors within a specific space - see the documentation for `QubitState` and `FermionState`.

Within this class, `StateString`-derived objects are mapped in a `dict`-like fashion to scalar coefficients, in order to represent a linear combination of basis states. Subclasses must define the `string_class` attribute, which gives the type (derived from `StateString`) of the object representing an individual basis state.

Examples

```
>>> qs0 = State([1, 1, 0, 0], 1)
>>> print(qs0)
(1, {0: 1, 1: 1, 2: 0, 3: 0})
>>> qss = StateString({0:1,1:1,2:0,3:0})
>>> qs1 = State(qss, 1)
>>> print(qs1)
(1, {0: 1, 1: 1, 2: 0, 3: 0})
>>> qss0 = StateString({0:1,1:1,2:0,3:0})
>>> qss1 = StateString({0:1,1:0,2:1,3:0})
>>> qs2 = State(((0.9, qss0), (0.1, qss1)))
>>> print(qs2)
(0.9, {0: 1, 1: 1, 2: 0, 3: 0}), (0.1, {0: 1, 1: 0, 2: 1, 3: 0})
>>> qss0 = StateString({0:1,1:1,2:0,3:0})
>>> qss1 = StateString({0:1,1:0,2:1,3:0})
>>> qs3 = State({qss0: 0.9, qss1: 0.1})
>>> print(qs3)
(0.9, {0: 1, 1: 1, 2: 0, 3: 0}), (0.1, {0: 1, 1: 0, 2: 1, 3: 0})
```

Parameters

- `data` (`Union[List[int], StateString, Tuple[Tuple[Union[int, float, complex, Expr], StateString]], Dict[StateString, Union[int, float, complex, Expr]]]`, 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` – 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 against which the norm of the difference is checked.

Return type

`bool`

property basis_states: List[StateStringT]

Returns the basis states with nonzero coefficient as a `list`.

clone()

Performs shallow copy of the object.

Return type

`TypeVar(SYMBOLICTYPE, bound= Symbolic)`

property coefficients: List[int | float | complex | Expr]

Returns the scalar coefficients of individual basis vectors as a `list`.

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 `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 ansatzes and digital quantum simulation. Otherwise, it may remove terms relevant to computation. Each expression is of the form $f(a_1, a_2, \dots, a_n)$ for some symbols a_i . $|f(a_1, a_2, \dots, a_n)|$ 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(state_vector, modes=None, reverse=False)`

Returns an instance representing the provided ndarray state vector.

This method will take a state vector as a `ndarray` column vector and convert it to a `State` object. Modes - and the bit position in the state index to which each corresponds - can be defined with the `modes` parameter. In the absence of this parameter, a default register of the appropriate size is implicit, with the index being assumed to correspond to an ILO-BE ordering of the default register.

In keeping with numpy convention, 1-D arrays and 2-D arrays with shape $(N, 1)$ are treated as column vectors. The length N of the vector must be either 0 or a power of 2. Differently shaped arrays are not accepted.

Parameters

- **state_vector** (ndarray) – The state vector representing the state to be created.
- **modes** (`Optional[Iterable[TypeVar(ModeT)]]`, default: `None`) – An ordering of modes used for defining indices, from most significant to least significant. Default behaviour as above.
- **reverse** (default: `False`) – If `True`, reverse bit order of state vector indices (e.g. for changing endianness).

Returns

An instance of this class representing the provided `ndarray` state vector.

Raises

`ValueError` – If the input state vector has invalid shape.

`classmethod from_sparray(state_vector, modes=None, reverse=False)`

Returns an instance representing the provided scipy sparse array `state_vector`.

This method will take a state vector as a scipy sparse array and convert it to a `State` object. Modes - and the bit position in the state index to which each corresponds - can be defined with the `modes` parameter. In the absence of this parameter, a default register of the appropriate size is implicit, with the index being assumed to correspond to an ILO-BE ordering of the default register.

The shape of the array must be $(N, 1)$ where N is either 0 or a power of 2. Differently shaped arrays are not accepted.

Parameters

- **state_vector** (spmatrix) – The state vector representing the state to be created.
- **modes** (`Optional[Iterable[TypeVar(ModeT)]]`, default: `None`) – An ordering of modes used for defining indices, from most significant to least significant. Default behaviour as above.
- **reverse** (default: `False`) – If `True`, reverse bit order of state vector indices (e.g. for changing endianness).

Returns

An instance of this class representing `state_vector`.

`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

`TypeVar(StateT, bound= State)` – Child class object.

`get_numeric_representation()`

Placeholder, unimplemented method.

`get_symbolic_representation()`

Placeholder, unimplemented method.

`is_all_coeff_complex()`

Check if all coefficients have complex values.

Returns

`bool` – `False` if a non-complex value occurs before any free symbols in the `dict` values, or `True` if no non-complex values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any non-complex values in the coefficients.

`is_all_coeff_imag()`

Check if all coefficients have purely imaginary values.

Returns

`bool` – `False` if a non-complex value occurs before any free symbols in the `dict` values, or `True` if no non-complex values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any non-imaginary values in the coefficients.

`is_all_coeff_real()`

Check if all coefficients have real values.

Returns

`bool` – `False` if a non-real value occurs before any free symbols in the `dict` values, or `True` if no non-real values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any non-real values in the `dict` coefficients.

`is_all_coeff_symbolic()`

Check if all coefficients contain free symbols.

Returns

`bool` – Whether all coefficients contain free symbols.

`is_any_coeff_complex()`

Check if any coefficients have complex values.

Returns

`bool` – `True` if a complex value occurs before any free symbols in the `dict` values, or `False` if no complex values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any complex values in the coefficients.

is_any_coeff_imag()

Check if any coefficients have imaginary values.

Returns

`bool` – True if an imaginary value occurs before any free symbols in the `dict` values, or `False` if no imaginary values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any imaginary values in the coefficients.

is_any_coeff_real()

Check if any coefficients have real values.

Returns

`bool` – True if a real value occurs before any free symbols in the `dict` values, or `False` if no real values occur.

⚠ Warning

Returns `None` if a free symbol occurs before any real values in the coefficients.

is_any_coeff_symbolic()

Check if any coefficients contain free symbols.

Returns

`bool` – Whether any coefficients contain free symbols.

is_basis_state()

Checks this object represents a scalar multiple of a single basis state.

Return type

`bool`

is_normalized(*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_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]]`

classmethod key_from_str(`key_str`)

Converts a string to an instance of this class's basis state type.

Parameters

`key_str` (`str`)

Return type

`TypeVar(StateStringT, bound= StateString)`

classmethod load_h5(`name`)

Loads a state object from .h5 file.

Parameters

`name` (`Union[str, Group]`) – Name of .h5 file to be loaded.

Returns

`TypeVar(StateT, bound= State)` – Loaded integral operator object.

`make_hashable()`

Return a hashable representation of the object.

Returns

`str` – A string representation of this instance.

`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` – This instance.

`property n_symbols: int`

Returns the number of free symbols in the object.

`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.

`property num_modes: int`

Returns the number of modes.

`print_table()`

Print dictionary formatted as a table.

Return type

`NoReturn`

`remove_global_phase()`

Returns a copy with a global phase applied such that the first element has a real coefficient.

This is an alias for `set_global_phase()` - see this method for greater control over the phase to be applied.

Returns

`LinearDictCombiner` – A copy of the object with a global phase applied such that the first element has a real coefficient.

`reversed_order()`

Reverses the order of terms and returns it as a new object.

Return type

LinearDictCombiner

save_h5 (name)

Dumps state object to .h5 file.

Parameters**name** (Union[str, Group]) – Destination filename of .h5 file.**Return type**

None

set_global_phase (phase=0.0)

Returns a copy with a global phase applied such that the first element has the desired phase.

Parameters

- **phase** (Union[int, float, complex, Expr], default: 0.0) – The phase to yield on the first element, in half-turns (i.e. multiples of pi).

- **phase.** (A symbolic expression can be assigned to)

Returns

LinearDictCombiner – A copy of the object with the desired global phase applied.

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: StateStringTReturns the stored configuration as a *StateString*, if only a single configuration is stored.**Raises**`RuntimeError` – if more or less than one configuration is stored**Returns**

The single configuration occupation vector.

split ()Generates *State* objects containing only single configurations, preserving coefficients.**Return type**

Generator[TypeVar(StateT, bound= State), None, None]

string_classalias of *StateString***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]) – A mapping for substitution of free symbols.

Returns

`TypeVar(SYMBOLICTYPE, bound= Symbolic)` – A copy of self with symbols substituted according to the provided map.

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 `sympify()`.
- `kwargs (Any)` – Kwargs to be passed to `sympify()`.

Returns

`LinearDictCombiner` – Updated instance of `LinearDictCombiner`.

Raises

`RuntimeError` – Sympification fails.

property terms: List [Any]

Returns dictionary keys.

to_ndarray (modes=None, dtype=complex, reverse=False)

Returns a state vector representation of this state.

Modes - and the bit position in the state index to which each corresponds - can be defined with the `modes` parameter. In the absence of this parameter, a default register of the appropriate size is implicit, with the index being assumed to correspond to an ILO-BE ordering of the default register. See below for examples.

Parameters

- `modes (Optional[List[TypeVar(ModeT)]])`, default: `None` – An ordering of modes used for defining indices, from most significant to least significant. Default behaviour as above.
- `dtype (default: complex)` – The numpy `dtype` to be used for the returned vector.
- `reverse (bool, default: False)` – If `True`, reverse bit order of state vector indices (e.g. for changing endianness).

Returns

`ndarray` – A dense numpy `ndarray` representation of the state.

➊ Examples

```
>>> qss0 = StateString({0:1,1:1,2:0,3:0})
>>> qss1 = StateString({0:1,1:0,2:1,3:0})
>>> qs = State({qss0: 0.9, qss1: 0.1})
>>> state_vector = qs.to_ndarray()
```

```

>>> print(state_vector)
[[0. +0.j]
 [0. +0.j]
 [0.1+0.j]
 [0. +0.j]
 [0.9+0.j]
 [0. +0.j]
 [0. +0.j]
 [0. +0.j]]
>>> state_vector = qs.to_ndarray([0,2,3,1])
>>> print(state_vector)
[[0. +0.j]
 [0. +0.j]
 [0.9+0.j]
 [0. +0.j]
 [0. +0.j]
 [0.1+0.j]
 [0. +0.j]
 [0. +0.j]
 [0. +0.j]]

```

`to_sparray(modes=None, matrix_type=scipy.sparse.csr_array, dtype=complex, reverse=False)`

Returns a scipy sparse array state vector representation of this state.

Modes - and the bit position in the state index to which each corresponds - can be defined with the `modes` parameter. In the absence of this parameter, a default register of the appropriate size is implicit, with the index being assumed to correspond to an ILO-BE ordering of the default register. See [to_ndarray\(\)](#) for examples.

Parameters

- **modes** (`Optional[List[TypeVar(ModeT)]]`, default: `None`) – An ordering of modes used for defining indices, from most significant to least significant. Default behaviour as above.
- **matrix_type** (default: `scipy.sparse.csr_array`) – The type of the sparse matrix to be generated.
- **dtype** (default: `complex`) – The numpy `dtype` to be used for the returned vector.
- **reverse** (`bool`, default: `False`) – If `True`, reverse bit order of state vector indices (e.g.

for changing endianness).

Returns

A scipy sparse array representation of the state.

unsympify (*precision*=15, *partial*=False)

Unsympifies dictionary values.

Replaces symbolic expressions with their corresponding numeric values.

Parameters

- **precision** (`int`, default: 15) – The number of decimal digits of precision used for evaluation.
- **partial** (`bool`, default: False) – Set to True to allow partial unsympification where terms containing free symbols are present. By default, free symbols in any coefficient will cause an exception.

Returns

`LinearDictCombiner` – Updated instance of `LinearDictCombiner`.

Raises

`TypeError` – Unsympification fails.

vdot (*other*)

Calculates the inner product with another `State` object.

The other `State` is treated as the ket, this one is treated as the bra - i.e. the coefficient

Parameters

- **other** (`TypeVar(StateT, bound= State)`) – Another `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())
()
```

class StateString(*data*)

Bases: `UserDict`, `Generic[ModeT]`

Represents a basis vector in an arbitrary Hilbert space.

This class is generic with regard to the nature of the underlying modes. Typically, one should use subclasses which represent basis vectors within a specific space - see the documentation for `QubitStateString` and `FermionStateString`. Subclasses must define the `mode_class` attribute, which determines the class of the underlying mode.

Parameters

- **data** (`Union[Dict[TypeVar(ModeT), Union[int, bool]], List[Union[int, bool]], Tuple[Union[int, bool]]]`)

property all_modes: FrozenSet

Returns all mode objects in the space of which this *StateString* is a basis state.

classmethod from_index(state_index, modes=None, reverse=False)

Creates an instance from the index of a state vector corresponding to a single basis state.

Modes - and the bit position in the state index to which each corresponds - can be defined with the *modes* parameter. In the absence of this parameter, a default register of the appropriate size is implicit, with the index being assumed to correspond to an ILO-BE ordering of the default register. See below for examples.

Parameters

- **state_index** (`int`) – The index of a state vector corresponding to the basis state to which the created object will correspond.
- **modes** (`Optional[Iterable[TypeVar(ModeT)]]`, default: `None`) – A collection of modes, the state values of which are represented by the state vector, in the appropriate ordering.
- **reverse** (default: `False`) – If `True`, reverse the bits of the index.

Returns

`TypeVar(StateStringT, bound= StateString)` – An instance of this class corresponding to the basis state to which the provided index corresponds.

Examples

```
>>> index = 12
>>> statestring = StateString.from_index(index)
>>> print(statestring)
{0: 1, 1: 1, 2: 0, 3: 0}
>>> statestring = StateString.from_index(index, modes=[3, 0, 1, 2])
>>> print(statestring)
{3: 1, 0: 1, 1: 0, 2: 0}
```

classmethod from_list_int(data)

Construct a *StateString* from a *list* of integers or booleans.

The integers within the *list* should correspond to the states of individual modes. Modes will be assumed to be indexed in the `[0, len(data)]` range in ascending order.

Parameters

data (`List[Union[int, bool]]`) – State values to assign for the default mode set.

Returns

`TypeVar(StateStringT, bound= StateString)` – An instance of this class representing the basis state with provided state values, using a default set of modes.

classmethod from_string(data)

Generate a class instance from a string.

This method will make a best effort to interpret a string as an instance of this class. It is provided largely for backwards compatibility. Caution is advised.

Parameters

data (`str`) – A string from which to attempt to generate an instance of this class.

Returns

`TypeVar(StateStringT, bound= StateString)` – A class instance generated from the input string.

property hamming_weight: int

Returns the Hamming weight of the basis state.

classmethod load_h5(name)

Loads a state string object from .h5 file.

Parameters

`name (Union[str, Group])` – Name of .h5 file to be loaded.

Returns

`TypeVar(StateStringT, bound= StateString)` – Loaded integral operator object.

mode_class

alias of `int`

property num_modes: int

Returns the number of modes in the space of which this `StateString` is a basis state.

occupations_ordered()

Returns a `list` of integer or boolean occupation numbers corresponding to modes ordered in increasing lexicographic order.

Return type

`List[Union[int, bool]]`

save_h5(name)

Dumps state string object to .h5 file.

Parameters

`name (Union[str, Group])` – Destination filename of .h5 file.

Return type

`None`

to_index(modes=None, reverse=False)

Returns the index of the state vector to which this basis state corresponds.

The ordering of modes when defining indexing into the state vector can be controlled with the `modes` parameter. By default, `modes` will be ordered following an ILO-BE convention - see examples.

Parameters

- `modes (Optional[Iterable[TypeVar(ModeT)]]`, default: `None`) – An ordering of modes used for defining state vector indices, from most significant to least significant.
- `reverse (default: False)` – If `True`, reverse the order of bits in the state vector index.

Returns

`int` – A state vector index corresponding to this basis state.

Raises

`ValueError` – If provided modes are not the same as those of the state.

i **Examples**

```
>>> statestring = StateString({0:1,1:1,2:0,3:0})
>>> index = statestring.to_index()
>>> print(index)
12
>>> index = statestring.to_index([0,2,3,1])
>>> print(index)
9
```

27.14 inquanto.symmetry

Module for point-group symmetry analysis.

`class PointGroup(point_group)`

Bases: `object`

Point group class with associated symmetry processing methods.

Parameters

`point_group (str)` – The point group label as a string - e.g. "C_{2v}".

`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 (Union[list[int], ndarray])` – Representation to be reduced.

Returns

`list[tuple[int, 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, Union[tuple[str, ...], tuple[int, ...], int, str]]` – 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[Union[tuple[str, ...], tuple[int, ...], int, str], 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[str, dict[str, Union[tuple[str, ...], tuple[int, ...], int, str]]]` – A dictionary with point group labels as keys ("C_{2v}" 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, Union[tuple[str, ...], tuple[int, ...], int, str]]` – 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 (`list[str]`) – Iterable containing irrep symbols which belong to the `PointGroup` object's group.

Returns

`tuple[list[tuple[int, str]], ndarray]` – 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.

Return type

`None`

static supported_groups ()

Return a `list` of point groups supported by the `PointGroup` object.

Return type

`list`

class TapererZ2 (symmetry_operators, symmetry_sectors, skip_fast_find_x_operators=False)

Bases: `object`

Performs \mathbb{Z}_2 symmetry qubit tapering.

The procedure followed is as described by Bravyi et al. ([arXiv:1701.08213 \[quant-ph\]](#)). This technique uses \mathbb{Z}_2 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 an 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 upon instantiation and cached. For advanced usage, these may be manually regenerated with `find_x_operators()` and `tapering_unitary()`.

Parameters

- **`symmetry_operators`** (`list[SymmetryOperatorPauli]`) – A list of \mathbb{Z}_2 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`

\mathbb{Z}_2 symmetry operators for which to use tapering.

`symmetry_sectors`

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.

`find_x_operators(skip_minimal=False, regenerate=False, cache_on_regeneration=True)`

Find a list of X operators for generating tapering unitaries.

The key equation in this procedure is arXiv:1701.08213 [quant-ph] eq. 61. Generated operators include exclusively X or I . The `QubitOperatorString` with index i in the returned list will anticommute with the symmetry operator of index i contained in this instance, and will 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 `SymmetryZ2Qubit`, 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 expensive 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`** (`bool`, default: `False`) – Set to `True` to skip attempting to find single qubit X operators.
- **`regenerate`** (`bool`, default: `False`) – 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_regeneration`** (`bool`, default: `True`) – If set, cached x operators will be overwritten if X operators are regenerated.

Returns

`tuple[list[QubitOperatorString], bool]` – Operators including only X operators used to form the tapering unitary and a boolean, indicating whether single qubit X operators were calculated.

tapered_operator(*qubit_operator*, *relabel_qubits=False*, *taperable_qubits=None*)

Given a *QubitOperator*, find its tapered form.

The tapered operator is as described by Bravyi, Gambetta, Mezzacapo & Temme (arXiv:1701.08213 [quant-ph]), with the strings of Pauli *X* operators 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*** (*Optional[Iterable[Qubit]]*, default: *None*) – For advanced usage - specify an explicit iterable 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*, *relabel_qubits=False*)

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.
- ***relabel_qubits*** (*bool*, default: *False*) – Set to *True* to relabel all qubits by decreasing index according to number of qubits removed.

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 \mathbb{Z}_2 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 - but this behaviour can be controlled with the *regenerate* and *cache_on_regenerate* parameters. The optional *x_operators* is a *list* of *QubitOperatorString* objects including only *X* or *I*, in which the element with index *i* will anticommute with the symmetry operator of index *i* contained in this instance, and will commute 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** (`Union[list[QubitOperatorString], list[SymmetryOperatorPauli], None]`, default: `None`) – For advanced usage, a `list` of operators including only X operators used to form the tapering unitary may be provided. If `None` (the default), cached X operators will be used.

Returns

`QubitOperator` – The Clifford tapering unitary.

INQUANTO-EXTENSIONS API REFERENCE

28.1 inquanto-pyscf

InQuanto PySCF extension.

```
class AVAS(aolabels, aolabels_vir=None, threshold=0.2, threshold_vir=0.2, n_occ=None, n_vir=None,
           n_occ_active=None, n_vir_active=None, minao='minao', with_iao=False, force_halves_active=False,
           freeze_half_filled=False, canonicalize=True, frozen=None, spin_as=None, verbose=None)
```

AVAS (Atomic Valence Active Space) / RE (Regional Embedding class) to construct MCSCF active space.

Parameters

- **aolabels** (`List[str]`) – AO labels for AO active space, for example ['Cr 3d', 'Cr 4s'] or ["1 C 2p", "2 C 2p"].
- **aolabels_vir** (`List[str]`, default: `None`) – If given, separate AO labels for the virtual orbitals. If not given, aolabels is used.
- **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 localised occupied orbitals to create. If specified, the value of threshold is ignored.
- **n_vir** (`Optional[int]`, default: `None`) – None or number of localised virtual orbitals to create. If specified, the value of threshold_vir is ignored.
- **n_occ_active** (`Optional[int]`, default: `None`) – Optional. If specified, number of returned occupied orbitals in the active space (out of the total number of localised occupied orbitals defined by n_occ).
- **n_vir_active** (`Optional[int]`, default: `None`) – Optional. If specified, number of returned virtual orbitals in the active space (out of the total number of localised virtual orbitals defined by n_vir).
- **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 2.5 option 2 or 3 of the [AVAS](#) paper for more details.

- **`freeze_half_filled`** (`bool`, default: `False`) – If True, all half-filled orbitals (if present) are frozen, i.e. excluded from the AVAS transformation and from the active space.
- **`canonicalize`** (`bool`, default: `True`) – Block-diagonalize and symmetrize the core, active and virtual orbitals.
- **`frozen`** (`Union[List[int], List[List[int]]]`, default: `None`) – List of orbitals to be excluded from the AVAS method.
- **`spin_as`** (`int`, default: `None`) – Number of unpaired electrons in the active space.
- **`verbose`** (`int`, default: `None`) – Control PySCF verbosity.

i 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.

i 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.

Parameters

`mf` (`HF`) – PySCF mean-field object.

Returns

`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – Unitary matrix to transform the original MO coeffs.

`dump_flags` ()

Print all modifiable options and their current values.

Return type

`None`

`frozenf` (*mf*)

Return a list of frozen orbitals.

Parameters

`mf` (`HF`) – PySCF mean-field object.

Returns

`List[int]` – Frozen orbital indices.

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_co]]

Returns the original MO coefficient matrix.

Returns

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\[\[TypeVar\\(_ScalarType_co, bound= generic, covariant=True\\)\\]\\]\\]\]\(#\) – 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(ncas, nelecas, ncore=None, max_cycle=50, conv_tol=1e-7, conv_tol_grad=None,
fix_spin_squared=None, spin_squared_tolerance=0.01, transf=None, init_orbitals=None,
print_ci_coeff=False, ci_print_cutoff=0.1)
```

Postprocess orbitals with Complete Active Space Self-Consistent Field (CASSCF) method for building molecular integrals.

Parameters

- **ncas** ([int](#)) – number of active spatial orbitals passed to [pyscf.mcscf.CASSCF](#).
- **nelecas** ([int](#)) – number of active electrons passed to [pyscf.mcscf.CASSCF](#).
- **ncore** ([Optional\[int\]](#), default: [None](#)) – (optional) number of core orbitals passed to [pyscf.mcscf.CASSCF](#). In most cases, it is unnecessary as PySCF computes ncore correctly. Can be necessary when the active space contains some “chemical core” orbitals.
- **max_cycle** ([int](#), default: 50) – maximum number of CASSCF macroiterations.
- **conv_tol** ([float](#), default: $1e-7$) – energy convergence threshold.
- **conv_tol_grad** ([Optional\[float\]](#), default: [None](#)) – orbital gradient convergence threshold.
- **fix_spin_squared** ([Optional\[float\]](#), default: [None](#)) – desired value of S^2 . If not [None](#), a penalty term is added to the energy to drive optimisation to the desired state.
- **spin_squared_tolerance** ([float](#), default: 0.01) – tolerance of S^2 .

- **transf** (`Union[Transf, Callable, None]`, default: `None`) – orbital transformation (e.g. AVAS) to be performed before CASSCF.
- **init_orbitals** (`Optional[ndarray]`, default: `None`) – initial orbital coefficients, can be from a different geometry.
- **print_ci_coeff** (`bool`, default: `False`) – If True, the determinants which have coefficients $> ci_print_cutoff$ will be printed out.
- **ci_print_cutoff** (`float`, default: `0.1`) – Tolerance for printing CI coefficients.

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.

Parameters

`mf` (`HF`) – PySCF mean-field object.

Returns

`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – Unitary matrix to transform the original MO coeffs.

`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_co]]`

Returns the original MO coefficient matrix.

Returns

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 ChemistryDriverPySCFEmbeddingGammaRHF (geometry=None, zmatrix=None, cell=None, basis=None,
                                             ecp=None, charge=0, exp_to_discard=None, df='GDF',
                                             dimension=3, frozen=None, transf=None, verbose=0,
                                             output=None, functional='b3lyp', transf_inner=None,
                                             frozen_inner=None, level_shift=1.0e6, precision=1e-9)
```

Projection-based embedding. Partially based on PsiEmbed.

Implements the [Projection-based embedding](#) method. Runs a RHF (if `functional` is `None`) or RKS calculation on the whole system and then creates an embedding, the subsystem is defined by the orbitals selected with `frozen`.

Parameters

- `geometry` (`Union[List, str, GeometryPeriodic]`, default: `None`) – Molecular geometry.
- `zmatrix` (`str`, default: `None`) – Z matrix representation of molecular geometry (Used only if `geometry` is not specified).
- `cell` (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`, default: `None`) – unit cell vectors (if not specified in the `geometry` object).
- `basis` (`Any`, default: `None`) – Atomic basis set valid for `Mole` class.
- `ecp` (`Any`, default: `None`) – Effective core potentials.
- `charge` (`int`, default: `0`) – Total charge.
- `exp_to_discard` (`float`, default: `None`) – Exponent to discard a primitive Gaussian.
- `df` (`str`, default: "GDF") – Type of density fitting ("GDF", "FFTDF", "AFTDF" or "MDF").
- `dimension` (`int`, default: `3`) – Number of spatial dimensions.
- `frozen` (`Union[List[int], Callable[[RHF], int]]`, default: `None`) – Frozen orbital information.
- `transf` (`Transf`, default: `None`) – Orbital transformer.
- `verbose` (`int`, default: `0`) – Control PySCF verbosity.
- `output` (`str`, default: `None`) – Specify log file name. If `None`, logs are printed to STDOUT.
- `functional` (`str`, default: "b3lyp") – KS functional to use for the system calculation, or `None` if RHF is desired.
- `transf_inner` (`Union[Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], Transf]`, default: `None`) – Orbital transformer to be used on the sub-system Hamiltonian.
- `frozen_inner` (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`) – List of frozen orbitals in the sub-system Hamiltonian.
- `level_shift` (`float`, default: `1.0e6`) – Controls the projection-based embedding.
- `precision` (`float`, default: `1e-9`) – Ewald sum precision.

`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** (default: `True`) – Reduce infinite point groups, e.g., C_{00v} -> C_{2v}.

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, df='GDF', 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]]]]]`, default: `None`) – Frozen core specified as either list or callable.
- **df** (`str`, default: "GDF") – Type of density fitting ("GDF", "FFTDF", "AFTDF" or "MDF").

- **transf** (`Union[Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], Transf]`, default: `None`) – Orbital transformation function.
- **transf_inner** (`Union[Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], Transf]`, default: `None`) – Orbital transformation function to be applied to the embedded Hamiltonian.
- **frozen_inner** (`(Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]], default: None)` – Frozen orbitals in the embedded Hamiltonian.
- **level_shift** (`float`, default: `1.0e6`) – Projection-based embedding level shift.

Returns

`ChemistryDriverPySCFEmbeddingGammaRHF` – PySCF driver.

property frozen: List[int] | List[List[int]]

Return the frozen orbital information.

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_ac0_correction(rdm)

Not implemented. To obtain AC0 correction to WFT-in-DFT, use `get_subsystem_driver()`.

Parameters

`rdm` (`Tuple[ndarray[Any, dtype[float]], ndarray[Any, dtype[float]]]`)

get_cascl_1234pdms()

Calculate 1-, 2-, 3- and 4-PDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1-, 2-, 3- and 4-PDMs.

get_cascl_12rdms()

Calculate 1-RDM and 2-RDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1- and 2-RDMs.

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`.

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_double_factorized_system(tol1=-1, tol2=None, method=DecompositionMethod.EIG,`
`diagonalize_one_body=True, diagonalize_one_body_offset=True,`
`combine_one_body_terms=True)`

Calculate double-factorized Hamiltonian operator, Fock space, and Hartree-Fock state.

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 which is to be double-factorized, and S is a one-body energy offset 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.

First level of factorization decomposes the electron repulsion integral (ERI) tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or pivoted, incomplete Cholesky decomposition (`method='cho'`). For details about Cholesky decomposition, refer to References [1] and [2]. The second factorization is diagonalization of the V_{pq}^t matrix for each t: $V_{pq}^t = \sum_u^{N_\lambda} U_{pu}^t \lambda_u^t U_{qu}^t$.

At the first factorization stage, truncation depends on the decomposition method. For ‘eig’, we discard eigenvalues, starting from the smallest, until the sum of those discarded exceeds `tol1`. For ‘cho’, the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. 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 decomposition of ERI matrix. If negative, no truncation is performed.
- `tol2` (`Optional[float]`, default: `None`) – Truncation threshold for second decomposition. If `None`, same as `tol1`.

- **method** (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`) – Decomposition method used for the first level of factorization. ‘eig’ for an eigenvalue decomposition, ‘cho’ for a pivoted, incomplete Cholesky decomposition.
- **diagonalize_one_body** (`bool`, default: `True`) – Whether to diagonalize the physical one-body integrals h_{pq} .
- **diagonalize_one_body_offset** (`bool`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq} .
- **combine_one_body_terms** (`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

`Tuple[DoubleFactorizedHamiltonian, FermionSpace, FermionState]` – Hamiltonian operator storing two-body integrals in double factorized form, Fock space, Fock state.

References

1. Henrik Koch, Alfredo Sánchez de Merás, and Thomas Bondo Pedersen. Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.*, 118(21): 9481-9484, 2003. URL: <https://doi.org/10.1063/1.1578621>, doi:10.1063/1.1578621
2. Evgeny Epifanovsky, Dmitry Zuev, Xintian Feng, Kirill Khistyayev, Yang Shao, and Anna I. Krylov. General implementation of the resolution-of-the-identity and Cholesky representations of electron repulsion integrals within coupled-cluster and equation-of-motion methods: Theory and benchmarks. *J. Chem. Phys.*, 139(13): 134105, 2013. URL: <https://doi.org/10.1063/1.4820484>, doi:10.1063/1.4820484

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 single-electron excitation amplitudes.
- **t2** (`Optional[ndarray]`, default: `None`) – Guess two-electron 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 single-electron excitation amplitudes.
- **t2** (`Optional[ndarray]`, default: `None`) – Guess two-electron excitation amplitudes.
- **compact** (`bool`, default: `False`) – Enable compact form of excitation for the restricted wavefunctions.
- **antihermitian** (`bool`, default: `True`) – Returns anti-Hermitian operators $T - T^\dagger$ if specified.

Returns

`FermionOperatorList` – Excitation operator list.

get_lowdin_system (`method='lowdin'`, `store_ao=False`)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

- **method** (`str`, default: "lowdin") – Method passed to PySCF's `orth.orth_ao()`.
- **store_ao** (`bool`, default: `False`) – If True, the returned Hamiltonian operator is of type `PySCFChemistryRestrictedIntegralOperator` or `PySCFChemistryUnrestrictedIntegralOperator`, which stores the underlying atomic orbitals internally.

Returns

`Tuple[Union[ChemistryRestrictedIntegralOperator, ChemistryUnrestrictedIntegralOperator, PySCFChemistryRestrictedIntegralOperator, PySCFChemistryUnrestrictedIntegralOperator], FermionSpace, Union[RestrictedOneBodyRDM, UnrestrictedOneBodyRDM]]` – Fermion Hamiltonian, Fock space, Fock state.

↳ See also

`orth.orth_ao` documentation.

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_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction(rdm)

Not implemented. To obtain NEVPT2 correction to WFT-in-DFT, use `get_subsystem_driver()`.

Parameters

`rdms` (`Tuple`)

get_orbital_coefficients()

Returns orbital coefficients.

Returns

`ndarray` – Orbital coefficients.

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_co, bound= generic, covariant=True)]]` – Two-body reduced density matrix.

Note

This object will be replaced with an `RestrictedTwoBodyRDM` class to avoid returning a raw 4D tensor.

get_subsystem_driver(frozen=None, transf=None)

Generate a driver object wrapping the current active space Hamiltonian.

Parameters

- **frozen** (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: None) – Frozen orbital information (applies to the subsystem driver).
- **transf** (`Union[Callable[[array], array], Transf]`, default: None) – Orbital transformer (applies to the subsystem driver).

Returns

`BasePySCFDriver` – PySCF driver object wrapping the current active space Hamiltonian.

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.
- **as** (*Uses the same convention*) – PySCF. Currently, supports s1, s4 and s8 index permutation symmetries.

Returns

`Tuple[Union[ChemistryRestrictedIntegralOperator, ChemistryRestrictedIntegralOperatorCompact], 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` 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[PySCFChemistryRestrictedIntegralOperator, Optional[FermionSpace], Optional[FermionState]]` – Fermion Hamiltonian, Fock space, Fock state.

`make_actives_contiguous()`

Reorder orbitals so that active orbitals form a contiguous block.

Reorders orbital coefficients, orbital energies, occupations and the list of frozen (active) orbitals. Implemented for RHF, ROHF and UHF.

`property mf_energy: float`

Return the total mean-field energy.

Returns

Total mean-field energy.

`property mf_type: str`

Return the mean-field type as a string, with options including "RHF", "ROHF", and "UHF".

Returns

Mean-field type name.

`property n_electron: int`

Return the total number of electrons in the active space.

Returns

Total number of electrons.

`property n_orb: int`

Return the number of spatial orbitals.

Returns

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.

set_checkfile(chkfile, init_guess=True)

Set checkpoint file name.

The PySCF calculation results will be saved to the checkpoint file.

Parameters

- `chkfile (str)` – name of checkpoint file.
- `init_guess (bool, default: True)` – If True and the checkpoint file exists, the initial guess will be read from the checkpoint file.

set_diis_space_dimension(diis_space_dimension=8)

Set number of DIIS vectors.

Parameters

`diis_space_dimension (int, default: 8)` – dimension of the DIIS space.

set_init_orbitals(init_orbs)

Sets the initial guess orbitals for the SCF.

Parameters

`init_orbs (ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]])` – Initial orbital coefficients.

Return type

`None`

set_level_shift(level_shift_value=0)

Set value of the artificial shift applied to virtual orbitals during SCF.

Note

Level shifting in PySCF changes the HF energy, even though the last SCF iteration is performed without it.

Parameters

`level_shift_value (float, default: 0)` – value of the level shift parameter.

set_max_scf_cycles(max_cycles=50)

Set maximum number of SCF cycles.

Parameters

`max_cycles (int, default: 50)` – maximum number of SCF cycles.

```
class ChemistryDriverPySCFEmbeddingGammaROHF_UHF (geometry=None, zmatrix=None, cell=None,
                                                 basis=None, ecp=None, charge=0, multiplicity=1,
                                                 exp_to_discard=None, df='GDF', dimension=3,
                                                 frozen=None, transf=None, verbose=0,
                                                 output=None, embedded_spin=None,
                                                 functional='b3lyp', transf_inner=None,
                                                 frozen_inner=None, level_shift=1.0e6,
                                                 precision=1e-9)
```

Projection-based embedding. Partially based on PsiEmbed.

Implements Projection-based embedding method. Runs a ROHF (if functional is None) or ROKS calculation on the whole system and then creates an embedding, the subsystem is defined by the orbitals selected with frozen. The embedded calculation uses the same type of density fitting as the system calculation, is run as ROHF and subsequently converted to UHF. The Hamiltonian operator returned is unrestricted.

Parameters

- **geometry** (`Union[List, str, GeometryPeriodic]`, default: `None`) – Molecular geometry.
- **zmatrix** (`str`, default: `None`) – Z matrix representation of molecular geometry (Used only if geometry is not specified).
- **cell** (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`, default: `None`) – unit cell vectors (if not specified in the geometry object).
- **basis** (`Any`, default: `None`) – Atomic basis set valid for Mole class.
- **ecp** (`Any`, default: `None`) – Effective core potentials.
- **charge** (`int`, default: `0`) – Total charge.
- **multiplicity** (`int`, default: `1`) – Spin multiplicity of the total system.
- **exp_to_discard** (`float`, default: `None`) – Exponent to discard a primitive Gaussian.
- **df** (`str`, default: "GDF") – Type of density fitting ("GDF", "FFTDF", "AFTDF" or "MDF").
- **dimension** (`int`, default: `3`) – Number of spatial dimensions.
- **frozen** (`Union[List[int], Callable[[RHF], int]]`, default: `None`) – Frozen orbital information.
- **transf** (`Transf`, default: `None`) – Orbital transformer.
- **verbose** (`int`, default: `0`) – Control PySCF verbosity.
- **output** (`str`, default: `None`) – Specify log file name. If `None`, logs are printed to STDOUT.
- **embedded_spin** (`int`, default: `None`) – number of unpaired electrons in the sub-system. `None` means the same as in the total system.
- **functional** (`str`, default: "b3lyp") – KS functional to use for the system calculation, or `None` if RHF is desired.
- **transf_inner** (`Union[Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], Transf]`, default: `None`) – Orbital transformer to be used on the sub-system Hamiltonian.
- **frozen_inner** (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`) – List of frozen orbitals in the sub-system Hamiltonian.

- **level_shift** (`float`, default: `1.0e6`) – Controls the projection-based embedding.
- **precision** (`float`, default: `1e-9`) – Ewald sum precision.

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** (default: `True`) – Reduce infinite point groups, e.g., C_{oo}v → 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, df='GDF', 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.

Parameters

- **mf** (ROHF) – 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) – Frozen core specified as either list or callable.
- **transf** (Union[Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], Transf], default: None) – Orbital transformation function.
- **level_shift** (float, default: 1.0e6) – Projection-based embedding level shift.
- **df** (str, default: "GDF")
- **transf_inner** (Union[Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], Transf], default: None)
- **frozen_inner** (Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]], default: None)
- **embedded_spin** (int, default: None)

Returns

ChemistryDriverPySCFEmbeddingGammaROHF_UHF – PySCF driver.

```
property frozen: List[int] | List[List[int]]
```

Return the frozen orbital information.

```
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_ac0_correction(rdm)
```

Not implemented. To obtain AC0 correction to WFT-in-DFT, use get_subsystem_driver().

Parameters

rdm (Tuple[ndarray[Any, dtype[float]], ndarray[Any, dtype[float]]])

```
get_casci_1234pdms()
```

Calculate 1-, 2-, 3- and 4-PDM from a CASCI wavefunction.

Returns

Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]] – A tuple of 1-, 2-, 3- and 4-PDMs.

```
get_casci_12rdms()
```

Calculate 1-RDM and 2-RDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1- and 2-RDMs.

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_double_factorized_system (*tol1*=-1, *tol2*=None, *method*=`DecompositionMethod.EIG`, *diagonalize_one_body*=True, *diagonalize_one_body_offset*=True, *combine_one_body_terms*=True)

Calculate double-factorized Hamiltonian operator, Fock space, and Hartree-Fock state.

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 which is to be double-factorized, and S is a one-body energy offset 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.

First level of factorization decomposes the electron repulsion integral (ERI) tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or pivoted, incomplete Cholesky decomposition (`method='cho'`). For details about Cholesky decomposition, refer to References [1] and [2]. 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 the first factorization stage, truncation depends on the decomposition method. For ‘eig’, we discard eigenvalues, starting from the smallest, until the sum of those discarded exceeds `tol1`. For ‘cho’, the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. 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 decomposition of ERI matrix. If negative, no truncation is performed.
- `tol2` (`Optional[float]`, default: `None`) – Truncation threshold for second decomposition. If `None`, same as `tol1`.
- `method` (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`) – Decomposition method used for the first level of factorization. ‘eig’ for an eigenvalue decomposition, ‘cho’ for a pivoted, incomplete Cholesky decomposition.
- `diagonalize_one_body` (`bool`, default: `True`) – Whether to diagonalize the physical one-body integrals h_{pq} .
- `diagonalize_one_body_offset` (`bool`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq} .
- `combine_one_body_terms` (`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

`Tuple[DoubleFactorizedHamiltonian, FermionSpace, FermionState]` – Hamiltonian operator storing two-body integrals in double factorized form, Fock space, Fock state.

References

1. Henrik Koch, Alfredo Sánchez de Merás, and Thomas Bondo Pedersen. Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.*, 118(21): 9481-9484, 2003. URL: <https://doi.org/10.1063/1.1578621>, doi:10.1063/1.1578621
2. Evgeny Epifanovsky, Dmitry Zuev, Xintian Feng, Kirill Khistyayev, Yang Shao, and Anna I. Krylov. General implementation of the resolution-of-the-identity and Cholesky representations of electron repulsion integrals within coupled-cluster and equation-of-motion methods: Theory and benchmarks. *J. Chem. Phys.*, 139(13): 134105, 2013. URL: <https://doi.org/10.1063/1.4820484>, doi:10.1063/1.4820484

`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 single-electron excitation amplitudes.
- `t2` (`Optional[ndarray]`, default: `None`) – Guess two-electron 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 single-electron excitation amplitudes.
- **t2** (`Optional[ndarray]`, default: `None`) – Guess two-electron excitation amplitudes.
- **compact** (`bool`, default: `False`) – Enable compact form of excitation for the restricted wavefunctions.
- **antihermitian** (`bool`, default: `True`) – Returns anti-Hermitian operators $T - T^\dagger$ if specified.

Returns

`FermionOperatorList` – Excitation operator list.

`get_lowdin_system(method='lowdin', store_ao=False)`

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

- **method** (`str`, default: "lowdin") – Method passed to PySCF's `orth.orth_ao()`.
- **store_ao** (`bool`, default: `False`) – If True, the returned Hamiltonian operator is of type `PySCFChemistryRestrictedIntegralOperator` or `PySCFChemistryUnrestrictedIntegralOperator`, which stores the underlying atomic orbitals internally.

Returns

`Tuple[Union[ChemistryRestrictedIntegralOperator, ChemistryUnrestrictedIntegralOperator, PySCFChemistryRestrictedIntegralOperator, PySCFChemistryUnrestrictedIntegralOperator], FermionSpace, Union[RestrictedOneBodyRDM, UnrestrictedOneBodyRDM]]` – Fermion Hamiltonian, Fock space, Fock state.

See also

`orth.orth_ao` documentation.

`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_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction (rdms)

Not implemented. To obtain NEVPT2 correction to WFT-in-DFT, use `get_subsystem_driver()`.

Parameters

`rdms` (`Tuple`)

get_orbital_coefficients ()

Returns orbital coefficients.

Returns

`ndarray` – Orbital coefficients.

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_co, bound= generic, covariant=True)]]` – Two-body reduced density matrix.

Note

This object will be replaced with an `RestrictedTwoBodyRDM` class to avoid returning a raw 4D tensor.

get_subsystem_driver (frozen=None, transf=None)

Generate a driver object wrapping the current active space Hamiltonian.

Parameters

- `frozen` (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`) – Frozen orbital information (applies to the subsystem driver).
- `transf` (`Union[Callable[[array], array], Transf]`, default: `None`) – Orbital transformer (applies to the subsystem driver).

Returns

`BasePySCFDriver` – PySCF driver object wrapping the current active space Hamiltonian.

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[ChemistryUnrestrictedIntegralOperator, ChemistryUnrestrictedIntegralOperatorCompact], FermionSpace, FermionState]` – Fermion Hamiltonian, Fock space.

get_system_ao (run_hf=True)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Output hamiltonian operator will be of type `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[PySCFChemistryUnrestrictedIntegralOperator, Optional[FermionSpace], Optional[FermionState]]` – Fermion Hamiltonian, Fock space, Fock state.

make_actives_contiguous()

Reorder orbitals so that active orbitals form a contiguous block.

Reorders orbital coefficients, orbital energies, occupations and the list of frozen (active) orbitals. Implemented for RHF, ROHF and UHF.

property mf_energy: float

Return the total mean-field energy.

Returns

Total mean-field energy.

property mf_type: str

Return the mean-field type as a string, with options including "RHF", "ROHF", and "UHF".

Returns

Mean-field type name.

property n_electron: int

Return the total number of electrons in the active space.

Returns

Total number of electrons.

property n_orb: int

Return the number of spatial orbitals.

Returns

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 mcsdf.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.

set_checkfile(chkfile, init_guess=True)
    Set checkpoint file name.

    The PySCF calculation results will be saved to the checkpoint file.

    Parameters
        • chkfile (str) – name of checkpoint file.
        • init_guess (bool, default: True) – If True and the checkpoint file exists, the initial guess will be read from the checkpoint file.

set_diis_space_dimension(diis_space_dimension=8)
    Set number of DIIS vectors.

    Parameters
        diis_space_dimension (int, default: 8) – dimension of the DIIS space.

set_init_orbitals(init_orbs)
    Sets the initial guess orbitals for the SCF.

    Parameters
        init_orbs (ndarray[TypeVar(_ScalarType_co, bound=generic, covariant=True)]) – Initial orbital coefficients.

    Return type
        None
```

`set_level_shift (level_shift_value=0)`

Set value of the artificial shift applied to virtual orbitals during SCF.

Note

Level shifting in PySCF changes the HF energy, even though the last SCF iteration is performed without it.

Parameters

`level_shift_value (float, default: 0)` – value of the level shift parameter.

`set_max_scf_cycles (max_cycles=50)`

Set maximum number of SCF cycles.

Parameters

`max_cycles (int, default: 50)` – maximum number of SCF cycles.

```
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,
                                         df=False)
```

Projection-based embedding. Partially based on PsiEmbed.

Implements Projection-based embedding method. Runs a RHF (if *functional* is *None*) or RKS calculation on the whole system and then creates an embedding, the subsystem is defined by the orbitals selected with *frozen*. The embedded calculation is run as RHF.

Parameters

- `geometry (Union[List, str, Geometry], default: None)` – Molecular geometry.
- `zmatrix (str, default: None)` – Z matrix representation of molecular geometry (Used only if *geometry* is not specified).
- `basis (Any, default: None)` – Atomic basis set valid for Mole class.
- `ecp (Any, default: None)` – Effective core potentials.
- `charge (int, default: 0)` – Total charge.
- `frozen (Union[List[int], Callable[[RHF], int]], default: None)` – Frozen orbital information.
- `transf (Transf, default: None)` – Orbital transformer.
- `verbose (int, default: 0)` – Control PySCF verbosity.
- `output (str, default: None)` – Specify log file name. If None, logs are printed to STDOUT.
- `point_group_symmetry (bool, default: False)` – Enable point group symmetry.
- `functional (str, default: "b3lyp")` – KS functional to use for the system calculation, or None if RHF is desired.
- `transf_inner (Union[Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], Transf], default: None)` – Orbital transformer to be used on the sub-system Hamiltonian.

- **frozen_inner** (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`) – List of frozen orbitals in the sub-system Hamiltonian.
- **level_shift** (`float`, default: `1.0e6`) – Controls the projection-based embedding.
- **df** (`bool`, default: `False`) – Use density fitting.

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** (default: `True`) – Reduce infinite point groups, e.g., C_{oo}v -> C₂v.

Returns

`Tuple[str, List[str]]` – Point group symmetry, list of orbital irreps.

i 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', 'Biu', 'Biu'])
```

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]]]]]`, default: `None`) – Frozen core specified as either list or callable.
- **transf** (`Union[Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], Transf]`, default: `None`) – Orbital transformation function.
- **level_shift** (`float`, default: `1.0e6`) – Projection-based embedding level shift.
- **transf_inner** (`Union[Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], Transf]`, default: `None`)
- **frozen_inner** (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`)

Returns

ChemistryDriverPySCFEmbeddingRHF – PySCF driver.

property frozen: `List[int] | List[List[int]]`

Return the frozen orbital information.

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_ac0_correction(*rdms*)

Not implemented. To obtain AC0 correction to WFT-in-DFT, use `get_subsystem_driver()`.

Parameters

`rdms` (`Tuple[ndarray[Any, dtype[float]], ndarray[Any, dtype[float]]]`)

get_casci_1234pdms()

Calculate 1-, 2-, 3- and 4-PDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1-, 2-, 3- and 4-PDMs.

get_casci_12rdms()

Calculate 1-RDM and 2-RDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1- and 2-RDMs.

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_double_factorized_system(tol1=-1, tol2=None, method=DecompositionMethod.EIG, diagonalize_one_body=True, diagonalize_one_body_offset=True, combine_one_body_terms=True)

Calculate double-factorized Hamiltonian operator, Fock space, and Hartree-Fock state.

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 which is to be double-factorized, and S is a one-body energy offset 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.

First level of factorization decomposes the electron repulsion integral (ERI) tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or pivoted, incomplete Cholesky decomposition (`method='cho'`). For details about Cholesky decomposition, refer to References [1] and [2]. The second factorization is diagonalization of the V_{pq}^t matrix for each t: $V_{pq}^t = \sum_u^{N_\lambda} U_{pu}^t \lambda_u^t U_{qu}^t$.

At the first factorization stage, truncation depends on the decomposition method. For ‘eig’, we discard eigenvalues, starting from the smallest, until the sum of those discarded exceeds `tol1`. For ‘cho’, the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr}\omega_r W_{qr}$. 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 decomposition of ERI matrix. If negative, no truncation is performed.
- **tol2** (`Optional[float]`, default: `None`) – Truncation threshold for second decomposition. If `None`, same as `tol1`.
- **method** (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`) – Decomposition method used for the first level of factorization. ‘eig’ for an eigenvalue decomposition, ‘cho’ for a pivoted, incomplete Cholesky decomposition.
- **diagonalize_one_body** (`bool`, default: `True`) – Whether to diagonalize the physical one-body integrals h_{pq} .
- **diagonalize_one_body_offset** (`bool`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq} .
- **combine_one_body_terms** (`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

`Tuple[DoubleFactorizedHamiltonian, FermionSpace, FermionState]` – Hamiltonian operator storing two-body integrals in double factorized form, Fock space, Fock state.

ⓘ References

1. Henrik Koch, Alfredo Sánchez de Merás, and Thomas Bondo Pedersen. Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.*, 118(21): 9481-9484, 2003. URL: <https://doi.org/10.1063/1.1578621>, doi:10.1063/1.1578621
2. Evgeny Epifanovsky, Dmitry Zuev, Xintian Feng, Kirill Khistyayev, Yang Shao, and Anna I. Krylov. General implementation of the resolution-of-the-identity and Cholesky representations of electron repulsion integrals within coupled-cluster and equation-of-motion methods: Theory and benchmarks. *J. Chem. Phys.*, 139(13): 134105, 2013. URL: <https://doi.org/10.1063/1.4820484>, doi:10.1063/1.4820484

`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 single-electron excitation amplitudes.

- **t2** (`Optional[ndarray]`, default: `None`) – Guess two-electron 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 single-electron excitation amplitudes.
- **t2** (`Optional[ndarray]`, default: `None`) – Guess two-electron excitation amplitudes.
- **compact** (`bool`, default: `False`) – Enable compact form of excitation for the restricted wavefunctions.
- **antihermitian** (`bool`, default: `True`) – Returns anti-Hermitian operators $T - T^\dagger$ if specified.

Returns

`FermionOperatorList` – Excitation operator list.

get_lowdin_system (`method='lowdin', store_ao=False`)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

- **method** (`str`, default: `"lowdin"`) – Method passed to PySCF's `orth.orth_ao()`.
- **store_ao** (`bool`, default: `False`) – If True, the returned Hamiltonian operator is of type `PySCFChemistryRestrictedIntegralOperator` or `PySCFChemistryUnrestrictedIntegralOperator`, which stores the underlying atomic orbitals internally.

Returns

`Tuple[Union[ChemistryRestrictedIntegralOperator, ChemistryUnrestrictedIntegralOperator, PySCFChemistryRestrictedIntegralOperator, PySCFChemistryUnrestrictedIntegralOperator], FermionSpace, Union[RestrictedOneBodyRDM, UnrestrictedOneBodyRDM]]` – Fermion Hamiltonian, Fock space, Fock state.

 **See also**

`orth.orth_ao` documentation.

get_mulliken_pop()

Interface for Mulliken population analysis of PySCF.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction(*rdms*)

Not implemented. To obtain NEVPT2 correction to WFT-in-DFT, use `get_subsystem_driver()`.

Parameters

rdms (`Tuple`)

get_orbital_coefficients()

Returns orbital coefficients.

Returns

`ndarray` – Orbital coefficients.

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_co, bound= generic, covariant=True)]]` – Two-body reduced density matrix.

Note

This object will be replaced with an `RestrictedTwoBodyRDM` class to avoid returning a raw 4D tensor.

get_subsystem_driver(*frozen=None, transf=None*)

Generate a driver object wrapping the current active space Hamiltonian.

Parameters

- **frozen** (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`) – Frozen orbital information (applies to the subsystem driver).
- **transf** (`Union[Callable[[array], array], Transf]`, default: `None`) – Orbital transformer (applies to the subsystem driver).

Returns

`BasePySCFDriver` – PySCF driver object wrapping the current active space Hamiltonian.

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.

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.

make_actives_contiguous ()

Reorder orbitals so that active orbitals form a contiguous block.

Reorders orbital coefficients, orbital energies, occupations and the list of frozen (active) orbitals. Implemented for RHF, ROHF and UHF.

property mf_energy: float

Return the total mean-field energy.

Returns

Total mean-field energy.

property mf_type: str

Return the mean-field type as a string, with options including "RHF", "ROHF", and "UHF".

Returns

Mean-field type name.

property n_electron: int

Return the total number of electrons in the active space.

Returns

Total number of electrons.

property n_orb: int

Return the number of spatial orbitals.

Returns

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 `mcsfc.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.

`set_checkfile`(*chkfile*, *init_guess=True*)

Set checkpoint file name.

The PySCF calculation results will be saved to the checkpoint file.

Parameters

- `chkfile` (`str`) – name of checkpoint file.
- `init_guess` (`bool`, default: `True`) – If `True` and the checkpoint file exists, the initial guess will be read from the checkpoint file.

`set_diis_space_dimension`(*diis_space_dimension=8*)

Set number of DIIS vectors.

Parameters

`diis_space_dimension` (`int`, default: 8) – dimension of the DIIS space.

`set_init_orbitals`(*init_orbs*)

Sets the initial guess orbitals for the SCF.

Parameters

`init_orbs` (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]`) – Initial orbital coefficients.

Return type

`None`

`set_level_shift`(*level_shift_value=0*)

Set value of the artificial shift applied to virtual orbitals during SCF.

 **Note**

Level shifting in PySCF changes the HF energy, even though the last SCF iteration is performed without it.

Parameters

- level_shift_value** (`float`, default: 0) – value of the level shift parameter.

set_max_scf_cycles (`max_cycles=50`)

Set maximum number of SCF cycles.

Parameters

- max_cycles** (`int`, default: 50) – maximum number of SCF cycles.

```
class ChemistryDriverPySCFEmbeddingROHF(chemistry=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='b3lyp5', transf_inner=None,
                                         frozen_inner=None, level_shift=1.0e6, df=False)
```

Projection-based embedding. Partially based on PsiEmbed.

Implements Projection-based embedding method. Runs a ROHF (if `functional` is `None`) or ROKS calculation on the whole system and then creates an embedding, the subsystem is defined by the orbitals selected with `frozen`. The embedded calculation is run as ROHF. The Hamiltonian operator returned is restricted (the embedding potential is averaged over spin channels).

Parameters

- **geometry** (`Union[List, str, Geometry]`, default: `None`) – Molecular geometry.
- **zmatrix** (`str`, default: `None`) – Z matrix representation of molecular geometry (Used only if `geometry` is not specified).
- **basis** (`Any`, default: `None`) – Atomic basis set valid for `Mole` class.
- **ecp** (`Any`, default: `None`) – Effective core potentials.
- **charge** (`int`, default: 0) – Total charge.
- **multiplicity** (`int`, default: 1) – Spin multiplicity of the total system
- **frozen** (`Any`, default: `None`) – Frozen orbital information.
- **transf** (`Transf`, default: `None`) – Orbital transformer.
- **verbose** (`int`, default: 0) – Control PySCF verbosity.
- **output** (`str`, default: `None`) – Specify log file name. If `None`, logs are printed to STDOUT.
- **point_group_symmetry** (`bool`, default: `False`) – Enable point group symmetry.
- **embedded_spin** (`int`, default: `None`) – number of unpaired electrons in the sub-system. `None` means the same as in the total system.
- **functional** (`str`, default: "b3lyp5") – KS functional to use for the system calculation, or `None` if RHF is desired.
- **transf_inner** (`Union[Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]], Transf]`, default: `None`) – Orbital transformer to be used on the sub-system Hamiltonian.
- **frozen_inner** (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`) – List of frozen orbitals in the sub-system Hamiltonian.
- **level_shift** (`float`, default: `1.0e6`) – Controls the projection-based embedding.
- **df** (`bool`, default: `False`) – Use density fitting.

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 (default: True)` – 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[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], Transf], default: None) – Orbital transformation function (to be used on the ROKS input).
- **transf_inner** (Union[Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], 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.

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_ac0_correction(rdns)

Not implemented. To obtain AC0 correction to WFT-in-DFT, use get_subsystem_driver().

Parameters

rdms (Tuple[ndarray[Any, dtype[float]], ndarray[Any, dtype[float]]])

get_casci_1234pdms()

Calculate 1-, 2-, 3- and 4-PDM from a CASCI wavefunction.

Returns

Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]] – A tuple of 1-, 2-, 3- and 4-PDMs.

get_casci_12rdms()

Calculate 1-RDM and 2-RDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1- and 2-RDMs.

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_double_factorized_system(*tol1*=-1, *tol2*=None, *method*=`DecompositionMethod.EIG`, *diagonalize_one_body*=True, *diagonalize_one_body_offset*=True, *combine_one_body_terms*=True)

Calculate double-factorized Hamiltonian operator, Fock space, and Hartree-Fock state.

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 which is to be double-factorized, and S is a one-body energy offset 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.

First level of factorization decomposes the electron repulsion integral (ERI) tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or pivoted, incomplete Cholesky decomposition (`method='cho'`). For details about Cholesky decomposition, refer to References [1] and [2]. The second factorization is diagonalization of the V_{pq}^t matrix for each t: $V_{pq}^t = \sum_u^{N_\lambda} U_{pu}^t \lambda_u^t U_{qu}^t$.

At the first factorization stage, truncation depends on the decomposition method. For ‘eig’, we discard eigenvalues, starting from the smallest, until the sum of those discarded exceeds `tol1`. For ‘cho’, the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. 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 decomposition of ERI matrix. If negative, no truncation is performed.
- `tol2` (`Optional[float]`, default: `None`) – Truncation threshold for second decomposition. If `None`, same as `tol1`.
- `method` (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`) – Decomposition method used for the first level of factorization. ‘eig’ for an eigenvalue decomposition, ‘cho’ for a pivoted, incomplete Cholesky decomposition.
- `diagonalize_one_body` (`bool`, default: `True`) – Whether to diagonalize the physical one-body integrals h_{pq} .
- `diagonalize_one_body_offset` (`bool`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq} .
- `combine_one_body_terms` (`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

`Tuple[DoubleFactorizedHamiltonian, FermionSpace, FermionState]` – Hamiltonian operator storing two-body integrals in double factorized form, Fock space, Fock state.

References

1. Henrik Koch, Alfredo Sánchez de Merás, and Thomas Bondo Pedersen. Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.*, 118(21): 9481-9484, 2003. URL: <https://doi.org/10.1063/1.1578621>, doi:10.1063/1.1578621
2. Evgeny Epifanovsky, Dmitry Zuev, Xintian Feng, Kirill Khistyayev, Yang Shao, and Anna I. Krylov. General implementation of the resolution-of-the-identity and Cholesky representations of electron repulsion integrals within coupled-cluster and equation-of-motion methods: Theory and benchmarks. *J. Chem. Phys.*, 139(13): 134105, 2013. URL: <https://doi.org/10.1063/1.4820484>, doi:10.1063/1.4820484

`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 single-electron excitation amplitudes.
- `t2` (`Optional[ndarray]`, default: `None`) – Guess two-electron 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 single-electron excitation amplitudes.
- **t2** (`Optional[ndarray]`, default: `None`) – Guess two-electron excitation amplitudes.
- **compact** (`bool`, default: `False`) – Enable compact form of excitation for the restricted wavefunctions.
- **antihermitian** (`bool`, default: `True`) – Returns anti-Hermitian operators $T - T^\dagger$ if specified.

Returns

`FermionOperatorList` – Excitation operator list.

`get_lowdin_system(method='lowdin', store_ao=False)`

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

- **method** (`str`, default: "lowdin") – Method passed to PySCF's `orth.orth_ao()`.
- **store_ao** (`bool`, default: `False`) – If True, the returned Hamiltonian operator is of type `PySCFChemistryRestrictedIntegralOperator` or `PySCFChemistryUnrestrictedIntegralOperator`, which stores the underlying atomic orbitals internally.

Returns

`Tuple[Union[ChemistryRestrictedIntegralOperator, ChemistryUnrestrictedIntegralOperator, PySCFChemistryRestrictedIntegralOperator, PySCFChemistryUnrestrictedIntegralOperator], FermionSpace, Union[RestrictedOneBodyRDM, UnrestrictedOneBodyRDM]]` – Fermion Hamiltonian, Fock space, Fock state.

 **See also**

`orth.orth_ao` documentation.

`get_mulliken_pop()`

Interface for Mulliken population analysis of PySCF.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction(*rdms*)

Not implemented. To obtain NEVPT2 correction to WFT-in-DFT, use `get_subsystem_driver()`.

Parameters

rdms (`Tuple`)

get_orbital_coefficients()

Returns orbital coefficients.

Returns

`ndarray` – Orbital coefficients.

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_co, bound= generic, covariant=True)]]` – Two-body reduced density matrix.

Note

This object will be replaced with an `RestrictedTwoBodyRDM` class to avoid returning a raw 4D tensor.

get_subsystem_driver(*frozen=None, transf=None*)

Generate a driver object wrapping the current active space Hamiltonian.

Parameters

- **frozen** (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`) – Frozen orbital information (applies to the subsystem driver).
- **transf** (`Union[Callable[[array], array], Transf]`, default: `None`) – Orbital transformer (applies to the subsystem driver).

Returns

`BasePySCFDriver` – PySCF driver object wrapping the current active space Hamiltonian.

get_system(*symmetry=I*)

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.

Returns

`Tuple[Union[ChemistryRestrictedIntegralOperator, ChemistryRestrictedIntegralOperatorCompact], 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` 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.

make_actives_contiguous ()

Reorder orbitals so that active orbitals form a contiguous block.

Reorders orbital coefficients, orbital energies, occupations and the list of frozen (active) orbitals. Implemented for RHF, ROHF and UHF.

property mf_energy: float

Return the total mean-field energy.

Returns

Total mean-field energy.

property mf_type: str

Return the mean-field type as a string, with options including "RHF", "ROHF", and "UHF".

Returns

Mean-field type name.

property n_electron: int

Return the total number of electrons in the active space.

Returns

Total number of electrons.

property n_orb: int

Return the number of spatial orbitals.

Returns

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 `mcsfc.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.

`set_checkfile`(*chkfile*, *init_guess=True*)

Set checkpoint file name.

The PySCF calculation results will be saved to the checkpoint file.

Parameters

- `chkfile` (`str`) – name of checkpoint file.
- `init_guess` (`bool`, default: `True`) – If `True` and the checkpoint file exists, the initial guess will be read from the checkpoint file.

`set_diis_space_dimension`(*diis_space_dimension=8*)

Set number of DIIS vectors.

Parameters

`diis_space_dimension` (`int`, default: 8) – dimension of the DIIS space.

`set_init_orbitals`(*init_orbs*)

Sets the initial guess orbitals for the SCF.

Parameters

`init_orbs` (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]`) – Initial orbital coefficients.

Return type

`None`

`set_level_shift`(*level_shift_value=0*)

Set value of the artificial shift applied to virtual orbitals during SCF.

 **Note**

Level shifting in PySCF changes the HF energy, even though the last SCF iteration is performed without it.

Parameters

level_shift_value (`float`, default: 0) – value of the level shift parameter.

set_max_scf_cycles (`max_cycles=50`)

Set maximum number of SCF cycles.

Parameters

max_cycles (`int`, default: 50) – maximum number of SCF cycles.

```
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='b3lyp5',
                                              transf_inner=None, frozen_inner=None, level_shift=1.0e6,
                                              df=False)
```

Projective embedding. Partially based on PsiEmbed.

Implements Projection-based embedding method. Runs a ROHF (if `functional` is `None`) or ROKS calculation on the whole system and then creates an embedding, the subsystem is defined by the orbitals selected with `frozen`. The embedded calculation is run as ROHF and subsequently converted to UHF. The Hamiltonian operator returned is unrestricted.

Parameters

- **geometry** (`Union[List, str, Geometry]`, default: `None`) – Molecular geometry.
- **zmatrix** (`str`, default: `None`) – Z matrix representation of molecular geometry (Used only if `geometry` is not specified).
- **basis** (`Any`, default: `None`) – Atomic basis set valid for `Mole` class.
- **ecp** (`Any`, default: `None`) – Effective core potentials.
- **charge** (`int`, default: 0) – Total charge.
- **multiplicity** (`int`, default: 1) – Spin multiplicity of the total system
- **frozen** (`Any`, default: `None`) – Frozen orbital information.
- **transf** (`Transf`, default: `None`) – Orbital transformer.
- **verbose** (`int`, default: 0) – Control PySCF verbosity.
- **output** (`str`, default: `None`) – Specify log file name. If `None`, logs are printed to `STDOUT`.
- **point_group_symmetry** (`bool`, default: `False`) – Enable point group symmetry.
- **embedded_spin** (`int`, default: `None`) – number of unpaired electrons in the sub-system. `None` means the same as in the total system.
- **functional** (`str`, default: "b3lyp5") – KS functional to use for the system calculation, or `None` if RHF is desired.
- **transf_inner** (`Union[Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], Transf]`, default: `None`) – Orbital transformer to be used on the sub-system Hamiltonian.
- **frozen_inner** (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`) – List of frozen orbitals in the sub-system Hamiltonian.
- **level_shift** (`float`, default: `1.0e6`) – Controls the projection-based embedding.

- `df` (`bool`, default: `False`) – Use density fitting.

`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` (default: `True`) – Reduce infinite point groups, e.g., Coov -> C2v.

Returns

`Tuple[str, List[str]]` – Point group symmetry, list of orbital irreps.

➊ 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[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], Transf], default: None) – Orbital transformation function (to be used on the ROKS input).
- **transf_inner**(Union[Callable[[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]], 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.

```
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_ac0_correction(rdms)
```

Not implemented. To obtain AC0 correction to WFT-in-DFT, use get_subsystem_driver().

Parameters

rdms (Tuple[ndarray[Any, dtype[float]], ndarray[Any, dtype[float]]])

```
get_casci_1234pdms()
```

Calculate 1-, 2-, 3- and 4-PDM from a CASCI wavefunction.

Returns

Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]] – A tuple of 1-, 2-, 3- and 4-PDMs.

```
get_casci_12rdms()
```

Calculate 1-RDM and 2-RDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1- and 2-RDMs.

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_double_factorized_system(*tol1*=-1, *tol2*=None, *method*=`DecompositionMethod.EIG`, *diagonalize_one_body*=True, *diagonalize_one_body_offset*=True, *combine_one_body_terms*=True)

Calculate double-factorized Hamiltonian operator, Fock space, and Hartree-Fock state.

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 which is to be double-factorized, and S is a one-body energy offset 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.

First level of factorization decomposes the electron repulsion integral (ERI) tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or pivoted, incomplete Cholesky decomposition (`method='cho'`). For details about Cholesky decomposition, refer to References [1] and [2]. 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 the first factorization stage, truncation depends on the decomposition method. For ‘eig’, we discard eigenvalues, starting from the smallest, until the sum of those discarded exceeds `tol1`. For ‘cho’, the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. 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 decomposition of ERI matrix. If negative, no truncation is performed.
- `tol2` (`Optional[float]`, default: `None`) – Truncation threshold for second decomposition. If `None`, same as `tol1`.
- `method` (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`) – Decomposition method used for the first level of factorization. ‘eig’ for an eigenvalue decomposition, ‘cho’ for a pivoted, incomplete Cholesky decomposition.
- `diagonalize_one_body` (`bool`, default: `True`) – Whether to diagonalize the physical one-body integrals h_{pq} .
- `diagonalize_one_body_offset` (`bool`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq} .
- `combine_one_body_terms` (`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

`Tuple[DoubleFactorizedHamiltonian, FermionSpace, FermionState]` – Hamiltonian operator storing two-body integrals in double factorized form, Fock space, Fock state.

References

1. Henrik Koch, Alfredo Sánchez de Merás, and Thomas Bondo Pedersen. Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.*, 118(21): 9481-9484, 2003. URL: <https://doi.org/10.1063/1.1578621>, doi:10.1063/1.1578621
2. Evgeny Epifanovsky, Dmitry Zuev, Xintian Feng, Kirill Khistyayev, Yang Shao, and Anna I. Krylov. General implementation of the resolution-of-the-identity and Cholesky representations of electron repulsion integrals within coupled-cluster and equation-of-motion methods: Theory and benchmarks. *J. Chem. Phys.*, 139(13): 134105, 2013. URL: <https://doi.org/10.1063/1.4820484>, doi:10.1063/1.4820484

`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 single-electron excitation amplitudes.
- `t2` (`Optional[ndarray]`, default: `None`) – Guess two-electron 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 single-electron excitation amplitudes.
- **t2** (`Optional[ndarray]`, default: `None`) – Guess two-electron excitation amplitudes.
- **compact** (`bool`, default: `False`) – Enable compact form of excitation for the restricted wavefunctions.
- **antihermitian** (`bool`, default: `True`) – Returns anti-Hermitian operators $T - T^\dagger$ if specified.

Returns

`FermionOperatorList` – Excitation operator list.

`get_lowdin_system(method='lowdin', store_ao=False)`

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

- **method** (`str`, default: "lowdin") – Method passed to PySCF's `orth.orth_ao()`.
- **store_ao** (`bool`, default: `False`) – If True, the returned Hamiltonian operator is of type `PySCFChemistryRestrictedIntegralOperator` or `PySCFChemistryUnrestrictedIntegralOperator`, which stores the underlying atomic orbitals internally.

Returns

`Tuple[Union[ChemistryRestrictedIntegralOperator, ChemistryUnrestrictedIntegralOperator, PySCFChemistryRestrictedIntegralOperator, PySCFChemistryUnrestrictedIntegralOperator], FermionSpace, Union[RestrictedOneBodyRDM, UnrestrictedOneBodyRDM]]` – Fermion Hamiltonian, Fock space, Fock state.

 **See also**

[orth.orth_ao documentation](#).

`get_mulliken_pop()`

Interface for Mulliken population analysis of PySCF.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction(rdm)

Not implemented. To obtain NEVPT2 correction to WFT-in-DFT, use get_subsystem_driver().

Parameters

rdms (`Tuple`)

get_orbital_coefficients()

Returns orbital coefficients.

Returns

`ndarray` – Orbital coefficients.

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_co, bound= generic, covariant=True)]]` – Two-body reduced density matrix.

Note

This object will be replaced with an `RestrictedTwoBodyRDM` class to avoid returning a raw 4D tensor.

get_subsystem_driver(frozen=None, transf=None)

Generate a driver object wrapping the current active space Hamiltonian.

Parameters

- **frozen** (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`) – Frozen orbital information (applies to the subsystem driver).
- **transf** (`Union[Callable[[array], array], Transf]`, default: `None`) – Orbital transformer (applies to the subsystem driver).

Returns

`BasePySCFDriver` – PySCF driver object wrapping the current active space Hamiltonian.

get_system(symmetry=1)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Parameters

symmetry (`Union[str, int]`, default: 1)

Return type

`Tuple[Union[ChemistryUnrestrictedIntegralOperator, ChemistryUnrestrictedIntegralOperatorCompact], FermionSpace, FermionState]`

get_system_ao(run_hf=True)

Calculate fermionic Hamiltonian operator, Fock space, and Hartree Fock state.

Output hamiltonian operator will be of type `PySCFChemistryRestrictedIntegralOperator` 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.

`make_actives_contiguous()`

Reorder orbitals so that active orbitals form a contiguous block.

Reorders orbital coefficients, orbital energies, occupations and the list of frozen (active) orbitals. Implemented for RHF, ROHF and UHF.

`property mf_energy: float`

Return the total mean-field energy.

Returns

Total mean-field energy.

`property mf_type: str`

Return the mean-field type as a string, with options including "RHF", "ROHF", and "UHF".

Returns

Mean-field type name.

`property n_electron: int`

Return the total number of electrons in the active space.

Returns

Total number of electrons.

`property n_orb: int`

Return the number of spatial orbitals.

Returns

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 `mcsf.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.

set_checkfile(chkfile, init_guess=True)

Set checkpoint file name.

The PySCF calculation results will be saved to the checkpoint file.

Parameters

- `chkfile` (`str`) – name of checkpoint file.
- `init_guess` (`bool`, default: `True`) – If `True` and the checkpoint file exists, the initial guess will be read from the checkpoint file.

set_diis_space_dimension(diis_space_dimension=8)

Set number of DIIS vectors.

Parameters

`diis_space_dimension` (`int`, default: 8) – dimension of the DIIS space.

set_init_orbitals(init_orbs)

Sets the initial guess orbitals for the SCF.

Parameters

`init_orbs` (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]`) – Initial orbital coefficients.

Return type

`None`

set_level_shift(level_shift_value=0)

Set value of the artificial shift applied to virtual orbitals during SCF.

Note

Level shifting in PySCF changes the HF energy, even though the last SCF iteration is performed without it.

Parameters

`level_shift_value` (`float`, default: 0) – value of the level shift parameter.

```
set_max_scf_cycles(max_cycles=50)
```

Set maximum number of SCF cycles.

Parameters

- **max_cycles** (`int`, default: 50) – maximum number of SCF cycles.

```
class ChemistryDriverPySCFGammaRHF(geometry=None, cell=None, basis=None, ecp=None, pseudo=None, charge=0, exp_to_discard=None, frozen=None, transf=None, dimension=3, output=None, space_group_symmetry=False, precision=1e-9, verbose=0, soscf=False, df='GDF')
```

PySCF driver for Gamma-point RHF calculations.

Parameters

- **geometry** (`Union[List, str, GeometryPeriodic]`, default: `None`) – Molecular geometry.
- **cell** (`ndarray`, default: `None`) – Unit cell parameter. If provided, overrides `geometry`. `unit_cell`.
- **basis** (`Any`, default: `None`) – Atomic basis set valid for `Mole` class.
- **ecp** (`Any`, default: `None`) – Effective core potentials.
- **pseudo** (`Any`, default: `None`) – Pseudo potentials.
- **charge** (`int`, default: 0) – Total charge.
- **exp_to_discard** (`Optional[float]`, default: `None`) – Exponent to discard a primitive Gaussian.
- **frozen** (`Union[List[int], Callable[[RHF], List[int]]]`, default: `None`) – Frozen orbital information.
- **transf** (`Transf`, default: `None`) – Orbital transformer.
- **dimension** (`int`, default: 3) – Number of spatial dimensions.
- **output** (`Optional[str]`, default: `None`) – Specify log file name. If `None`, logs are printed to `STDOUT`.
- **space_group_symmetry** (`bool`, default: `False`) – Whether to use space group symmetry.
- **precision** (`float`, default: `1e-9`) – Ewald sum precision.
- **verbose** (`int`, default: 0) – Control PySCF verbosity.
- **soscf** (`bool`, default: `False`) – Use Second-Order SCF solver (Newton's method).
- **df** (`str`, default: "GDF") – Density fitting function name.

```
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** (default: `True`) – Reduce infinite point groups, e.g., Coov -> C2v.

Returns

`Tuple[str, List[str]]` – Point group symmetry, list of orbital irreps.

❶ 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', 'Biu', 'Biu'])
```

`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]]]]]`, default: `None`) – Frozen core specified as either list or callable.
- **transf** (`Union[Callable[[ndarray], ndarray], Transf]`, default: `None`) – Orbital transformation function.

Returns

`ChemistryDriverPySCFGammaRHF` – PySCF driver.

`property frozen: List[int] | List[List[int]]`

Return the frozen orbital information.

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_ac0_correction (rdms)

Compute the AC0 correction to the energy from the provided density matrices.

Requires installation of `pyscf-ac0` extension, which is partially based on `GAMMCOR` and is distributed separately.

Parameters

`rdms` (`Tuple[ndarray[Any, dtype[float]], ndarray[Any, dtype[float]]]`) – A tuple of one- and two-particle reduced density matrices as RDMs as `(rdm1, rdm2)`.

Returns

The AC0 correction to the energy.

 **See also**

For more information on AC0 correction, refer to: - DOI: 10.1021/acs.jctc.8b00213

get_casci_1234pdms ()

Calculate 1-, 2-, 3- and 4-PDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1-, 2-, 3- and 4-PDMs.

get_casci_12rdms ()

Calculate 1-RDM and 2-RDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1- and 2-RDMs.

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_double_factorized_system(tol1=-1, tol2=None, method=DecompositionMethod.EIG,
                             diagonalize_one_body=True, diagonalize_one_body_offset=True,
                             combine_one_body_terms=True)
```

Calculate double-factorized Hamiltonian operator, Fock space, and Hartree-Fock state.

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 which is to be double-factorized, and S is a one-body energy offset 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.

First level of factorization decomposes the electron repulsion integral (ERI) tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or pivoted, incomplete Cholesky decomposition (`method='cho'`). For details about Cholesky decomposition, refer to References [1] and [2]. The second factorization is diagonalization of the V_{pq}^t matrix for each t: $V_{pq}^t = \sum_u^{N_\lambda} U_{pu}^t \lambda_u^t U_{qu}^t$.

At the first factorization stage, truncation depends on the decomposition method. For ‘eig’, we discard eigenvalues, starting from the smallest, until the sum of those discarded exceeds `tol1`. For ‘cho’, the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. 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 decomposition of ERI matrix. If negative, no truncation is performed.
- **tol2** (`Optional[float]`, default: `None`) – Truncation threshold for second decomposition. If `None`, same as `tol1`.
- **method** (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`) – Decomposition method used for the first level of factorization. ‘eig’ for an eigenvalue decomposition, ‘cho’ for a pivoted, incomplete Cholesky decomposition.
- **diagonalize_one_body** (`bool`, default: `True`) – Whether to diagonalize the physical one-body integrals h_{pq} .
- **diagonalize_one_body_offset** (`bool`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq} .

- `combine_one_body_terms` (`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

`Tuple[DoubleFactorizedHamiltonian, FermionSpace, FermionState]` – Hamiltonian operator storing two-body integrals in double factorized form, Fock space, Fock state.

References

1. Henrik Koch, Alfredo Sánchez de Merás, and Thomas Bondo Pedersen. Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.*, 118(21): 9481-9484, 2003. URL: <https://doi.org/10.1063/1.1578621>, doi:10.1063/1.1578621
2. Evgeny Epifanovsky, Dmitry Zuev, Xintian Feng, Kirill Khistyayev, Yang Shao, and Anna I. Krylov. General implementation of the resolution-of-the-identity and Cholesky representations of electron repulsion integrals within coupled-cluster and equation-of-motion methods: Theory and benchmarks. *J. Chem. Phys.*, 139(13): 134105, 2013. URL: <https://doi.org/10.1063/1.4820484>, doi:10.1063/1.4820484

`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 single-electron excitation amplitudes.
- `t2` (`Optional[ndarray]`, default: `None`) – Guess two-electron 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 single-electron excitation amplitudes.
- `t2` (`Optional[ndarray]`, default: `None`) – Guess two-electron excitation amplitudes.
- `compact` (`bool`, default: `False`) – Enable compact form of excitation for the restricted wavefunctions.
- `antihermitian` (`bool`, default: `True`) – Returns anti-Hermitian operators $T - T^\dagger$ if specified.

Returns

`FermionOperatorList` – Excitation operator list.

get_lowdin_system(*method='lowdin'*, *store_ao=False*)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

- **method** (`str`, default: "lowdin") – Method passed to PySCF's `orth.orth_ao()`.
- **store_ao** (`bool`, default: False) – If True, the returned Hamiltonian operator is of type `PySCFChemistryRestrictedIntegralOperator` or `PySCFChemistryUnrestrictedIntegralOperator`, which stores the underlying atomic orbitals internally.

Returns

`Tuple[Union[ChemistryRestrictedIntegralOperator, ChemistryUnrestrictedIntegralOperator, PySCFChemistryRestrictedIntegralOperator, PySCFChemistryUnrestrictedIntegralOperator], FermionSpace, Union[RestrictedOneBodyRDM, UnrestrictedOneBodyRDM]]` – Fermion Hamiltonian, Fock space, Fock state.

↳ **See also**

`orth.orth_ao` documentation.

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_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, 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, ..., pdm4).

Returns

The NEVPT2 correction to the energy.

get_orbital_coefficients()

Returns orbital coefficients.

Returns

`ndarray` – Orbital coefficients.

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_co, bound= generic, covariant=True)]]` – Two-body reduced density matrix.

Note

This object will be replaced with an `RestrictedTwoBodyRDM` class to avoid returning a raw 4D tensor.

get_subsystem_driver(frozen=None, transf=None)

Generate a driver object wrapping the current active space Hamiltonian.

Parameters

- **frozen** (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`) – Frozen orbital information (applies to the subsystem driver).
- **transf** (`Union[Callable[[array], array], Transf]`, default: `None`) – Orbital transformer (applies to the subsystem driver).

Returns

`BasePySCFDriver` – PySCF driver object wrapping the current active space Hamiltonian.

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.

`make_actives_contiguous()`

Reorder orbitals so that active orbitals form a contiguous block.

Reorders orbital coefficients, orbital energies, occupations and the list of frozen (active) orbitals. Implemented for RHF, ROHF and UHF.

`property mf_energy: float`

Return the total mean-field energy.

Returns

Total mean-field energy.

`property mf_type: str`

Return the mean-field type as a string, with options including "RHF", "ROHF", and "UHF".

Returns

Mean-field type name.

`property n_electron: int`

Return the total number of electrons in the active space.

Returns

Total number of electrons.

`property n_orb: int`

Return the number of spatial orbitals.

Returns

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 `mcsf.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.

set_checkfile(chkfile, init_guess=True)

Set checkpoint file name.

The PySCF calculation results will be saved to the checkpoint file.

Parameters

- `chkfile` (`str`) – name of checkpoint file.
- `init_guess` (`bool`, default: `True`) – If `True` and the checkpoint file exists, the initial guess will be read from the checkpoint file.

set_diis_space_dimension(diis_space_dimension=8)

Set number of DIIS vectors.

Parameters

`diis_space_dimension` (`int`, default: 8) – dimension of the DIIS space.

set_init_orbitals(init_orbs)

Sets the initial guess orbitals for the SCF.

Parameters

`init_orbs` (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]`) – Initial orbital coefficients.

Return type

`None`

set_level_shift(level_shift_value=0)

Set value of the artificial shift applied to virtual orbitals during SCF.

Note

Level shifting in PySCF changes the HF energy, even though the last SCF iteration is performed without it.

Parameters

`level_shift_value` (`float`, default: 0) – value of the level shift parameter.

```
set_max_scf_cycles(max_cycles=50)
```

Set maximum number of SCF cycles.

Parameters

- max_cycles** (`int`, default: 50) – maximum number of SCF cycles.

```
class ChemistryDriverPySCFGammaROHF(geometry=None, cell=None, basis=None, ecp=None, pseudo=None, charge=0, multiplicity=1, exp_to_discard=None, frozen=None, transf=None, dimension=3, output=None, space_group_symmetry=False, precision=1e-9, verbose=0, soscf=False, df='GDF')
```

PySCF driver for Gamma-point ROHF calculations.

Parameters

- **geometry** (`Union[List, str, GeometryPeriodic]`, default: `None`) – Molecular geometry.
- **cell** (`ndarray`, default: `None`) – Unit cell parameter. If provided, overrides `geometry.unit_cell`.
- **basis** (`Any`, default: `None`) – Atomic basis set valid for `Mole` class.
- **ecp** (`Any`, default: `None`) – Effective core potentials.
- **pseudo** (`Any`, default: `None`) – Pseudo potentials.
- **charge** (`int`, default: 0) – Total charge.
- **multiplicity** (`int`, default: 1) – Spin multiplicity, $2S+1$.
- **exp_to_discard** (`Optional[float]`, default: `None`) – Exponent to discard a primitive Gaussian.
- **frozen** (`Union[List[int], Callable[[RHF], List[int]]]`, default: `None`) – Frozen orbital information.
- **transf** (`Transf`, default: `None`) – Orbital transformer.
- **dimension** (`int`, default: 3) – Number of spatial dimensions.
- **output** (`Optional[str]`, default: `None`) – Specify log file name. If `None`, logs are printed to `STDOUT`.
- **space_group_symmetry** (`bool`, default: `False`) – Whether to use space group symmetry.
- **precision** (`float`, default: `1e-9`) – Ewald sum precision.
- **verbose** (`int`, default: 0) – Control PySCF verbosity.
- **soscf** (`bool`, default: `False`) – Use Second-Order SCF solver (Newton's method).
- **df** (`str`, default: "GDF") – Density fitting function name.

```
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** (default: `True`) – 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]]]]]`, default: `None`) – Frozen core specified as either list or callable.
- **transf** (`Union[Callable[[ndarray], ndarray], Transf]`, default: `None`) – Orbital transformation function.

Returns

`ChemistryDriverPySCFGammaROHF` – PySCF driver.

property frozen: List[int] | List[List[int]]

Return the frozen orbital information.

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_ac0_correction(rdm)

Compute the AC0 correction to the energy from the provided density matrices.

Requires installation of `pyscf-ac0` extension, which is partially based on `GAMMCOR` and is distributed separately.

Parameters

`rdm` (`Tuple[ndarray[Any, dtype[float]], ndarray[Any, dtype[float]]]`) – A tuple of one- and two-particle reduced density matrices as RDMs as `(rdm1, rdm2)`.

Returns

The AC0 correction to the energy.

See also

For more information on AC0 correction, refer to: - DOI: 10.1021/acs.jctc.8b00213

get_casci_1234pdms()

Calculate 1-, 2-, 3- and 4-PDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1-, 2-, 3- and 4-PDMs.

get_casci_12rdms()

Calculate 1-RDM and 2-RDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1- and 2-RDMs.

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_double_factorized_system(tol1=-1, tol2=None, method=DecompositionMethod.EIG,
diagonalize_one_body=True, diagonalize_one_body_offset=True,
combine_one_body_terms=True)
```

Calculate double-factorized Hamiltonian operator, Fock space, and Hartree-Fock state.

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 which is to be double-factorized, and S is a one-body energy offset 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.

First level of factorization decomposes the electron repulsion integral (ERI) tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or pivoted, incomplete Cholesky decomposition (`method='cho'`). For details about Cholesky decomposition, refer to References [1] and [2]. The second factorization is diagonalization of the V_{pq}^t matrix for each t: $V_{pq}^t = \sum_u^{N_\lambda^t} U_{pu}^t U_{qu}^t \lambda_u^t U_{qu}^t$.

At the first factorization stage, truncation depends on the decomposition method. For ‘eig’, we discard eigenvalues, starting from the smallest, until the sum of those discarded exceeds `tol1`. For ‘cho’, the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. 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 decomposition of ERI matrix. If negative, no truncation is performed.
- **tol2** (`Optional[float]`, default: `None`) – Truncation threshold for second decomposition. If `None`, same as `tol1`.
- **method** (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`) – Decomposition method used for the first level of factorization. ‘eig’ for an eigenvalue decomposition, ‘cho’ for a pivoted, incomplete Cholesky decomposition.
- **diagonalize_one_body** (`bool`, default: `True`) – Whether to diagonalize the physical one-body integrals h_{pq} .

- `diagonalize_one_body_offset` (`bool`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq} .
- `combine_one_body_terms` (`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

`Tuple[DoubleFactorizedHamiltonian, FermionSpace, FermionState]` – Hamiltonian operator storing two-body integrals in double factorized form, Fock space, Fock state.

References

1. Henrik Koch, Alfredo Sánchez de Merás, and Thomas Bondo Pedersen. Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.*, 118(21): 9481-9484, 2003. URL: <https://doi.org/10.1063/1.1578621>, doi:10.1063/1.1578621
2. Evgeny Epifanovsky, Dmitry Zuev, Xintian Feng, Kirill Khistyayev, Yang Shao, and Anna I. Krylov. General implementation of the resolution-of-the-identity and Cholesky representations of electron repulsion integrals within coupled-cluster and equation-of-motion methods: Theory and benchmarks. *J. Chem. Phys.*, 139(13): 134105, 2013. URL: <https://doi.org/10.1063/1.4820484>, doi:10.1063/1.4820484

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 single-electron excitation amplitudes.
- `t2` (`Optional[ndarray]`, default: `None`) – Guess two-electron 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 single-electron excitation amplitudes.
- `t2` (`Optional[ndarray]`, default: `None`) – Guess two-electron excitation amplitudes.
- `compact` (`bool`, default: `False`) – Enable compact form of excitation for the restricted wavefunctions.

- **antihermitian** (`bool`, default: `True`) – Returns anti-Hermitian operators $T - T^\dagger$ if specified.

Returns

`FermionOperatorList` – Excitation operator list.

get_lowdin_system (`method='lowdin'`, `store_ao=False`)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

- **method** (`str`, default: "lowdin") – Method passed to PySCF's `orth.orth_ao()`.
- **store_ao** (`bool`, default: `False`) – If True, the returned Hamiltonian operator is of type `PySCFChemistryRestrictedIntegralOperator` or `PySCFChemistryUnrestrictedIntegralOperator`, which stores the underlying atomic orbitals internally.

Returns

`Tuple[Union[ChemistryRestrictedIntegralOperator, ChemistryUnrestrictedIntegralOperator, PySCFChemistryRestrictedIntegralOperator, PySCFChemistryUnrestrictedIntegralOperator], FermionSpace, Union[RestrictedOneBodyRDM, UnrestrictedOneBodyRDM]]` – Fermion Hamiltonian, Fock state.

↳ **See also**

`orth.orth_ao` documentation.

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_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, 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, ..., pdm4).

Returns

The NEVPT2 correction to the energy.

get_orbital_coefficients ()

Returns orbital coefficients.

Returns

`ndarray` – Orbital coefficients.

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_co, bound= generic, covariant=True)]]` – Two-body reduced density matrix.

Note

This object will be replaced with an `RestrictedTwoBodyRDM` class to avoid returning a raw 4D tensor.

get_subsystem_driver(frozen=None, transf=None)

Generate a driver object wrapping the current active space Hamiltonian.

Parameters

- **frozen** (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`) – Frozen orbital information (applies to the subsystem driver).
- **transf** (`Union[Callable[[array], array], Transf]`, default: `None`) – Orbital transformer (applies to the subsystem driver).

Returns

`BasePySCFDriver` – PySCF driver object wrapping the current active space Hamiltonian.

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.

make_actives_contiguous ()

Reorder orbitals so that active orbitals form a contiguous block.

Reorders orbital coefficients, orbital energies, occupations and the list of frozen (active) orbitals. Implemented for RHF, ROHF and UHF.

property mf_energy: float

Return the total mean-field energy.

Returns

Total mean-field energy.

property mf_type: str

Return the mean-field type as a string, with options including "RHF", "ROHF", and "UHF".

Returns

Mean-field type name.

property n_electron: int

Return the total number of electrons in the active space.

Returns

Total number of electrons.

property n_orb: int

Return the number of spatial orbitals.

Returns

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 `mcsfc.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.

`set_checkfile`(*chkfile*, *init_guess=True*)

Set checkpoint file name.

The PySCF calculation results will be saved to the checkpoint file.

Parameters

- `chkfile` (`str`) – name of checkpoint file.
- `init_guess` (`bool`, default: `True`) – If `True` and the checkpoint file exists, the initial guess will be read from the checkpoint file.

`set_diis_space_dimension`(*diis_space_dimension=8*)

Set number of DIIS vectors.

Parameters

`diis_space_dimension` (`int`, default: 8) – dimension of the DIIS space.

`set_init_orbitals`(*init_orbs*)

Sets the initial guess orbitals for the SCF.

Parameters

`init_orbs` (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]`) – Initial orbital coefficients.

Return type

`None`

`set_level_shift`(*level_shift_value=0*)

Set value of the artificial shift applied to virtual orbitals during SCF.

Note

Level shifting in PySCF changes the HF energy, even though the last SCF iteration is performed without it.

Parameters

`level_shift_value (float, default: 0)` – value of the level shift parameter.

`set_max_scf_cycles (max_cycles=50)`

Set maximum number of SCF cycles.

Parameters

`max_cycles (int, default: 50)` – maximum number of SCF cycles.

`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 (ndarray, default: None)` – Initial density matrix.
- `frozen (Union[List[int], Callable[[SCF], List[int]]], default: None)` – List of frozen orbitals.
- `transf (Union[Callable[[array], array], Transf], 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.

`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_ac0_correction(rdm)

Compute the AC0 correction to the energy from the provided density matrices.

Requires installation of `pyscf-ac0` extension, which is partially based on `GAMMCOR` and is distributed separately.

Parameters

`rdm` (`Tuple[ndarray[Any, dtype[float]], ndarray[Any, dtype[float]]]`) – A tuple of one- and two-particle reduced density matrices as RDMs as (rdm1, rdm2).

Returns

The AC0 correction to the energy.

 **See also**

For more information on AC0 correction, refer to: - DOI: 10.1021/acs.jctc.8b00213

get_cascl_1234pdms()

Calculate 1-, 2-, 3- and 4-PDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1-, 2-, 3- and 4-PDMs.

get_cascl_12rdms()

Calculate 1-RDM and 2-RDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1- and 2-RDMs.

get_double_factorized_system(tol1=-1, tol2=None, method=DecompositionMethod.EIG, diagonalize_one_body=True, diagonalize_one_body_offset=True, combine_one_body_terms=True)

Calculate double-factorized Hamiltonian operator, Fock space, and Hartree-Fock state.

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 which is to be double-factorized, and S is a one-body energy offset 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.

First level of factorization decomposes the electron repulsion integral (ERI) tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or pivoted, incomplete Cholesky decomposition (`method='cho'`). For details about Cholesky decomposition, refer to References [1] and [2]. 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 the first factorization stage, truncation depends on the decomposition method. For ‘eig’, we discard eigenvalues, starting from the smallest, until the sum of those discarded exceeds `tol1`. For ‘cho’, the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. 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 decomposition of ERI matrix. If negative, no truncation is performed.
- **tol2** (`Optional[float]`, default: `None`) – Truncation threshold for second decomposition. If `None`, same as `tol1`.
- **method** (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`) – Decomposition method used for the first level of factorization. ‘eig’ for an eigenvalue decomposition, ‘cho’ for a pivoted, incomplete Cholesky decomposition.
- **diagonalize_one_body** (`bool`, default: `True`) – Whether to diagonalize the physical one-body integrals h_{pq} .
- **diagonalize_one_body_offset** (`bool`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq} .
- **combine_one_body_terms** (`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

`Tuple[DoubleFactorizedHamiltonian, FermionSpace, FermionState]` – Hamiltonian operator storing two-body integrals in double factorized form, Fock space, Fock state.

References

1. Henrik Koch, Alfredo Sánchez de Merás, and Thomas Bondo Pedersen. Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.*, 118(21): 9481-9484, 2003. URL: <https://doi.org/10.1063/1.1578621>, doi:10.1063/1.1578621
2. Evgeny Epifanovsky, Dmitry Zuev, Xintian Feng, Kirill Khistyayev, Yang Shao, and Anna I. Krylov. General implementation of the resolution-of-the-identity and Cholesky representations of electron repulsion integrals within coupled-cluster and equation-of-motion methods: Theory and benchmarks. *J. Chem. Phys.*, 139(13): 134105, 2013. URL: <https://doi.org/10.1063/1.4820484>, doi:10.1063/1.4820484

`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 single-electron excitation amplitudes.
- **t2** (`Optional[ndarray]`, default: `None`) – Guess two-electron 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 single-electron excitation amplitudes.
- **t2** (`Optional[ndarray]`, default: `None`) – Guess two-electron excitation amplitudes.
- **compact** (`bool`, default: `False`) – Enable compact form of excitation for the restricted wavefunctions.
- **antihermitian** (`bool`, default: `True`) – Returns anti-Hermitian operators $T - T^\dagger$ if specified.

Returns

`FermionOperatorList` – Excitation operator list.

`get_lowdin_system`(`method='lowdin'`, `store_ao=False`)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

- **method** (`str`, default: "lowdin") – Method passed to PySCF's `orth.orth_ao()`.
- **store_ao** (`bool`, default: `False`) – If True, the returned Hamiltonian operator is of type `PySCFChemistryRestrictedIntegralOperator` or `PySCFChemistryUnrestrictedIntegralOperator`, which stores the underlying atomic orbitals internally.

Returns

`Tuple[Union[ChemistryRestrictedIntegralOperator, ChemistryUnrestrictedIntegralOperator, PySCFChemistryRestrictedIntegralOperator, PySCFChemistryUnrestrictedIntegralOperator], FermionSpace, Union[RestrictedOneBodyRDM, UnrestrictedOneBodyRDM]]` – Fermion Hamiltonian, Fock space, Fock state.

 See also

`orth.orth_ao` documentation.

`get_nevpt2_correction(rdm)`

Compute the Strongly contracted NEVPT2 correction to the energy from the provided density matrices.

Parameters

- **rdm** (`Tuple`) – A tuple of reduced density matrices as PDMs in PySCF's un-reordered format as (pdm1, ..., pdm4).

Returns

The NEVPT2 correction to the energy.

get_one_body_rdm()

Compute one-body restricted density 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_orbital_coefficients()

Returns orbital coefficients.

Returns

ndarray – Orbital coefficients.

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_co*, bound=*generic*, covariant=True)]] – Two-body reduced density matrix.

Note

This object will be replaced with an *RestrictedTwoBodyRDM* class to avoid returning a raw 4D tensor.

get_subsystem_driver(frozen=None, transf=None)

Generate a driver object wrapping the current active space Hamiltonian.

Parameters

- **frozen** (*Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]*, default: None) – Frozen orbital information (applies to the subsystem driver).
- **transf** (*Union[Callable[[array], array], Transf]*, default: None) – Orbital transformer (applies to the subsystem driver).

Returns

BasePySCFDriver – PySCF driver object wrapping the current active space Hamiltonian.

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.`

`make_actives_contiguous ()`

Reorder orbitals so that active orbitals form a contiguous block.

Reorders orbital coefficients, orbital energies, occupations and the list of frozen (active) orbitals. Implemented for RHF, ROHF and UHF.

`property mf_energy: float`

Return the total mean-field energy.

>Returns

Total mean-field energy.

`property mf_type: str`

Return the mean-field type as a string, with options including "RHF", "ROHF", and "UHF".

>Returns

Mean-field type name.

`property mo_coeff: ndarray[Any, dtype[_ScalarType_co]]`

Return the molecular orbital coefficient matrix.

>Returns

MO coefficients.

`property n_electron: int`

Return the total number of electrons in the active space.

Returns

Total number of electrons.

property n_orb: int

Return the number of spatial orbitals.

Returns

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 `mcsf.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.

set_checkfile(chkfile, init_guess=True)

Set checkpoint file name.

The PySCF calculation results will be saved to the checkpoint file.

Parameters

- `chkfile` (`str`) – name of checkpoint file.
- `init_guess` (`bool`, default: `True`) – If True and the checkpoint file exists, the initial guess will be read from the checkpoint file.

set_diis_space_dimension(diis_space_dimension=8)

Set number of DIIS vectors.

Parameters

`diis_space_dimension` (`int`, default: 8) – dimension of the DIIS space.

set_init_orbitals(*init_orbs*)

Sets the initial guess orbitals for the SCF.

Parameters

init_orbs (ndarray[*Any*, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]) – Initial orbital coefficients.

Return type

None

set_level_shift(*level_shift_value*=0)

Set value of the artificial shift applied to virtual orbitals during SCF.

Note

Level shifting in PySCF changes the HF energy, even though the last SCF iteration is performed without it.

Parameters

level_shift_value (*float*, default: 0) – value of the level shift parameter.

set_max_scf_cycles(*max_cycles*=50)

Set maximum number of SCF cycles.

Parameters

max_cycles (*int*, default: 50) – maximum number of SCF cycles.

```
class ChemistryDriverPySCFMolecularRHF(geometery=None, zmatrix=None, basis=None, ecp=None,
                                         charge=0, frozen=None, transf=None, verbose=0, output=None,
                                         point_group_symmetry=False,
                                         point_group_symmetry_subgroup=None, soscf=False, df=False)
```

RHF calculations.

Parameters

- **geometery** (*Union[List, str, Geometry]*, default: *None*) – Molecular geometry.
- **zmatrix** (*str*, default: *None*) – Z matrix representation of molecular geometry (Used only if *geometry* is not specified).
- **basis** (*Any*, default: *None*) – Atomic basis set valid for *Mole* class.
- **ecp** (*Any*, default: *None*) – Effective core potentials.
- **charge** (*int*, default: 0) – Total charge.
- **frozen** (*Union[List[int], Callable[[RHF], int]]*, default: *None*) – Frozen orbital information.
- **transf** (*Transf*, default: *None*) – Orbital transformer.
- **verbose** (*int*, default: 0) – Control PySCF verbosity.
- **output** (*str*, default: *None*) – Specify log file name. If *None*, logs are printed to STDOUT.
- **point_group_symmetry** (*bool*, default: *False*) – Enable point group symmetry.
- **point_group_symmetry_subgroup** (*str*, default: *None*) – Use this point group symmetry instead of full point group symmetry.
- **soscf** (*bool*, default: *False*) – Use Second-Order SCF solver (Newton's method).

- **df** (`bool`, default: `False`) – Use density fitting.

`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** (default: `True`) – Reduce infinite point groups, e.g., Coov -> C2v.

Returns

`Tuple[str, List[str]]` – Point group symmetry, list of orbital irreps.

➊ 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]]]]], default: None) – Frozen core specified as either list or callable.
- **transf** (Union[Callable[[ndarray], ndarray], Transf], default: None) – Orbital transformation function.

Returns

ChemistryDriverPySCFMolecularRHF – PySCF driver.

```
property frozen: List[int] | List[List[int]]
```

Return the frozen orbital information.

```
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_ac0_correction(rdm)
```

Compute the AC0 correction to the energy from the provided density matrices.

Requires installation of `pyscf-ac0` extension, which is partially based on `GAMMCOR` and is distributed separately.

Parameters

- **rdm** (Tuple[ndarray[Any, dtype[float]], ndarray[Any, dtype[float]]]) – A tuple of one- and two-particle reduced density matrices as RDMs as (rdm1, rdm2).

Returns

The AC0 correction to the energy.

See also

For more information on AC0 correction, refer to: - DOI: 10.1021/acs.jctc.8b00213

```
get_cascl_1234pdms()
```

Calculate 1-, 2-, 3- and 4-PDM from a CASCI wavefunction.

Returns

- `Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1-, 2-, 3- and 4-PDMs.

```
get_cascl_12rdms()
```

Calculate 1-RDM and 2-RDM from a CASCI wavefunction.

Returns

- `Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1- and 2-RDMs.

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_double_factorized_system(*tol1*=-1, *tol2*=None, *method*=*DecompositionMethod.EIG*,
diagonalize_one_body=True, *diagonalize_one_body_offset*=True,
combine_one_body_terms=True)

Calculate double-factorized Hamiltonian operator, Fock space, and Hartree-Fock state.

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 which is to be double-factorized, and S is a one-body energy offset 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.

First level of factorization decomposes the electron repulsion integral (ERI) tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or pivoted, incomplete Cholesky decomposition (`method='cho'`). For details about Cholesky decomposition, refer to References [1] and [2]. The second factorization is diagonalization of the V_{pq}^t matrix for each t: $V_{pq}^t = \sum_u^{N_\lambda} U_{pu}^t \lambda_u^t U_{qu}^t$.

At the first factorization stage, truncation depends on the decomposition method. For ‘eig’, we discard eigenvalues, starting from the smallest, until the sum of those discarded exceeds `tol1`. For ‘cho’, the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. 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 decomposition of ERI matrix. If negative, no truncation is performed.
- **tol2** (`Optional[float]`, default: None) – Truncation threshold for second decomposition. If None, same as `tol1`.
- **method** (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`) – Decomposition method used for the first level of factorization. ‘eig’ for an eigenvalue decomposition, ‘cho’ for a pivoted, incomplete Cholesky decomposition.
- **diagonalize_one_body** (`bool`, default: `True`) – Whether to diagonalize the physical one-body integrals h_{pq} .
- **diagonalize_one_body_offset** (`bool`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq} .
- **combine_one_body_terms** (`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

`Tuple[DoubleFactorizedHamiltonian, FermionSpace, FermionState]` – Hamiltonian operator storing two-body integrals in double factorized form, Fock space, Fock state.

References

1. Henrik Koch, Alfredo Sánchez de Merás, and Thomas Bondo Pedersen. Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.*, 118(21): 9481-9484, 2003. URL: <https://doi.org/10.1063/1.1578621>, doi:10.1063/1.1578621
2. Evgeny Epifanovsky, Dmitry Zuev, Xintian Feng, Kirill Khistyayev, Yang Shao, and Anna I. Krylov. General implementation of the resolution-of-the-identity and Cholesky representations of electron repulsion integrals within coupled-cluster and equation-of-motion methods: Theory and benchmarks. *J. Chem. Phys.*, 139(13): 134105, 2013. URL: <https://doi.org/10.1063/1.4820484>, doi:10.1063/1.4820484

`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 single-electron excitation amplitudes.
- **t2** (`Optional[ndarray]`, default: None) – Guess two-electron 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 single-electron excitation amplitudes.
- **t2** (*Optional[ndarray]*, default: None) – Guess two-electron excitation amplitudes.
- **compact** (*bool*, default: False) – Enable compact form of excitation for the restricted wavefunctions.
- **antihermitian** (*bool*, default: True) – Returns anti-Hermitian operators $T - T^\dagger$ if specified.

Returns

FermionOperatorList – Excitation operator list.

get_lowdin_system(*method*='lowdin', *store_ao*=False)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

- **method** (*str*, default: "lowdin") – Method passed to PySCF's `orth.orth_ao()`.
- **store_ao** (*bool*, default: False) – If True, the returned Hamiltonian operator is of type *PySCFChemistryRestrictedIntegralOperator* or *PySCFChemistryUnrestrictedIntegralOperator*, which stores the underlying atomic orbitals internally.

Returns

Tuple[Union[ChemistryRestrictedIntegralOperator, ChemistryUnrestrictedIntegralOperator, PySCFChemistryRestrictedIntegralOperator, PySCFChemistryUnrestrictedIntegralOperator], FermionSpace, Union[RestrictedOneBodyRDM, UnrestrictedOneBodyRDM]] – Fermion Hamiltonian, Fock space, Fock state.

 **See also**

[orth.orth_ao documentation](#).

get_mulliken_pop()

Interface for Mulliken population analysis of PySCF.

Returns

Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, 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, ..., pdm4).

Returns

The NEVPT2 correction to the energy.

`get_orbital_coefficients()`

Returns orbital coefficients.

Returns

`ndarray` – Orbital coefficients.

`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_co, bound= generic, covariant=True)]]` – Two-body reduced density matrix.

Note

This object will be replaced with an `RestrictedTwoBodyRDM` class to avoid returning a raw 4D tensor.

`get_subsystem_driver(frozen=None, transf=None)`

Generate a driver object wrapping the current active space Hamiltonian.

Parameters

- `frozen` (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`) – Frozen orbital information (applies to the subsystem driver).
- `transf` (`Union[Callable[[array], array], Transf]`, default: `None`) – Orbital transformer (applies to the subsystem driver).

Returns

`BasePySCFDriver` – PySCF driver object wrapping the current active space Hamiltonian.

`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.`

`make_actives_contiguous()`

Reorder orbitals so that active orbitals form a contiguous block.

Reorders orbital coefficients, orbital energies, occupations and the list of frozen (active) orbitals. Implemented for RHF, ROHF and UHF.

`property mf_energy: float`

Return the total mean-field energy.

Returns

Total mean-field energy.

`property mf_type: str`

Return the mean-field type as a string, with options including "RHF", "ROHF", and "UHF".

Returns

Mean-field type name.

`property n_electron: int`

Return the total number of electrons in the active space.

Returns

Total number of electrons.

`property n_orb: int`

Return the number of spatial orbitals.

Returns

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 mcsfc.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.

set_checkfile(chkfile, init_guess=True)
    Set checkpoint file name.

    The PySCF calculation results will be saved to the checkpoint file.

    Parameters
        • chkfile (str) – name of checkpoint file.
        • init_guess (bool, default: True) – If True and the checkpoint file exists, the initial guess will be read from the checkpoint file.

set_diis_space_dimension(diis_space_dimension=8)
    Set number of DIIS vectors.

    Parameters
        diis_space_dimension (int, default: 8) – dimension of the DIIS space.

set_init_orbitals(init_orbs)
    Sets the initial guess orbitals for the SCF.

    Parameters
        init_orbs (ndarray[TypeVar(_ScalarType_co, bound=generic, covariant=True)]) – Initial orbital coefficients.

    Return type
        None
```

`set_level_shift (level_shift_value=0)`

Set value of the artificial shift applied to virtual orbitals during SCF.

Note

Level shifting in PySCF changes the HF energy, even though the last SCF iteration is performed without it.

Parameters

`level_shift_value (float, default: 0)` – value of the level shift parameter.

`set_max_scf_cycles (max_cycles=50)`

Set maximum number of SCF cycles.

Parameters

`max_cycles (int, default: 50)` – maximum number of SCF cycles.

```
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,
                                                   solvent_epsilon=COSMO_SOLVENT['water'],
                                                   verbose=0, output=None,
                                                   point_group_symmetry=False,
                                                   point_group_symmetry_subgroup=None, soscf=False,
                                                   df=False)
```

PySCF driver for molecular RHF, with Quantum Mechanics - Molecular Mechanics (QMMM) and COnductor-like Screening MOdel (COSMO) calculations.

Parameters

- `geometry (Union[List, str], default: None)` – Molecular geometry.
- `zmatrix (str, default: None)` – Z matrix representation of molecular geometry (Used only if `geometry` is not specified).
- `basis (Any, default: None)` – Atomic basis set valid for `Mole` class.
- `ecp (Any, default: None)` – Effective core potentials.
- `charge (int, default: 0)` – Total charge.
- `frozen (Any, default: None)` – Frozen orbital information.
- `transf (Transf, default: None)` – Orbital transformer.
- `do_qmmm (bool, default: None)` – If True, do QMMM embedding. MM geometry and charges are required.
- `mm_charges (Any, default: None)` – Point charge values for MM region.
- `mm_geometry (Any, default: None)` – Geometry of MM region.
- `do_mm_coulomb (bool, default: None)` – If True, calculate MM-MM Coulomb interaction energy. Needs to be added as a constant to the final energy: $\langle H \rangle + \text{self.e_mm_coulomb}$
- `do_cosmo (bool, default: None)` – If True, adds implicit water (COSMO) to the mean-field calculation. `get_system_legacy()` returns $H + v_{\text{solvent}}$ as fermion operator. The resulting final energy needs to be corrected as $\langle H + v_{\text{solvent}} \rangle + \text{self.cosmo_correction}$.

Where `self.cosmo_correction = E_cosmo - Tr[v_solvent * rdm1]`. If `do_cosmo=True`, `self.cosmo_correction` is defined after `self._run_hf()`.

- **`solvent_epsilon`** (`float`, default: `COSMO_SOLVENT["water"]`) – Dielectric constant of the solvent in the COSMO model.
- **`verbose`** (`int`, default: 0) – Control SCF verbosity.
- **`output`** (`str`, default: `None`) – Specify log file name. If `None`, logs are printed to `STDOUT`.
- **`point_group_symmetry`** (`bool`, default: `False`) – Enable point group symmetry.
- **`point_group_symmetry_subgroup`** (`str`, default: `None`) – Use this point group symmetry instead of full point group symmetry.
- **`soscf`** (`bool`, default: `False`) – Use Second-Order SCF solver (Newton's method).
- **`df`** (`bool`, default: `False`) – Use density fitting.

`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` (default: `True`) – Reduce infinite point groups, e.g., C₀₀ -> C_{2v}.

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]]]]]`, default: `None`) – Frozen core specified as either list or callable.
- `transf` (`Union[Callable[[ndarray], ndarray], Transf]`, default: `None`) – Orbital transformation function.

Returns

`ChemistryDriverPySCFMolecularRHFQMMMCOSMO` – PySCF driver.

`property frozen: List[int] | List[List[int]]`

Return the frozen orbital information.

`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_ac0_correction(rdms)`

Compute the AC0 correction to the energy from the provided density matrices.

Requires installation of `pyscf-ac0` extension, which is partially based on `GAMMCOR` and is distributed separately.

Parameters

`rdms` (`Tuple[ndarray[Any, dtype[float]], ndarray[Any, dtype[float]]]`) – A tuple of one- and two-particle reduced density matrices as RDMs as (rdm1, rdm2).

Returns

The AC0 correction to the energy.

 **See also**

For more information on AC0 correction, refer to: - DOI: 10.1021/acs.jctc.8b00213

get_casci_1234pdms()

Calculate 1-, 2-, 3- and 4-PDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1-, 2-, 3- and 4-PDMs.

get_casci_12rdms()

Calculate 1-RDM and 2-RDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1- and 2-RDMs.

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_double_factorized_system(tol1=-1, tol2=None, method=DecompositionMethod.EIG, diagonalize_one_body=True, diagonalize_one_body_offset=True, combine_one_body_terms=True)

Calculate double-factorized Hamiltonian operator, Fock space, and Hartree-Fock state.

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 which is to be double-factorized, and S is a one-body energy offset 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.

First level of factorization decomposes the electron repulsion integral (ERI) tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or pivoted, incomplete Cholesky decomposition (`method='cho'`). For details about Cholesky decomposition, refer to References [1] and [2]. 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 the first factorization stage, truncation depends on the decomposition method. For ‘eig’, we discard eigenvalues, starting from the smallest, until the sum of those discarded exceeds `tol1`. For ‘cho’, the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. 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 decomposition of ERI matrix. If negative, no truncation is performed.
- `tol2` (`Optional[float]`, default: None) – Truncation threshold for second decomposition. If None, same as `tol1`.
- `method` (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`) – Decomposition method used for the first level of factorization. ‘eig’ for an eigenvalue decomposition, ‘cho’ for a pivoted, incomplete Cholesky decomposition.
- `diagonalize_one_body` (`bool`, default: True) – Whether to diagonalize the physical one-body integrals h_{pq} .
- `diagonalize_one_body_offset` (`bool`, default: True) – Whether to diagonalize the one-body offset integrals s_{pq} .
- `combine_one_body_terms` (`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

`Tuple[DoubleFactorizedHamiltonian, FermionSpace, FermionState]` – Hamiltonian operator storing two-body integrals in double factorized form, Fock space, Fock state.

References

1. Henrik Koch, Alfredo Sánchez de Merás, and Thomas Bondo Pedersen. Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.*, 118(21): 9481-9484, 2003. URL: <https://doi.org/10.1063/1.1578621>, doi:10.1063/1.1578621
2. Evgeny Epifanovsky, Dmitry Zuev, Xintian Feng, Kirill Khistyayev, Yang Shao, and Anna I. Krylov. General implementation of the resolution-of-the-identity and Cholesky representations of electron repulsion integrals within coupled-cluster and equation-of-motion methods: Theory and benchmarks. *J. Chem. Phys.*, 139(13): 134105, 2013. URL: <https://doi.org/10.1063/1.4820484>, doi:10.1063/1.4820484

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 single-electron excitation amplitudes.
- **t2** (*Optional*[ndarray], default: None) – Guess two-electron 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 single-electron excitation amplitudes.
- **t2** (*Optional*[ndarray], default: None) – Guess two-electron excitation amplitudes.
- **compact** (*bool*, default: False) – Enable compact form of excitation for the restricted wavefunctions.
- **antihermitian** (*bool*, default: True) – Returns anti-Hermitian operators $T - T^\dagger$ if specified.

Returns

FermionOperatorList – Excitation operator list.

get_lowdin_system (*method*='lowdin', *store_ao*=False)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

- **method** (*str*, default: "lowdin") – Method passed to PySCF's `orth.orth_ao()`.

- **store_ao** (`bool`, default: `False`) – If True, the returned Hamiltonian operator is of type `PySCFChemistryRestrictedIntegralOperator` or `PySCFChemistryUnrestrictedIntegralOperator`, which stores the underlying atomic orbitals internally.

Returns

`Tuple[Union[ChemistryRestrictedIntegralOperator, ChemistryUnrestrictedIntegralOperator, PySCFChemistryRestrictedIntegralOperator, PySCFChemistryUnrestrictedIntegralOperator], FermionSpace, Union[RestrictedOneBodyRDM, UnrestrictedOneBodyRDM]]` – Fermion Hamiltonian, Fock space, Fock state.

↳ **See also**

[orth.orth_ao](#) documentation.

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_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction(rdm)

Compute the Strongly contracted NEVPT2 correction to the energy from the provided density matrices.

Parameters

rdm (`Tuple`) – A tuple of reduced density matrices as PDMs in PySCF's un-reordered format as `(pdm1, ..., pdm4)`.

Returns

The NEVPT2 correction to the energy.

get_orbital_coefficients()

Returns orbital coefficients.

Returns

`ndarray` – Orbital coefficients.

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_co, bound= generic, covariant=True)]]` – Two-body reduced density matrix.

Note

This object will be replaced with an `RestrictedTwoBodyRDM` class to avoid returning a raw 4D tensor.

get_subsystem_driver(frozen=None, transf=None)

Generate a driver object wrapping the current active space Hamiltonian.

Parameters

- **frozen** (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`) – Frozen orbital information (applies to the subsystem driver).
- **transf** (`Union[Callable[[array], array], Transf]`, default: `None`) – Orbital transformer (applies to the subsystem driver).

Returns

`BasePySCFDriver` – PySCF driver object wrapping the current active space Hamiltonian.

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.

`make_actives_contiguous()`

Reorder orbitals so that active orbitals form a contiguous block.

Reorders orbital coefficients, orbital energies, occupations and the list of frozen (active) orbitals. Implemented for RHF, ROHF and UHF.

`property mf_energy: float`

Return the total mean-field energy.

Returns

Total mean-field energy.

`property mf_type: str`

Return the mean-field type as a string, with options including "RHF", "ROHF", and "UHF".

Returns

Mean-field type name.

`property n_electron: int`

Return the total number of electrons in the active space.

Returns

Total number of electrons.

`property n_orb: int`

Return the number of spatial orbitals.

Returns

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.

set_checkfile(chkfile, init_guess=True)

Set checkpoint file name.

The PySCF calculation results will be saved to the checkpoint file.

Parameters

- `chkfile` (`str`) – name of checkpoint file.
- `init_guess` (`bool`, default: `True`) – If `True` and the checkpoint file exists, the initial guess will be read from the checkpoint file.

set_diis_space_dimension(diis_space_dimension=8)

Set number of DIIS vectors.

Parameters

`diis_space_dimension` (`int`, default: 8) – dimension of the DIIS space.

set_init_orbitals(init_orbs)

Sets the initial guess orbitals for the SCF.

Parameters

`init_orbs` (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]`) – Initial orbital coefficients.

Return type

`None`

set_level_shift(level_shift_value=0)

Set value of the artificial shift applied to virtual orbitals during SCF.

Note

Level shifting in PySCF changes the HF energy, even though the last SCF iteration is performed without it.

Parameters

`level_shift_value` (`float`, default: 0) – value of the level shift parameter.

set_max_scf_cycles(max_cycles=50)

Set maximum number of SCF cycles.

Parameters

`max_cycles` (`int`, default: 50) – maximum number of SCF cycles.

```
class ChemistryDriverPySCFMolecularROHF(geometery=None, zmatrix=None, basis=None, ecp=None,
                                         charge=0, multiplicity=1, frozen=None, transf=None,
                                         verbose=0, output=None, point_group_symmetry=False,
                                         point_group_symmetry_subgroup=None, soscf=False, df=False)
```

PySCF driver for molecular ROHF calculations.

Parameters

- **geometry** (`Union[List, str, Geometry]`, default: `None`) – Molecular geometry.
- **zmatrix** (`str`, default: `None`) – Z matrix representation of molecular geometry (Used only if `geometry` is not specified).
- **basis** (`Any`, default: `None`) – Atomic basis set valid for `Mole` class.
- **ecp** (`Any`, default: `None`) – Effective core potentials.
- **charge** (`int`, default: `0`) – Total charge.
- **multiplicity** (`int`, default: `1`) – Spin multiplicity, $2S+1$.
- **frozen** (`Any`, default: `None`) – Frozen orbital information.
- **transf** (`Transf`, default: `None`) – Orbital transformer.
- **verbose** (`int`, default: `0`) – Control PySCF verbosity.
- **output** (`str`, default: `None`) – Specify log file name. If `None`, logs are printed to `STDOUT`.
- **point_group_symmetry** (`bool`, default: `False`) – Enable point group symmetry.
- **point_group_symmetry_subgroup** (`str`, default: `None`) – Use this point group symmetry instead of full point group symmetry.
- **soscf** (`bool`, default: `False`) – Use Second-Order SCF solver (Newton's method).
- **df** (`bool`, default: `False`) – Use density fitting.

`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** (default: `True`) – Reduce infinite point groups, e.g., C₀₀ -> C_{2v}.

Returns

`Tuple[str, List[str]]` – Point group symmetry, list of orbital irreps.

❶ 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]]]]]`, default: `None`) – Frozen core specified as either list or callable.
- **transf** (`Union[Callable[[ndarray], ndarray], Transf]`, default: `None`) – Orbital transformation function.

Returns

`ChemistryDriverPySCFMolecularROHF` – PySCF driver.

property frozen: List[int] | List[List[int]]

Return the frozen orbital information.

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_ac0_correction (`rdms`)

Compute the AC0 correction to the energy from the provided density matrices.

Requires installation of pyscf-ac0 extension, which is partially based on GAMMCOR and is distributed separately.

Parameters

`rdms` (`Tuple[ndarray[Any, dtype[float]], ndarray[Any, dtype[float]]]`) – A tuple of one- and two-particle reduced density matrices as RDMs as (rdm1, rdm2).

Returns

The AC0 correction to the energy.

See also

For more information on AC0 correction, refer to: - DOI: 10.1021/acs.jctc.8b00213

`get_casci_1234pdms()`

Calculate 1-, 2-, 3- and 4-PDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1-, 2-, 3- and 4-PDMs.

`get_casci_12rdms()`

Calculate 1-RDM and 2-RDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1- and 2-RDMs.

`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_double_factorized_system(tol1=-1, tol2=None, method=DecompositionMethod.EIG,
                             diagonalize_one_body=True, diagonalize_one_body_offset=True,
                             combine_one_body_terms=True)
```

Calculate double-factorized Hamiltonian operator, Fock space, and Hartree-Fock state.

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 which is to be double-factorized, and S is a one-body energy offset 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.

First level of factorization decomposes the electron repulsion integral (ERI) tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or pivoted, incomplete Cholesky decomposition (`method='cho'`). For details about Cholesky decomposition, refer to References [1] and [2]. 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 the first factorization stage, truncation depends on the decomposition method. For ‘eig’, we discard eigenvalues, starting from the smallest, until the sum of those discarded exceeds `tol1`. For ‘cho’, the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. 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 decomposition of ERI matrix. If negative, no truncation is performed.
- `tol2` (`Optional[float]`, default: None) – Truncation threshold for second decomposition. If None, same as `tol1`.
- `method` (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`) – Decomposition method used for the first level of factorization. ‘eig’ for an eigenvalue decomposition, ‘cho’ for a pivoted, incomplete Cholesky decomposition.
- `diagonalize_one_body` (`bool`, default: True) – Whether to diagonalize the physical one-body integrals h_{pq} .
- `diagonalize_one_body_offset` (`bool`, default: True) – Whether to diagonalize the one-body offset integrals s_{pq} .
- `combine_one_body_terms` (`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

`Tuple[DoubleFactorizedHamiltonian, FermionSpace, FermionState]` – Hamiltonian operator storing two-body integrals in double factorized form, Fock space, Fock state.

References

- Henrik Koch, Alfredo Sánchez de Merás, and Thomas Bondo Pedersen. Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.*, 118(21): 9481-9484, 2003. URL: <https://doi.org/10.1063/1.1578621>, doi:10.1063/1.1578621
- Evgeny Epifanovsky, Dmitry Zuev, Xintian Feng, Kirill Khistyayev, Yang Shao, and Anna I. Krylov. General implementation of the resolution-of-the-identity and Cholesky representations of electron repulsion integrals within coupled-cluster and equation-of-motion methods: Theory and benchmarks. *J. Chem. Phys.*, 139(13): 134105, 2013. URL: <https://doi.org/10.1063/1.4820484>, doi:10.1063/1.4820484

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 single-electron excitation amplitudes.
- **t2** (*Optional*[ndarray], default: None) – Guess two-electron 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 single-electron excitation amplitudes.
- **t2** (*Optional*[ndarray], default: None) – Guess two-electron excitation amplitudes.
- **compact** (*bool*, default: False) – Enable compact form of excitation for the restricted wavefunctions.
- **antihermitian** (*bool*, default: True) – Returns anti-Hermitian operators $T - T^\dagger$ if specified.

Returns

FermionOperatorList – Excitation operator list.

get_lowdin_system (*method*='lowdin', *store_ao*=False)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

- **method** (*str*, default: "lowdin") – Method passed to PySCF's `orth.orth_ao()`.

- **store_ao** (`bool`, default: `False`) – If True, the returned Hamiltonian operator is of type `PySCFChemistryRestrictedIntegralOperator` or `PySCFChemistryUnrestrictedIntegralOperator`, which stores the underlying atomic orbitals internally.

Returns

`Tuple[Union[ChemistryRestrictedIntegralOperator, ChemistryUnrestrictedIntegralOperator, PySCFChemistryRestrictedIntegralOperator, PySCFChemistryUnrestrictedIntegralOperator], FermionSpace, Union[RestrictedOneBodyRDM, UnrestrictedOneBodyRDM]]` – Fermion Hamiltonian, Fock space, Fock state.

➡ See also

[orth.orth_ao](#) documentation.

get_mulliken_pop()

Interface for Mulliken population analysis of PySCF.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction(rdm)

Compute the Strongly contracted NEVPT2 correction to the energy from the provided density matrices.

Parameters

`rdm` (`Tuple`) – A tuple of reduced density matrices as PDMs in PySCF's un-reordered format as (pdm1, ..., pdm4).

Returns

The NEVPT2 correction to the energy.

get_orbital_coefficients()

Returns orbital coefficients.

Returns

`ndarray` – Orbital coefficients.

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_co, bound= generic, covariant=True)]]` – Two-body reduced density matrix.

ⓘ Note

This object will be replaced with an `RestrictedTwoBodyRDM` class to avoid returning a raw 4D tensor.

get_subsystem_driver(*frozen=None*, *transf=None*)

Generate a driver object wrapping the current active space Hamiltonian.

Parameters

- **frozen** (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`) – Frozen orbital information (applies to the subsystem driver).
- **transf** (`Union[Callable[[array], array], Transf]`, default: `None`) – Orbital transformer (applies to the subsystem driver).

Returns

`BasePySCFDriver` – PySCF driver object wrapping the current active space Hamiltonian.

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.

make_actives_contiguous()

Reorder orbitals so that active orbitals form a contiguous block.

Reorders orbital coefficients, orbital energies, occupations and the list of frozen (active) orbitals. Implemented for RHF, ROHF and UHF.

property mf_energy: float

Return the total mean-field energy.

Returns

Total mean-field energy.

property mf_type: str

Return the mean-field type as a string, with options including "RHF", "ROHF", and "UHF".

Returns

Mean-field type name.

property n_electron: int

Return the total number of electrons in the active space.

Returns

Total number of electrons.

property n_orb: int

Return the number of spatial orbitals.

Returns

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 `mcsf.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.

`set_checkfile(chkfile, init_guess=True)`

Set checkpoint file name.

The PySCF calculation results will be saved to the checkpoint file.

Parameters

- `chkfile` (`str`) – name of checkpoint file.
- `init_guess` (`bool`, default: `True`) – If `True` and the checkpoint file exists, the initial guess will be read from the checkpoint file.

`set_diis_space_dimension(diis_space_dimension=8)`

Set number of DIIS vectors.

Parameters

`diis_space_dimension` (`int`, default: 8) – dimension of the DIIS space.

`set_init_orbitals(init_orbs)`

Sets the initial guess orbitals for the SCF.

Parameters

`init_orbs` (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]`) – Initial orbital coefficients.

Return type

`None`

`set_level_shift(level_shift_value=0)`

Set value of the artificial shift applied to virtual orbitals during SCF.

Note

Level shifting in PySCF changes the HF energy, even though the last SCF iteration is performed without it.

Parameters

`level_shift_value` (`float`, default: 0) – value of the level shift parameter.

`set_max_scf_cycles(max_cycles=50)`

Set maximum number of SCF cycles.

Parameters

`max_cycles` (`int`, default: 50) – maximum number of SCF cycles.

```
class ChemistryDriverPySCFMolecularROHFQMMMCOSMO(geometery=None, zmatrix=None, basis=None,
                                                 ecp=None, charge=0, multiplicity=1, frozen=None,
                                                 transf=None, do_qmmm=None,
                                                 mm_charges=None, mm_geometry=None,
                                                 do_mm_coulomb=None, do_cosmo=None,
                                                 solvent_epsilon=COSMO_SOLVENT['water'],
                                                 verbose=0, output=None,
                                                 point_group_symmetry=False,
                                                 point_group_symmetry_subgroup=None,
                                                 soscf=False, df=False)
```

PySCF driver for molecular ROHF, with Quantum Mechanics - Molecular Mechanics (QMMM) and COnductor-like Screening MOdel (COSMO) calculations.

Parameters

- **geometry** (`Union[List, str]`, default: `None`) – Molecular geometry.
- **zmatrix** (`str`, default: `None`) – Z matrix representation of molecular geometry (Used only if `geometry` is not specified).
- **basis** (`Any`, default: `None`) – Atomic basis set valid for `Mole` class.
- **ecp** (`Any`, default: `None`) – Effective core potentials.
- **charge** (`int`, default: `0`) – Total charge.
- **multiplicity** (`int`, default: `1`) – Spin multiplicity, $2S+1$.
- **frozen** (`Any`, default: `None`) – Frozen orbital information.
- **transf** (`Transf`, default: `None`) – Orbital transformer.
- **do_qmmm** (`bool`, default: `None`) – If True, do QMMM embedding. MM geometry and charges are required.
- **mm_charges** (`Any`, default: `None`) – Point charge values for MM region.
- **mm_geometry** (`Any`, default: `None`) – Geometry of MM region.
- **do_mm_coulomb** (`bool`, default: `None`) – If True, calculate MM-MM Coulomb interaction energy. Needs to be added as a constant to the final energy: $\|H\| + \text{self.e_mm_coulomb}$.
- **do_cosmo** (`bool`, default: `None`) – If True, adds implicit water (COSMO) to the mean-field calculation. `get_system_legacy()` returns $H + v_{\text{solvent}}$ as fermion operator. The resulting final energy needs to be corrected as $\|H + v_{\text{solvent}}\| + \text{self.cosmo_correction}$. Where `self.cosmo_correction = E_{\text{cosmo}} - \text{Tr}[v_{\text{solvent}} \cdot \text{rdm1}]`. If `do_cosmo=True`, `self.cosmo_correction` is defined after `self._run_hf()`.
- **solvent_epsilon** (`float`, default: `COSMO_SOLVENT["water"]`) – Dielectric constant of the solvent in the COSMO model.
- **verbose** (`int`, default: `0`) – Control SCF verbosity.
- **output** (`str`, default: `None`) – Specify log file name. If `None`, logs are printed to `STDOUT`.
- **point_group_symmetry** (`bool`, default: `False`) – Enable point group symmetry.
- **point_group_symmetry_subgroup** (`str`, default: `None`) – Use this point group symmetry instead of full point group symmetry.
- **soscf** (`bool`, default: `False`) – Use Second-Order SCF solver (Newton's method).
- **df** (`bool`, default: `False`) – Use density fitting.

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 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` (default: `True`) – 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]]]]]`, default: `None`) – Frozen core specified as either list or callable.
- **transf** (`Union[Callable[[ndarray], ndarray], Transf]`, default: `None`) – Orbital transformation function.

Returns

`ChemistryDriverPySCFMolecularROHFQMMMCOSMO` – PySCF driver.

property frozen: List[int] | List[List[int]]

Return the frozen orbital information.

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_ac0_correction (rdms)

Compute the AC0 correction to the energy from the provided density matrices.

Requires installation of `pyscf-ac0` extension, which is partially based on `GAMMCOR` and is distributed separately.

Parameters

`rdms (Tuple[ndarray[Any, dtype[float]], ndarray[Any, dtype[float]]])` – A tuple of one- and two-particle reduced density matrices as RDMs as `(rdm1, rdm2)`.

Returns

The AC0 correction to the energy.

➡ See also

For more information on AC0 correction, refer to: - DOI: 10.1021/acs.jctc.8b00213

get_cascl_1234pdms ()

Calculate 1-, 2-, 3- and 4-PDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1-, 2-, 3- and 4-PDMs.

get_cascl_12rdms ()

Calculate 1-RDM and 2-RDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1- and 2-RDMs.

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_double_factorized_system(`tol1=-1, tol2=None, method=DecompositionMethod.EIG, diagonalize_one_body=True, diagonalize_one_body_offset=True, combine_one_body_terms=True`)

Calculate double-factorized Hamiltonian operator, Fock space, and Hartree-Fock state.

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 which is to be double-factorized, and S is a one-body energy offset 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.

First level of factorization decomposes the electron repulsion integral (ERI) tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or pivoted, incomplete Cholesky decomposition (`method='cho'`). For details about Cholesky decomposition, refer to References [1] and [2]. 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 the first factorization stage, truncation depends on the decomposition method. For ‘eig’, we discard eigenvalues, starting from the smallest, until the sum of those discarded exceeds `tol1`. For ‘cho’, the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. 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 decomposition of ERI matrix. If negative, no truncation is performed.
- **tol2** (`Optional[float]`, default: `None`) – Truncation threshold for second decomposition. If `None`, same as `tol1`.
- **method** (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`) – Decomposition method used for the first level of factorization. ‘eig’ for an eigenvalue decomposition, ‘cho’ for a pivoted, incomplete Cholesky decomposition.
- **diagonalize_one_body** (`bool`, default: `True`) – Whether to diagonalize the physical one-body integrals h_{pq} .
- **diagonalize_one_body_offset** (`bool`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq} .
- **combine_one_body_terms** (`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

`Tuple[DoubleFactorizedHamiltonian, FermionSpace, FermionState]` – Hamiltonian operator storing two-body integrals in double factorized form, Fock space, Fock state.

References

1. Henrik Koch, Alfredo Sánchez de Merás, and Thomas Bondo Pedersen. Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.*, 118(21): 9481-9484, 2003. URL: <https://doi.org/10.1063/1.1578621>, doi:10.1063/1.1578621
2. Evgeny Epifanovsky, Dmitry Zuev, Xintian Feng, Kirill Khistyayev, Yang Shao, and Anna I. Krylov. General implementation of the resolution-of-the-identity and Cholesky representations of electron repulsion integrals within coupled-cluster and equation-of-motion methods: Theory and benchmarks. *J. Chem. Phys.*, 139(13): 134105, 2013. URL: <https://doi.org/10.1063/1.4820484>, doi:10.1063/1.4820484

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 single-electron excitation amplitudes.
- **t2** (`Optional[ndarray]`, default: `None`) – Guess two-electron 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 single-electron excitation amplitudes.
- **t2** (*Optional[ndarray]*, default: None) – Guess two-electron excitation amplitudes.
- **compact** (*bool*, default: False) – Enable compact form of excitation for the restricted wavefunctions.
- **antihermitian** (*bool*, default: True) – Returns anti-Hermitian operators $T - T^\dagger$ if specified.

Returns

FermionOperatorList – Excitation operator list.

get_lowdin_system(*method='lowdin'*, *store_ao=False*)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

- **method** (*str*, default: "lowdin") – Method passed to PySCF's `orth.orth_ao()`.
- **store_ao** (*bool*, default: False) – If True, the returned Hamiltonian operator is of type *PySCFChemistryRestrictedIntegralOperator* or *PySCFChemistryUnrestrictedIntegralOperator*, which stores the underlying atomic orbitals internally.

Returns

Tuple[Union[ChemistryRestrictedIntegralOperator, ChemistryUnrestrictedIntegralOperator, PySCFChemistryRestrictedIntegralOperator, PySCFChemistryUnrestrictedIntegralOperator], FermionSpace, Union[RestrictedOneBodyRDM, UnrestrictedOneBodyRDM]] – Fermion Hamiltonian, Fock space, Fock state.

See also

[orth.orth_ao documentation](#).

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_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, 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, ..., pdm4).

Returns

The NEVPT2 correction to the energy.

get_orbital_coefficients ()

Returns orbital coefficients.

Returns

`ndarray` – Orbital coefficients.

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_co, bound= generic, covariant=True)]]` – Two-body reduced density matrix.

Note

This object will be replaced with an `RestrictedTwoBodyRDM` class to avoid returning a raw 4D tensor.

get_subsystem_driver (frozen=None, transf=None)

Generate a driver object wrapping the current active space Hamiltonian.

Parameters

- `frozen` (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`) – Frozen orbital information (applies to the subsystem driver).
- `transf` (`Union[Callable[[array], array], Transf]`, default: `None`) – Orbital transformer (applies to the subsystem driver).

Returns

`BasePySCFDriver` – PySCF driver object wrapping the current active space Hamiltonian.

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.`

`make_actives_contiguous()`

Reorder orbitals so that active orbitals form a contiguous block.

Reorders orbital coefficients, orbital energies, occupations and the list of frozen (active) orbitals. Implemented for RHF, ROHF and UHF.

`property mf_energy: float`

Return the total mean-field energy.

Returns

Total mean-field energy.

`property mf_type: str`

Return the mean-field type as a string, with options including "RHF", "ROHF", and "UHF".

Returns

Mean-field type name.

`property n_electron: int`

Return the total number of electrons in the active space.

Returns

Total number of electrons.

```
property n_orb: int
```

Return the number of spatial orbitals.

Returns

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 `mcsf.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.

```
set_checkfile (chkfile, init_guess=True)
```

Set checkpoint file name.

The PySCF calculation results will be saved to the checkpoint file.

Parameters

- `chkfile` (`str`) – name of checkpoint file.
- `init_guess` (`bool`, default: `True`) – If `True` and the checkpoint file exists, the initial guess will be read from the checkpoint file.

```
set_diis_space_dimension (diis_space_dimension=8)
```

Set number of DIIS vectors.

Parameters

`diis_space_dimension` (`int`, default: 8) – dimension of the DIIS space.

set_init_orbitals (*init_orbs*)

Sets the initial guess orbitals for the SCF.

Parameters

init_orbs (ndarray[*Any*, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]) – Initial orbital coefficients.

Return type

None

set_level_shift (*level_shift_value*=0)

Set value of the artificial shift applied to virtual orbitals during SCF.

Note

Level shifting in PySCF changes the HF energy, even though the last SCF iteration is performed without it.

Parameters

level_shift_value (*float*, default: 0) – value of the level shift parameter.

set_max_scf_cycles (*max_cycles*=50)

Set maximum number of SCF cycles.

Parameters

max_cycles (*int*, default: 50) – maximum number of SCF cycles.

```
class ChemistryDriverPySCFMolecularUHF(geometery=None, zmatrix=None, basis=None, ecp=None,
                                         charge=0, multiplicity=1, frozen=None, verbose=0, output=None,
                                         point_group_symmetry=False,
                                         point_group_symmetry_subgroup=None, soscf=False, df=False)
```

PySCF driver for molecular UHF calculations.

Parameters

- **geometery** (*Union[List, str, Geometry]*, default: *None*) – Molecular geometry.
- **zmatrix** (*str*, default: *None*) – Z matrix representation of molecular geometry (Used only if *geometry* is not specified).
- **basis** (*Any*, default: *None*) – Atomic basis set valid for *Mole* class.
- **ecp** (*Any*, default: *None*) – Effective core potentials.
- **charge** (*int*, default: 0) – Total charge.
- **multiplicity** (*int*, default: 1) – Spin multiplicity, 2S+1.
- **frozen** (*Any*, default: *None*) – Frozen orbital information.
- **verbose** (*int*, default: 0) – Control PySCF verbosity.
- **output** (*str*, default: *None*) – Specify log file name. If *None*, logs are printed to STDOUT.
- **point_group_symmetry** (*bool*, default: *False*) – Enable point group symmetry.
- **point_group_symmetry_subgroup** (*str*, default: *None*) – Use this point group symmetry instead of full point group symmetry.
- **soscf** (*bool*, default: *False*) – Use Second-Order SCF solver (Newton's method).

- `df` (`bool`, default: `False`) – Use density fitting.

`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` (default: `True`) – Reduce infinite point groups, e.g., Coov -> C2v.

Returns

`Tuple[str, List[str]]` – Point group symmetry, list of orbital irreps.

❶ 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]]]]]`, 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.

`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_ac0_correction(rdm)`

Compute the AC0 correction to the energy from the provided density matrices.

Requires installation of `pyscf-ac0` extension, which is partially based on `GAMMCOR` and is distributed separately.

Parameters

- `rdm` (`Tuple[ndarray[Any, dtype[float]], ndarray[Any, dtype[float]]]`) – A tuple of one- and two-particle reduced density matrices as RDMs as (rdm1, rdm2).

Returns

The AC0 correction to the energy.

See also

For more information on AC0 correction, refer to: - DOI: 10.1021/acs.jctc.8b00213

`get_cascl_1234pdms()`

Calculate 1-, 2-, 3- and 4-PDM from a CASCI wavefunction.

Returns

- `Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1-, 2-, 3- and 4-PDMs.

`get_cascl_12rdms()`

Calculate 1-RDM and 2-RDM from a CASCI wavefunction.

Returns

- `Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1- and 2-RDMs.

`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_double_factorized_system(`tol1=-1, tol2=None, method=DecompositionMethod.EIG,`
`diagonalize_one_body=True, diagonalize_one_body_offset=True,`
`combine_one_body_terms=True`)

Calculate double-factorized Hamiltonian operator, Fock space, and Hartree-Fock state.

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 which is to be double-factorized, and S is a one-body energy offset 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.

First level of factorization decomposes the electron repulsion integral (ERI) tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or pivoted, incomplete Cholesky decomposition (`method='cho'`). For details about Cholesky decomposition, refer to References [1] and [2]. 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 the first factorization stage, truncation depends on the decomposition method. For ‘eig’, we discard eigenvalues, starting from the smallest, until the sum of those discarded exceeds `tol1`. For ‘cho’, the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. 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 decomposition of ERI matrix. If negative, no truncation is performed.
- **tol2** (`Optional[float]`, default: `None`) – Truncation threshold for second decomposition. If `None`, same as `tol1`.
- **method** (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`) – Decomposition method used for the first level of factorization. ‘eig’ for an eigenvalue decomposition, ‘cho’ for a pivoted, incomplete Cholesky decomposition.
- **diagonalize_one_body** (`bool`, default: `True`) – Whether to diagonalize the physical one-body integrals h_{pq} .
- **diagonalize_one_body_offset** (`bool`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq} .
- **combine_one_body_terms** (`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

`Tuple[DoubleFactorizedHamiltonian, FermionSpace, FermionState]` – Hamiltonian operator storing two-body integrals in double factorized form, Fock space, Fock state.

References

1. Henrik Koch, Alfredo Sánchez de Merás, and Thomas Bondo Pedersen. Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.*, 118(21): 9481-9484, 2003. URL: <https://doi.org/10.1063/1.1578621>, doi:10.1063/1.1578621
2. Evgeny Epifanovsky, Dmitry Zuev, Xintian Feng, Kirill Khistyayev, Yang Shao, and Anna I. Krylov. General implementation of the resolution-of-the-identity and Cholesky representations of electron repulsion integrals within coupled-cluster and equation-of-motion methods: Theory and benchmarks. *J. Chem. Phys.*, 139(13): 134105, 2013. URL: <https://doi.org/10.1063/1.4820484>, doi:10.1063/1.4820484

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 single-electron excitation amplitudes.
- **t2** (`Optional[ndarray]`, default: `None`) – Guess two-electron 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 single-electron excitation amplitudes.
- **t2** (*Optional[ndarray]*, default: None) – Guess two-electron excitation amplitudes.
- **compact** (*bool*, default: False) – Enable compact form of excitation for the restricted wavefunctions.
- **antihermitian** (*bool*, default: True) – Returns anti-Hermitian operators $T - T^\dagger$ if specified.

Returns

FermionOperatorList – Excitation operator list.

get_lowdin_system(*method='lowdin'*, *store_ao=False*)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

- **method** (*str*, default: "lowdin") – Method passed to PySCF's `orth.orth_ao()`.
- **store_ao** (*bool*, default: False) – If True, the returned Hamiltonian operator is of type *PySCFChemistryRestrictedIntegralOperator* or *PySCFChemistryUnrestrictedIntegralOperator*, which stores the underlying atomic orbitals internally.

Returns

Tuple[Union[ChemistryRestrictedIntegralOperator, ChemistryUnrestrictedIntegralOperator, PySCFChemistryRestrictedIntegralOperator, PySCFChemistryUnrestrictedIntegralOperator], FermionSpace, Union[RestrictedOneBodyRDM, UnrestrictedOneBodyRDM]] – Fermion Hamiltonian, Fock space, Fock state.

↳ See also

`orth.orth_ao` documentation.

get_mulliken_pop()

Interface for Mulliken population analysis of PySCF.

Returns

Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, 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, ..., pdm4).

Returns

The NEVPT2 correction to the energy.

get_orbital_coefficients()

Returns orbital coefficients.

Returns

`ndarray` – Orbital coefficients.

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_co, bound= generic, covariant=True)]]` – Two-body reduced density matrix.

Note

This object will be replaced with an `RestrictedTwoBodyRDM` class to avoid returning a raw 4D tensor.

get_subsystem_driver(frozen=None, transf=None)

Generate a driver object wrapping the current active space Hamiltonian.

Parameters

- **frozen** (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`) – Frozen orbital information (applies to the subsystem driver).
- **transf** (`Union[Callable[[array], array], Transf]`, default: `None`) – Orbital transformer (applies to the subsystem driver).

Returns

`BasePySCFDriver` – PySCF driver object wrapping the current active space Hamiltonian.

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, Chem-`

istryUnrestrictedIntegralOperatorCompact], 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.

`make_actives_contiguous()`

Reorder orbitals so that active orbitals form a contiguous block.

Reorders orbital coefficients, orbital energies, occupations and the list of frozen (active) orbitals. Implemented for RHF, ROHF and UHF.

`property mf_energy: float`

Return the total mean-field energy.

Returns

Total mean-field energy.

`property mf_type: str`

Return the mean-field type as a string, with options including "RHF", "ROHF", and "UHF".

Returns

Mean-field type name.

`property n_electron: int`

Return the total number of electrons in the active space.

Returns

Total number of electrons.

`property n_orb: int`

Return the number of spatial orbitals.

Returns

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.

set_checkfile (chkfile, init_guess=True)

Set checkpoint file name.

The PySCF calculation results will be saved to the checkpoint file.

Parameters

- `chkfile` (`str`) – name of checkpoint file.
- `init_guess` (`bool`, default: `True`) – If `True` and the checkpoint file exists, the initial guess will be read from the checkpoint file.

set_diis_space_dimension (diis_space_dimension=8)

Set number of DIIS vectors.

Parameters

`diis_space_dimension` (`int`, default: 8) – dimension of the DIIS space.

set_init_orbitals (init_orbs)

Sets the initial guess orbitals for the SCF.

Parameters

`init_orbs` (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]`) – Initial orbital coefficients.

Return type

`None`

set_level_shift (level_shift_value=0)

Set value of the artificial shift applied to virtual orbitals during SCF.

Note

Level shifting in PySCF changes the HF energy, even though the last SCF iteration is performed without it.

Parameters

`level_shift_value` (`float`, default: 0) – value of the level shift parameter.

`set_max_scf_cycles` (`max_cycles=50`)

Set maximum number of SCF cycles.

Parameters

`max_cycles` (`int`, default: 50) – maximum number of SCF cycles.

```
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,
                                                    solvent_epsilon=COSMO_SOLVENT['water'],
                                                    verbose=0, output=None,
                                                    point_group_symmetry=False,
                                                    point_group_symmetry_subgroup=None, soscf=False,
                                                    df=False)
```

PySCF driver for molecular UHF, with Quantum Mechanics - Molecular Mechanics (QMMM) and COnductor-like Screening MOdel (COSMO) calculations.

Parameters

- `geometry` (`Union[List, str]`, default: None) – Molecular geometry.
- `zmatrix` (`str`, default: None) – Z matrix representation of molecular geometry (Used only if `geometry` is not specified).
- `basis` (`Any`, default: None) – Atomic basis set valid for `Mole` class.
- `ecp` (`Any`, default: None) – Effective core potentials.
- `charge` (`int`, default: 0) – Total charge.
- `multiplicity` (`int`, default: 1) – Spin multiplicity, 2S+1.
- `frozen` (`Any`, default: None) – Frozen orbital information.
- `do_qmmm` (`bool`, default: None) – If True, do QMMM embedding. MM geometry and charges are required.
- `mm_charges` (`Any`, default: None) – Point charge values for MM region.
- `mm_geometry` (`Any`, default: None) – Geometry of MM region.
- `do_mm_coulomb` (`bool`, default: None) – If True, calculate MM-MM Coulomb interaction energy. Needs to be added as a constant to the final energy: $\langle \mathbf{H} \rangle + \text{self.e_mm_coulomb}$
- `do_cosmo` (`bool`, default: None) – If True, adds implicit water (COSMO) to the mean-field calculation. `get_system_legacy()` returns $\mathbf{H} + \mathbf{v}_{\text{solvent}}$ as fermion operator. The resulting final energy needs to be corrected as $\langle \mathbf{H} + \mathbf{v}_{\text{solvent}} \rangle + \text{self.cosmo_correction}$. Where `self.cosmo_correction = E_cosmo - \text{Tr}[\mathbf{v}_{\text{solvent}} * \mathbf{rdm1}]`. If `do_cosmo=True`, `self.cosmo_correction` is defined after `self._run_hf()`.

- **solvent_epsilon** (`float`, default: `COSMO_SOLVENT["water"]`) – Dielectric constant of the solvent in the COSMO model.
- **verbose** (`int`, default: 0) – Control SCF verbosity.
- **output** (`str`, default: `None`) – Specify log file name. If `None`, logs are printed to `STDOUT`.
- **point_group_symmetry** (`bool`, default: `False`) – Enable point group symmetry.
- **point_group_symmetry_subgroup** (`str`, default: `None`) – Use this point group symmetry instead of full point group symmetry.
- **soscf** (`bool`, default: `False`) – Use Second-Order SCF solver (Newton's method).
- **df** (`bool`, default: `False`) – Use density fitting.

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` (default: `True`) – 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]]]]]`, 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.

`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_ac0_correction(rdm)`

Compute the AC0 correction to the energy from the provided density matrices.

Requires installation of `pyscf-ac0` extension, which is partially based on `GAMMCOR` and is distributed separately.

Parameters

`rdm` (`Tuple[ndarray[Any, dtype[float]], ndarray[Any, dtype[float]]]`) – A tuple of one- and two-particle reduced density matrices as RDMs as `(rdm1, rdm2)`.

Returns

The AC0 correction to the energy.

See also

For more information on AC0 correction, refer to: - DOI: 10.1021/acs.jctc.8b00213

`get_cascl_1234pdms()`

Calculate 1-, 2-, 3- and 4-PDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1-, 2-, 3- and 4-PDMs.

`get_cascl_12rdms()`

Calculate 1-RDM and 2-RDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1- and 2-RDMs.

`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_double_factorized_system(tol1=-1, tol2=None, method=DecompositionMethod.EIG, diagonalize_one_body=True, diagonalize_one_body_offset=True, combine_one_body_terms=True)`

Calculate double-factorized Hamiltonian operator, Fock space, and Hartree-Fock state.

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 which is to be double-factorized, and S is a one-body energy offset 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.

First level of factorization decomposes the electron repulsion integral (ERI) tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or pivoted, incomplete Cholesky decomposition (`method='cho'`). For details about Cholesky decomposition, refer to References [1] and [2]. 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 the first factorization stage, truncation depends on the decomposition method. For ‘eig’, we discard eigenvalues, starting from the smallest, until the sum of those discarded exceeds `tol1`. For ‘cho’, the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. 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 decomposition of ERI matrix. If negative, no truncation is performed.
- `tol2` (`Optional[float]`, default: None) – Truncation threshold for second decomposition. If None, same as `tol1`.
- `method` (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`) – Decomposition method used for the first level of factorization. ‘eig’ for an eigenvalue decomposition, ‘cho’ for a pivoted, incomplete Cholesky decomposition.
- `diagonalize_one_body` (`bool`, default: True) – Whether to diagonalize the physical one-body integrals h_{pq} .
- `diagonalize_one_body_offset` (`bool`, default: True) – Whether to diagonalize the one-body offset integrals s_{pq} .
- `combine_one_body_terms` (`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

`Tuple[DoubleFactorizedHamiltonian, FermionSpace, FermionState]` – Hamiltonian operator storing two-body integrals in double factorized form, Fock space, Fock state.

References

1. Henrik Koch, Alfredo Sánchez de Merás, and Thomas Bondo Pedersen. Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.*, 118(21): 9481-9484, 2003. URL: <https://doi.org/10.1063/1.1578621>, doi:10.1063/1.1578621
2. Evgeny Epifanovsky, Dmitry Zuev, Xintian Feng, Kirill Khistyayev, Yang Shao, and Anna I. Krylov. General implementation of the resolution-of-the-identity and Cholesky representations of electron

repulsion integrals within coupled-cluster and equation-of-motion methods: Theory and benchmarks. J. Chem. Phys., 139(13): 134105, 2013. URL: <https://doi.org/10.1063/1.4820484>, doi:10.1063/1.4820484

`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 single-electron excitation amplitudes.
- **t2** (*Optional[ndarray]*, default: None) – Guess two-electron 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 single-electron excitation amplitudes.
- **t2** (*Optional[ndarray]*, default: None) – Guess two-electron excitation amplitudes.
- **compact** (*bool*, default: False) – Enable compact form of excitation for the restricted wavefunctions.
- **antihermitian** (*bool*, default: True) – Returns anti-Hermitian operators $T - T^\dagger$ if specified.

Returns

FermionOperatorList – Excitation operator list.

`get_lowdin_system`(*method*='lowdin', *store_ao*=False)

Calculate Hamiltonian operator (fermion) in orthogonalized AO, Fock space.

Parameters

- **method** (*str*, default: "lowdin") – Method passed to PySCF's `orth.orth_ao()`.
- **store_ao** (*bool*, default: False) – If True, the returned Hamiltonian operator is of type *PySCFChemistryRestrictedIntegralOperator* or *PySCFChemistryUnrestrictedIntegralOperator*, which stores the underlying atomic orbitals internally.

Returns

Tuple[Union[ChemistryRestrictedIntegralOperator, ChemistryUnrestrictedIntegralOperator, PySCFChemistryRestrictedIntegralOperator, PySCFChemistryUnrestrictedIntegralOperator], FermionSpace]

`Union[RestrictedOneBodyRDM, UnrestrictedOneBodyRDM]]` – Fermion Hamiltonian, Fock space, Fock state.

See also

`orth.orth_ao` documentation.

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_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – Mulliken population, Mulliken atomic charges.

get_nevpt2_correction(rdm)

Compute the Strongly contracted NEVPT2 correction to the energy from the provided density matrices.

Parameters

`rdm` (`Tuple`) – A tuple of reduced density matrices as PDMs in PySCF's un-reordered format as `(pdm1, ..., pdm4)`.

Returns

The NEVPT2 correction to the energy.

get_orbital_coefficients()

Returns orbital coefficients.

Returns

`ndarray` – Orbital coefficients.

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_co, bound= generic, covariant=True)]]` – Two-body reduced density matrix.

Note

This object will be replaced with an `RestrictedTwoBodyRDM` class to avoid returning a raw 4D tensor.

`get_subsystem_driver` (`frozen=None, transf=None`)

Generate a driver object wrapping the current active space Hamiltonian.

Parameters

- `frozen` (`Union[List[int], List[List[int]], Callable[[SCF], Union[List[int], List[List[int]]]]]`, default: `None`) – Frozen orbital information (applies to the subsystem driver).
- `transf` (`Union[Callable[[array], array], Transf]`, default: `None`) – Orbital transformer (applies to the subsystem driver).

Returns

`BasePySCFDriver` – PySCF driver object wrapping the current active space Hamiltonian.

`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.

make_actives_contiguous ()
Reorder orbitals so that active orbitals form a contiguous block.

Reorders orbital coefficients, orbital energies, occupations and the list of frozen (active) orbitals. Implemented for RHF, ROHF and UHF.

property mf_energy: float
Return the total mean-field energy.

Returns
Total mean-field energy.

property mf_type: str
Return the mean-field type as a string, with options including "RHF", "ROHF", and "UHF".

Returns
Mean-field type name.

property n_electron: int
Return the total number of electrons in the active space.

Returns
Total number of electrons.

property n_orb: int
Return the number of spatial orbitals.

Returns
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 `mcsf.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.

`set_checkfile(chkfile, init_guess=True)`

Set checkpoint file name.

The PySCF calculation results will be saved to the checkpoint file.

Parameters

- `chkfile (str)` – name of checkpoint file.
- `init_guess (bool, default: True)` – If True and the checkpoint file exists, the initial guess will be read from the checkpoint file.

`set_diis_space_dimension(diis_space_dimension=8)`

Set number of DIIS vectors.

Parameters

`diis_space_dimension (int, default: 8)` – dimension of the DIIS space.

`set_init_orbitals(init_orbs)`

Sets the initial guess orbitals for the SCF.

Parameters

`init_orbs (ndarray[Any, dtype[TypeVar(_ScalarType_co, bound=generic, covariant=True)]]])` – Initial orbital coefficients.

Return type

`None`

`set_level_shift(level_shift_value=0)`

Set value of the artificial shift applied to virtual orbitals during SCF.

Note

Level shifting in PySCF changes the HF energy, even though the last SCF iteration is performed without it.

Parameters

`level_shift_value (float, default: 0)` – value of the level shift parameter.

`set_max_scf_cycles(max_cycles=50)`

Set maximum number of SCF cycles.

Parameters

`max_cycles (int, default: 50)` – maximum number of SCF cycles.

```
class ChemistryDriverPySCFMomentumRHF(geometry=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,
                                         space_group_symmetry=False, precision=1e-9, soscf=False)
```

PySCF driver for momentum-space RHF calculations.

Parameters

- `geometry (Union[List, str], default: None)` – Molecular geometry.
- `cell (List[List], default: None)` – Unit cell parameter. If provided, overrides `geometry`.
- `unit_cell.`

- **nks** (`List[int]`, default: `None`) – Number of k-points for each direction.
- **basis** (`Any`, default: `None`) – Atomic basis set valid for `Mole` class.
- **ecp** (`Any`, default: `None`) – Effective core potentials.
- **pseudo** (`Any`, default: `None`) – Pseudo potentials.
- **charge** (`int`, default: `0`) – Total charge.
- **exp_to_discard** (`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.
- **verbose** (`int`, default: `0`) – Control SCF verbosity.
- **output** (`str`, default: `None`) – Specify log file name. If `None`, logs are printed to `STDOUT`.
- **frozen** (`Any`, default: `None`) – Frozen orbital information.
- **space_group_symmetry** (`bool`, default: `False`) – Whether to use space group symmetry.
- **precision** (`float`, default: `1e-9`) – Ewald sum precision.
- **soscf** (`bool`, default: `False`) – Use second-order SCF (Newton converger).

`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]]]]]`, 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.

`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_casci_1234pdms()`

Calculate 1-, 2-, 3- and 4-PDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1-, 2-, 3- and 4-PDMs.

`get_casci_12rdms()`

Calculate 1-RDM and 2-RDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1- and 2-RDMs.

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.

Returns

Total mean-field energy.

property mf_type: str

Return the mean-field type as a string (e.g. “RHF”).

Returns

Mean-field type name.

property n_electron: int

Number of electrons in the active space.

Returns

Number of electrons in the active space.

property n_kp: int

Number of k points.

Returns

Number of k points.

property n_orb: int

Number of spatial orbitals in a unit cell.

Returns

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.

set_init_orbitals(init_orbs)

Sets the initial guess orbitals for the SCF.

Parameters

`init_orbs`(ndarray[`Any`, dtype[`TypeVar(_ScalarType_co, bound=generic, covariant=True)]]) – Initial orbital coefficients.`

Return type

`None`

```
class ChemistryDriverPySCFMomentumROHF(geometry=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,
                                         space_group_symmetry=False, precision=1e-9, soscf=False)
```

PySCF driver for momentum-space ROHF calculations.

Parameters

- `geometry`(Union[List, str], default: None) – Molecular geometry.
- `cell`(List[List], default: None) – Unit cell parameter. If provided, overrides `geometry`. `unit_cell`.
- `nks`(List[int], default: None) – Number of k-points for each direction.
- `basis`(Any, default: None) – Atomic basis set valid for `Mole` class.
- `ecp`(Any, default: None) – Effective core potentials.
- `pseudo`(Any, default: None) – Pseudo potentials.

- **multiplicity** (`int`, default: 1) – Spin multiplicity, $2S+1$.
- **charge** (`int`, default: 0) – Total charge.
- **exp_to_discard** (`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.
- **verbose** (`int`, default: 0) – Control SCF verbosity.
- **output** (`str`, default: `None`) – Specify log file name. If `None`, logs are printed to STDOUT.
- **frozen** (`Any`, default: `None`) – Frozen orbital information.
- **space_group_symmetry** (`bool`, default: `False`) – Whether to use space group symmetry.
- **precision** (`float`, default: $1e-9$) – Ewald sum precision.
- **soscf** (`bool`, default: `False`) – Use second-order SCF (Newton converger).

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]]]]]`, 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.

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_casci_1234pdms()

Calculate 1-, 2-, 3- and 4-PDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1-, 2-, 3- and 4-PDMs.

get_casci_12rdms()

Calculate 1-RDM and 2-RDM from a CASCI wavefunction.

Returns

`Tuple[ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – A tuple of 1- and 2-RDMs.

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.

Returns

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.

Returns

Total mean-field energy.

```
property mf_type: str
```

Return the mean-field type as a string (e.g. “RHF”).

Returns

Mean-field type name.

```
property n_electron: int
```

Number of electrons in the active space.

Returns

Number of electrons in the active space.

```
property n_kp: int
```

Number of k points.

Returns

Number of k points.

```
property n_orb: int
```

Number of spatial orbitals in a unit cell.

Returns

Number of spatial orbitals in a unit cell

```
print_json_report(*args, **kwargs)
```

Prints report in json format.

```
run_cascl(**kwargs)
```

Calculate the CASCI energy.

Parameters

****kwargs** – Keyword arguments to set attributes of the `mcsfc.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.

`set_init_orbitals`(*init_orbs*)

Sets the initial guess orbitals for the SCF.

Parameters

`init_orbs` (ndarray[`Any`, dtype[`TypeVar(_ScalarType_co, bound= generic, covariant=True)`]]) – Initial orbital coefficients.

Return type

`None`

`class DMETRHFFragmentPySCFActive`(*dmet, mask, name=None, frozen=None*)

Custom 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 a `solve_active()` method to solve the fragment problem within the chosen active space of orbitals, 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_co, 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` (`str`, default: `None`) – Optional name of the fragment.
- `frozen` (`Union[List[int], Callable[[SCF], List[int]]]`, 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 democratic mixing defined by DMET.

```
solve(hamiltonian_operator, fragment_energy_operator, n_electron)
```

Solves the fragment system using PySCF's RHF and a user-implemented *solve_active()* method.

First it performs an RHF calculation and the active space problem will be passed on to the *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_co, 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, 1-RDM 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_co, 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` (`str`, default: `None`) – Optional name of the fragment.
- `frozen` (`Union[List[int], Callable[[SCF], List[int]]]`, 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_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]] – Energy, 1-RDM, 2-RDM of the embedding system (fragment and bath).`

```
class DMETRHFFragmentPySCFCI(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** (Optional[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 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_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, 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_co, 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** (str, default: None) – Optional name of the fragment.
- **frozen** (*Union[List[int], Callable[[SCF], List[int]]]*, 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_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, 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** (*Optional[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_co, bound= generic, covariant=True)]], ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]]` – Energy, 1-RDM, 2-RDM of the embedding system (fragment and bath).

class FromActiveOrbitals (*active_orbitals*)

Selection of active orbitals.

This class helps specify the list of active orbitals based on user-provided information. See below for a HOMO-LUMO CI calculation for minimal basis LiH, where the use of this class with `FromActiveOrbitals(active_orbitals = [1, 2])` is equivalent to the explicit list `[0, 3, 4, 5, 6]`.

Parameters

`active_orbitals` (`List[int]`) – List of indices of active orbitals.

Note

Invalid (i.e., non-existent) orbital indices are ignored.

Examples

```
>>> from inquanto.extensions.pyscf import ChemistryDriverPySCFMolecularRHF, _
    <FromActiveOrbitals>
>>> driver = ChemistryDriverPySCFMolecularRHF(
...     geometry='Li 0 0 0; H 0 0 1.75',
...     basis='sto3g',
...     frozen=FromActiveOrbitals(active_orbitals = [1, 2]),
... )
```

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.

i 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.

i 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.

i 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_co, 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** (`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]]]`, default: `None`) – Frozen orbital information.

`solve` (*hamiltonian_operator*, *n_electron*)

Solves the fragment system using PySCF's ROHF and `solve_active()` method.

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** (`str`, default: `None`) – Name of the fragment.
- **multiplicity** (`int`, default: 1) – Multiplicity for the fragment ROHF.
- **frozen** (`Union[List[int], Callable[[SCF], List[int]]]`, 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** (*Optional[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_co, bound= generic, covariant=True)]]*) – 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.
- **frozen** (*Union[List[int], Callable[[SCF], List[int]]]*, 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** (*Optional[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 values 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=-1.0, tol2=None, method=DecompositionMethod.EIG, 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.

The Hamiltonian can be written as $\hat{H} = H_0 + \hat{H}_1 + \hat{S} + \hat{V}$ where $\hat{S} + \hat{V}$ is the Coulomb interaction. $V = \frac{1}{2} \sum_{ijkl} (ij|kl) a_i^\dagger a_j a_k^\dagger a_l$ is a reordered two-body operator and \hat{S} is a one-body offset term given by $\hat{S} = \sum_{ij} s_{ij} a_i^\dagger a_j$ where $s_{ij} = -\frac{1}{2} \sum_k (ik|kj)$. H_0 and \hat{H}_1 are the constant and one-electron terms respectively.

The first level of factorization decomposes the electron repulsion integrals tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or a pivoted, incomplete Cholesky decomposition (`method='cho'`) (see [J. Chem. Phys. 118, 9481–9484 \(2003\)](#) and [J. Chem. Phys. 139, 134105 \(2013\)](#)). 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 the first factorization stage, truncation depends on the decomposition method. With "`eig`", truncation is performed by discarding eigenvalues, starting from the smallest in magnitude, until the sum of those discarded magnitudes exceeds the threshold `tol1`. With "`cho`", the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding low-magnitude eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. One-body diagonalization is not truncated.

 **Warning**

This is 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: `-1.0`) – 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.
- `method` (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`)
 - Decomposition method used for the first level of factorization. "`eig`" for an eigenvalue decomposition, "`cho`" for a pivoted, incomplete Cholesky decomposition.
- `diagonalize_one_body` (`bool`, default: `True`) – Whether to diagonalize the physical one-body integrals h_{pq} .
- `diagonalize_one_body_offset` (`bool`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq}
- `combine_one_body_terms` (`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, diagonalized 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_co, bound= generic, covariant=True)]]` – Effective potential matrix.

`effective_potential_spin(rdm1)`

Calculates the contribution to the effective Coulomb potential due to a spin imbalance.

Parameters

`rdm1` (`RestrictedOneBodyRDM`) – Restricted, one-body reduced density matrix object.

Returns

`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]` – Effective potential matrix.

`energy(rdm1, rdm2=None)`

Calculates the 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, qubits=None)`

Performs qubit encoding (mapping), using provided mapping class, of this instance.

This proceeds by creation of a `FermionOperator` class, via the `to_FermionOperator()` method, before mapping to a `QubitOperator` using the provided mapping scheme. Please see the documentation for `to_FermionOperator()` for finer grained control over operator generation.

Parameters

- `mapping` (`Optional[QubitMapping]`, default: `None`) – `QubitMapping`-derived instance. Default mapping procedure is `QubitMappingJordanWigner`.
- `qubits` (`Optional[List[Qubit]]`, default: `None`) – The qubit register. If left as `None`, a default register will be assumed if possible. See `QubitMapping` documentation for further details.

Returns`QubitOperator` – Mapped `QubitOperator` object.`rotate(rotation, check_unitary=True, check_unitary_atol=1e-15)`

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_co, bound= generic, covariant=True)]]`) – Orthogonal rotation matrix.
- `check_unitary` (`bool`, default: `True`) – If True, performs a unitarity check on the rotation matrix.
- `check_unitary_atol` (`float`, default: `1e-15`) – Absolute tolerance for the unitarity check.

Returns`PySCFChemistryRestrictedIntegralOperator` – self after molecular orbital rotation.`run_rhf()`

Run a restricted Hartree-Fock calculation.

Sets MO coefficients internally.

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.

Parameters

- **yield_constant** (`bool`, default: `True`) – Whether to include a constant term.
- **yield_one_body** (`bool`, default: `True`) – Whether to include one-body terms.
- **yield_two_body** (`bool`, default: `True`) – Whether to include two-body terms.

Returns

FermionOperator – Integral operator in general fermionic operator form.

class PySCFChemistryUnrestrictedIntegralOperator(mol_or_mf, run_hf=True)

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.
- **run_hf** (`bool`, default: `True`)

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 values 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=-1.0, tol2=None, method=DecompositionMethod.EIG, 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.

The Hamiltonian can be written as $\hat{H} = H_0 + \hat{H}_1 + \hat{S} + \hat{V}$ where $\hat{S} + \hat{V}$ is the Coulomb interaction. $V = \frac{1}{2} \sum_{ijkl} (ij|kl) a_i^\dagger a_j a_k^\dagger a_l$ is a reordered two-body operator and \hat{S} is a one-body offset term given by $\hat{S} = \sum_{ij} s_{ij} a_i^\dagger a_j$ where $s_{ij} = -\frac{1}{2} \sum_k (ik|kj)$. H_0 and \hat{H}_1 are the constant and one-electron terms respectively.

The first level of factorization decomposes the electron repulsion integrals tensor into the form: $(pq|rs) = \sum_t^{N_\gamma} V_{pq}^t \gamma^t V_{rs}^t$. This may be performed using an eigenvalue decomposition (`method='eig'`), or a pivoted, incomplete Cholesky decomposition (`method="cho"`) (see [J. Chem. Phys. 118, 9481–9484 \(2003\)](#) and [J. Chem. Phys. 139, 134105 \(2013\)](#)). The second factorization is diagonalization of the V_{pq}^t matrix for each t: $V_{pq}^t = \sum_u^{N_\lambda} U_{pu}^t \lambda_u^t U_{qu}^t$.

At the first factorization stage, truncation depends on the decomposition method. With "eig", truncation is performed by discarding eigenvalues, starting from the smallest in magnitude, until the sum of those discarded magnitudes exceeds the threshold `tol1`. With "cho", the decomposition is constructed iteratively until the error is less than `tol1`. At the second factorization level, truncation is always performed by discarding low-magnitude eigenvalues.

One-body-like terms are consolidated and diagonalized by default: $\tilde{h}_{pq} = h_{pq} + s_{pq} = \sum_r W_{pr} \omega_r W_{qr}$. One-body diagonalization is not truncated.

 **Warning**

This is 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: `-1.0`) – 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.
- `method` (`Union[DecompositionMethod, str]`, default: `DecompositionMethod.EIG`)
 - Decomposition method used for the first level of factorization. "eig" for an eigenvalue decomposition, "cho" for a pivoted, incomplete Cholesky decomposition.
- `diagonalize_one_body` (`bool`, default: `True`) – Whether to diagonalize the physical one-body integrals h_{pq} .
- `diagonalize_one_body_offset` (`bool`, default: `True`) – Whether to diagonalize the one-body offset integrals s_{pq}

- `combine_one_body_terms` (`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, diagonalized 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`)

Calculates the 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, qubits=None)

Performs qubit encoding (mapping), using provided mapping class, of this instance.

This proceeds by creation of a `FermionOperator` class, via the `to_FermionOperator()` method, before mapping to a `QubitOperator` using the provided mapping scheme. Please see the documentation for `to_FermionOperator()` for finer grained control over operator generation.

Parameters

- `mapping` (`Optional[QubitMapping]`, default: `None`) – `QubitMapping`-derived instance. Default mapping procedure is `QubitMappingJordanWigner`.
- `qubits` (`Optional[List[Qubit]]`, default: `None`) – The qubit register. If left as `None`, a default register will be assumed if possible. See `QubitMapping` documentation for further details.

Returns

`QubitOperator` – Mapped `QubitOperator` object.

rotate(rotation_aa, rotation_bb, check_unitary=True, check_unitary_atol=1e-15)

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_co, bound= generic, covariant=True)]]`) – Orthogonal rotation matrix for the alpha spin block.
- `rotation_bb` (`ndarray[Any, dtype[TypeVar(_ScalarType_co, bound= generic, covariant=True)]]`) – Orthogonal rotation matrix for the beta spin block.
- `check_unitary` (`bool`, default: `True`) – If True, performs a unitarity check on the rotation matrix.
- `check_unitary_atol` (`float`, default: `1e-15`) – Absolute tolerance for the unitarity check.

Returns

`PySCFChemistryUnrestrictedIntegralOperator` – self after molecular orbital rotation.

run_uhf()

Run an unrestricted Hartree-Fock calculation.

Sets MO coefficients internally.

Raises

`RuntimeError` – If the UHF calculation does not converge.

Returns

`ndarray[Any, dtype[float]]` – MO coefficient matrices.

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`.

Parameters

- `yield_constant` (`bool`, default: `True`) – Whether to include a constant term.
- `yield_one_body` (`bool`, default: `True`) – Whether to include one-body terms.
- `yield_two_body` (`bool`, default: `True`) – Whether to include two-body terms.

Returns

`FermionOperator` – Integral operator in general fermionic operator form.

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_co, 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.

28.1.1 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`

Basic FMO driver class.

This class serves as a basic driver for the Fragment Molecular Orbital (FMO) method.

Parameters

- **hamiltonian_operator** (`Union[ChemistryRestrictedIntegralOperator, PySCFChemistryRestrictedIntegralOperator]`) – Stores integrals for full system in orthogonal localized basis.
- **scf_max_iteration** (`int`, default: 20) – Maximum iteration count for the outer self-consistent field (SCF) loop.
- **scf_tolerance** (`float`, default: `1e-5`) – Convergence tolerance defined as $|energy - energy'| < scf_{tolerance}$ for the outer loop.

References

1. Kazuo Kitaura, Eiji Ikeo, Toshio Asada, Tatsuya Nakano, and Masami Uebayasi. Fragment molecular orbital method: an approximate computational method for large molecules. *Chem. Phys. Lett.*, 313(3-4): 701-706, 1999. URL: [https://doi.org/10.1016/S0009-2614\(99\)00874-X](https://doi.org/10.1016/S0009-2614(99)00874-X), doi:10.1016/S0009-2614(99)00874-X
2. Takeshi Yamazaki, Shunji Matsuura, Ali Narimani, Anushervon Saidmuradov, and Arman Zaribafyan. Towards the practical application of near-term quantum computers in quantum chemistry simulations: A problem decomposition approach. arXiv:1806.01305, 2018. arXiv:1806.01305

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 FMO energies.

Parameters

- `fragments` (`List[FMOFragment]`) – List of FMO fragment solver.
- `dimer_fragments` (`Dict[Tuple, FMOFragment]`) – Dictionary of FMO fragment solvers 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

- **hamiltonian_operator** (*ChemistryRestrictedIntegralOperator*) – Hamiltonian operator of the fragment system.
- **fermion_space** (*FermionSpace*) – Fermion active space.
- **fermion_state** (*FermionState*) – Fermion state.

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.

28.2 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** (`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** (`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** (`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** (default: `None`) – Label type for atoms, can be “index”, “symbol” or `None`.

Returns

`NGLWidget` – An ngl widget showing the unit cell.

28.3 inquanto-phayes

InQuanto Phayes extension.

```
class AlgorithmBayesianQPE(phayes_state, k_max=None, error_rate=None, verbose=0)
```

Execute Bayesian QPE algorithm.

This class reproduces the workflow in [arXiv:2306.16608](https://arxiv.org/abs/2306.16608).

The Bayesian update part is based on the Quantinuum's `phayes` package.

Parameters

- `phayes_state` (`PhayesState`) – Initial state as a `PhayesState` object.
- `k_max` (`Optional[int]`, default: `None`) – Cap of the number of repeats of the CTRL-U circuit.
- `error_rate` (`Optional[Callable[[int, float], float]]`, default: `None`) – Error rate to be used for the noise-aware likelihood.
- `verbose` (`int`, default: `0`) – Control the verbosity.

```
build(protocol)
```

Set the IQPE protocol object.

Parameters

`protocol` (`BaseIterativePhaseEstimation`) – IQPE protocol to be used for handling the circuit.

Returns

`AlgorithmBayesianQPE` – self

```
final_pdf(phi)
```

Return the PDF as a function of phase.

Parameters

`phi` (`ndarray`) – Grid representation of the phase in [0, 2) (pytket convention).

Returns

`ndarray` – Probability distribution function.

```
final_value()
```

Return the energy estimate.

Returns

`tuple[float, float]` – Mean and the square root of the Holevo variance.

```
property has_updated: bool
```

Indicate if the Bayesian update is performed or not.

It returns `False` if no measurement outcome is available for some reasons such as discarding the measurement outcome by the error detection code. Repeat until success is effectively performed by calling `run()` again.

Returns

`True` if the Bayesian update is performed.

join (*handles_mapping*)

Retrieve the backend results through the protocol.

Parameters

handles_mapping (`List[Tuple[int, float, ResultHandle]]`) – List of job IDs (k, β , result handles).

property phayes_state: PhayesState

Current PhaseState object.

run ()

Run the algorithm.

run_async ()

Run the jobs asynchronously.

Returns

`List[Tuple[int, float, List[ResultHandle]]]` – List of job IDs (k, β , result handles).

CHAPTER
TWENTYNINE

CHANGELOG

29.1 InQuanto 3.7.0

22 October 2024

- Added asynchronous calibration to `SPAM` so hardware delivery is controllable
- Licensing modified for improved Quantinuum Nexus usage

29.2 InQuanto 3.6.1

3 September 2024

- Add `QCM4Computable` computable for quantum computed moments up to the 4th order
- Make remaining protocol classes compatible with `ProtocolList` by adding `build_protocols_from()` method definitions
- New examples for the factorized overlap protocol classes (e.g. `FactorizedOverlap`) with comparisons to `HadamardTestOverlap`
- Improve the information given in the dataframe outputted by protocols using the `dataframe_circuit_shot()` function
- Add `to_QubitPauliString()`, `from_QubitPauliString()` methods to `QubitOperatorString` class, and `to_QubitPauliOperator()`, and `from_QubitPauliOperator()` methods to the `QubitOperator` class to allow the conversion of `QubitOperator` to/from the pytket accessible `pytket.utils.QubitPauliOperator`
- Add `all_nontrivial_qubits()` property to `QubitOperator`, `QubitOperatorString`, and `QubitOperatorList` classes
- InQuanto now throws a warning when there are dependency conflicts or required upgrades
- Ensured calls to `numpy.einsum` correctly scale via `optimize` keyword argument
- Make InQuanto installable with poetry
- Bug fixes:
 - Fixed `load_xyz()` defaulting to Geometry base class when called
 - Fix bug where integer type parameters could cause unwarranted rounding of non-integer quantities giving incorrect results

29.3 InQuanto 3.5.8

23 August 2024

- Hotfix release latest pytket version

29.4 InQuanto 3.5.7

2 August 2024

- Hotfix pytket v1.31 dependency issues

29.5 InQuanto 3.5.6

10 July 2024

- Hotfix sympy v1.13 dependency issues

29.6 InQuanto 3.5.5

4 June 2024

- Upgraded Zeus dependency to v0.0.1b6

29.7 InQuanto 3.5.4

13 May 2024

- Upgraded Sourcedefender dependency to v14.0

29.8 InQuanto 3.5.3

10 May 2024

- Upgraded Sourcedefender dependency to v13.0

29.9 InQuanto 3.5.2

10 May 2024

- Upgraded Sourcedefender dependency to v12.0

29.10 InQuanto 3.5.1

07 May 2024

- Resolved scipy dependency conflict.
- Bug fixes:
 - Fixed a slowdown in qubit mappings caused in 3.5.0. Mapping speed should now be consistent with InQuanto versions <3.5.0.

- `qubit_encode()` for integral operators now takes the `qubits` argument, allowing `QubitMapping-BravyiKitaev` support.

29.11 InQuanto 3.5.0

24 April 2024

- Introducing new protocols: `ComputeUncomputeFactorizedOverlap` and `SwapFactorizedOverlap` for computing complex overlaps.
- Expanded `ProtocolList` support to the following averaging protocols:
 - `HadamardTest`
 - `SwapTest`
 - `ComputeUncompute`
 - `DestructiveSwapTest`
 - `HadamardTestOverlap`
 - `SwapFactorizedOverlap`
 - `ComputeUncomputeFactorizedOverlap`
- End-to-end runners for averaging protocols now support symbolic compilation with the `compile_symbolic` argument (see `get_runner()`).
- Protocols now allow control over pytket optimisation_level arg at the `build()` step.
- Updated pandas dependency to version 2.2.
- Added the `qubit_encode()` method for `FermionState` and `FermionStateString`.
- Bug fix: The `complex_type` arg in `HadamardTestDerivative` now supports string input.

29.12 InQuanto 3.4.2

11 April 2024

- Update examples algorithm_vqe_varCI.py and algorithm_vqe_varCI_h3p.py to use up-to-date `MultiConfigurationAnsatz` API
- Bug fixes:
 - Corrected Qiskit API usage in `get_noisy_backend()`
 - `PMSV` input symmetry validation is now more robust
 - Fixed `symbol_substitution()` in `FermionSpaceAnsatzChemicallyAwareUCCSD`

29.13 InQuanto 3.4.1

08 March 2024

- Upgrade Sourcedefender dependency

29.14 InQuanto 3.4.0

07 March 2024

- Significant improvements to circuit depth in `MultiConfigurationState` and `MultiConfigurationAnsatz`
- New ansatz `MultiConfigurationStateBox`, uses pytket's `StatePreparationBox`
- New composite computable `ExpectationValueSumComputable`
- New helper methods in `inquanto.ansatzes`: `rotate_ansatz_restricted()` and `reference_circuit_builder()`
- New examples for `MultiConfigurationStateBox`, `inquanto.operators.ChemistryRestrictedIntegralOperator.double_factorize()`, and using CCSD amplitudes as VQE initial parameters.
- Note: examples are now split between `inquanto` examples and `inquanto-extension` examples
- `disp` (display) option added for `MinimizerSPSA`
- Major improvements to API documentation throughout
- Bug fixes:
 - Fixed '`=`' operator for `SymbolDict`
 - Several fixes in `AlgorithmSCEOM`
 - `symbol_substitution()` with string maps now works for all ansatzes

29.15 InQuanto 3.3.1

31 January 2024

- Bug fix relating to non-symbolic computables in PauliAveraging protocol runner

29.16 InQuanto 3.3.0

30 January 2024

- Support for Python 3.12. Drop support for Python 3.9
- New method `dataframe_partitioning()` in PauliAveraging protocol for seeing relation between Pauli words and circuits.
- Driver for generating transverse-field Ising model hamiltonians (for example `DriverIsing1D`)
- Introducing `ProtocolList`. Basic support for collecting PauliAveraging protocols with different states.
- Bug fix relating to qubit indexing error in symmetry analysis of QubitOperators
- Other bug fixes and development improvements

29.17 InQuanto-PySCF 1.5.0

30 January 2024

- `get_double_factorized_system()` method added for PySCF drivers
- Python 3.12 support and deprecate 3.9

29.18 InQuanto-NGLView v0.7.1

30 January 2024

- Python 3.12 support and deprecate 3.9
- Hotfix dependency update qcelemental

29.19 InQuanto-Phayes v0.2.0

30 January 2024

- Python 3.12 support and deprecate 3.9

29.20 InQuanto 3.2.1

19 January 2024

- Bug fix relating to symbol substitution in PauliAveraging protocol runner

29.21 InQuanto 3.2.0

12 January 2024

- **Three new computables** added
 - `OverlapMatrixComputable` for representing the general overlap matrix between two states $S_{ij} = \langle \Psi_i | \hat{O} | \Psi_j \rangle$
 - `NonOrthogonalMatricesComputable` for representing matrices used by the non-orthogonal quantum eigensolver method
 - `SCEOMMatrixComputable` for representing the Quantum Self Consistent Equation of Motion matrix
- New protocol added `ProjectiveMeasurements` for measuring the probabilities of the basis states
- Added the simultaneous perturbation stochastic approximation (SPSA) minimizer `MinimizerSPSA`
- Added `trotterize_as_linear_combination()` to support second order
- `pd_safe_eigh()` utility method now available to solve generalized eigenvalue problem $HC = SCE$
- Express files added for Purvis' BeH₂ potential energy surface (see the [express manual page](#))
- Added method to report `point_group` from a FermionSpace `point_group()`
- Added Cholesky decomposition option to double factorization
- Added `ensure_hermitian()` to truncate non-hermitian matrix elements
- Minor fix to DMET typing
- Added launching and retrieving methods to QPE algorithms and protocols
- Improved state hashing and comparison methods
- Allow Pauli partitioning in HadamardTestOverlap to reduce measurement circuits (direct option)
- Improved symbol substitution (parameter) errors and flexibility
- Various other minor bug-fixes and improvements

29.22 InQuanto-PySCF 1.4.0

14 December 2023

- Functions to get RDMs and PDMs from CASCI wavefunctions (e.g. `get_casci_12rdms()`)
- Bug fixes

29.23 InQuanto 3.1.2

28 November 2023

- Hotfix for PMSV by adding stabilizer tolerances
- Support for error mitigation when using `protocol.build_from()`
- Clarified the `qubit_operator.eigenspectrum` method

29.24 InQuanto 3.1.1

15 November 2023

- Hotfix: added unitary check options to the `DMETRHF` embedding class

29.25 InQuanto 3.1.0

14 November 2023

- Python 3.11 support
- GraphColouringMethod <https://cqcl.github.io/tket/pytket/api/partition.html#pytket.partition.GraphColourMethod> is now configurable in `PauliAveraging` protocol
- New protocol `HadamardTestOverlap` for computing overlaps
- Various big fixes and small improvements
- Added compilation pass options to `ansatz`'s `to_CircuitAnsatz` method

29.26 InQuanto 3.0.2

17 October 2023

- Added support for `pytket` 1.21
- Updated `GeometryMolecular` to accept xyz strings
- Fixes for FCIDump reader
- Added exact diagonalization helper method to `QubitOperators.eigenspectrum()`

29.27 InQuanto 3.0.1

10 October 2023

- Improved license activation workflow
- Improved API documentation

29.28 InQuanto 3.0.0

19 September 2023

- **Quantum phase estimation** added
 - New algorithm class `AlgorithmDeterministicQPE`
 - New protocols for phase estimation:
 - * `CanonicalPhaseEstimation`
 - * `IterativePhaseEstimationSingleCircuit`
 - * `IterativePhaseEstimation`
 - * `IterativePhaseEstimationQuentinum`
 - * `IterativePhaseEstimationStatevector`
- **Green's functions** added
 - Constructed and run through the use of new composite computables:
 - * `KrylovSubspaceComputable`
 - * `LanczosMatrixComputable`
 - * `LanczosCoefficientsComputable`
 - * `ParticleGFComputable`
 - * `HoleGFComputable`
 - * `ManyBodyGFComputable`
- Added **Qermit integration** for quantum error mitigation: <https://github.com/CQCL/Qermit>
- Additions to the `inquanto.express` module
 - `get_system()` : for quickly retrieving only the hamiltonian, fermion space, and fermion state from an express datafile
 - `save_h5_system()` : for saving a hamiltonian, fermion space, and fermion state to h5 file
 - `get_noisy_backend()` : for generating a simple, configurable, noisy quantum simulator
- Overhaul of the `inquanto.computables` module
 - Computables no longer have responsibility for building or running circuits. They contain ingredients for computing an observable, and relations to other computables in an expression tree. Many have been renamed, and are now categorised into three submodules (* marks a new computable, and [old name if applicable]):
 - `inquanto.computables.primitive` (primitive objects for building computable expression trees):
 - * `ComputableNode`*
 - * `ComputableSingleChild`*
 - * `ComputableFunction`*
 - * `ComputableList`
 - * `ComputableTuple`[`Computables`]
 - * `ComputableNDArray`[`ComputableArray`]
 - `inquanto.computables.atomic` (these interact directly with inquanto protocols):
 - * `ExpectationValue`

- * *ExpectationValueNonHermitian*
- * *ExpectationValueDerivative**
- * *ExpectationValueBraDerivativeReal*
- * *ExpectationValueBraDerivativeImag*
- * *ExpectationValueKetDerivativeReal**
- * *ExpectationValueKetDerivativeImag**
- * *MetricTensorReal* [ComputableMetricTensorReal] (no longer multiplied by factor of 1/4)
- * *MetricTensorImag**
- * *Overlap*
- * *OverlapReal**
- * *OverlapImag**
- * *OverlapSquared*
- *inquanto.computables.composite* (these are composed of atomic computables, and always have “Computable” appended to the end of the name):
 - * *RestrictedOneBodyRDMComputable* [ComputableRestrictedOneBodyRDM]
 - * *UnrestrictedOneBodyRDMComputable* [ComputableUnrestrictedOneBodyRDM]
 - * *RestrictedOneBodyRDMRealComputable* [ComputableRestrictedOneBodyRDMReal]
 - * *UnrestrictedOneBodyRDMRealComputable* [ComputableUnrestrictedOneBodyRDMReal]
 - * *SpinlessNBodyRDMArrayRealComputable* [ComputableSpinlessNBodyRDMTensorReal]
 - * *SpinlessNBodyPDMArrayRealComputable* [ComputableSpinlessNBodyPDMTensorReal]
 - * *RDM1234RealComputable* [ComputableRDM1234Real]
 - * *PDM1234RealComputable* [ComputablePDM1234Real]
 - * *CommutatorComputable* [ComputableCommutator]
 - * *QSEMatricesComputable* [ComputableQSEMatrices]
- The following computables are no longer available:
 - * *ExpectationValueDerivativeReal*
 - * *OverlapSquaredKetDerivative*
- Overhaul of the *inquanto.protocols* module.
 - Protocols now have complete responsibility for building and running observable measurement circuits. They've also been renamed and refactored to be simpler to use. Many are pickle-able for easy pause-resume of experiments.
 - New protocol names [old name]:
 - * *PauliAveraging* [ProtocolDirect]
 - * *HadamardTest* [ProtocolIndirect]
 - * *SwapTest* [ProtocolCSP]
 - * *DestructiveSwapTest* [ProtocolDSP]
 - * *ComputeUncompute* [ProtocolVacuum]

- * *PhaseShift* [ProtocolPhaseShift]
- * *HadamardTestDerivative* [ProtocolHadamardDirectPauliY, ProtocolHadamardIndirectPauliY, ProtocolHadamardDirectPauliZ, ProtocolHadamardIndirectPauliZ]
- * *HadamardTestDerivativeOverlap* [ProtocolHadamardDerivativeOverlap]
- * *SparseStatevectorProtocol* [ProtocolStateVectorSparse]
- * *BackendStatevectorProtocol* [ProtocolStateVectorBackendSupport]
- * *SymbolicProtocol* [ProtocolSymbolic]
- The following protocols are no longer available:
 - * ProtocolMidMeasurementGradient
 - * ProtocolVacuumPhaseShift
- `inquanto.protocols.supporters` submodule is no longer available.
 - Error mitigation is performed using Qermit or with new noise mitigation classes (available from the `inquanto.protocols` module):
 - * *PMSV*
 - * *SPAM*
 - * *CombinedMitigation*
- Changes to the `inquanto.ansatzes` module
 - Renaming of some ansatzes [old name]:
 - * *MultiConfigurationAnsatz* [GivensAnsatz]
 - * *MultiConfigurationState* [MultiReferenceState]
 - `RealBasisRotationAnsatz` is replaced by three new classes, which have improved circuit depths:
 - * *RealGeneralizedBasisRotationAnsatz*
 - * *RealRestrictedBasisRotationAnsatz*
 - * *RealUnrestrictedBasisRotationAnsatz*
- `QubitState/FermionState` have been restructured to make their usage more consistent with `QubitOperator/FermionOperator`
- Many bug fixes

29.28.1 InQuanto-PySCF 1.3.0

- Upgraded to PySCF 2.3.0
- Many more configurable options for drivers
- More memory-efficient DMET calculations with `inquanto-pyscf` integral operators
- Support for starting calculations from checkpoint files
- CAS-AC0 calculations via `pyscf-ac0` <https://github.com/CQCL/pyscf-ac0>
- WFT-in-DFT embedding, for molecular and periodic systems
- Compatibility with InQuanto 3.0

29.29 2.1.1

27 April 2023

- Some performance improvements
- Fixed networkx dependence
- Fixed numpy errors due to _typing
- Other small fixes

A PDF version of InQuanto Version 2 documentation is also available: [InQuanto-v211-PDF](#).

29.30 2.1.0

27 March 2023

- Added double factorization `inquanto.operators.DoubleFactorizedHamiltonian`
- Added Real Basis Rotation Ansatz `inquanto.ansatze.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

29.31 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`.

29.32 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`.

29.33 1.2.2

22 September 2022

- Fixed a typo in auxiliary expressions in ADAPT and VQS.

29.34 1.2.1

21 September 2022

- Added import for `QubitMappingParity` at module level.

29.35 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`.

29.36 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 `ProtocolHadamardDirectPauliY()`)
- 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

29.37 1.0.5

04 July 2022

- Added ability for authentication by password.
- Enabled support for any pytket v1.x release.

29.38 1.0.4

15 June 2022

- Various minor bugfixes.
- Improved validation output.
- Updated sourcedefender dependency version.

29.39 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.

29.40 1.0.2

01 June 2022

- Fixed bug in contracted systems symmetries.
- Added finite difference metric tensor tests.

29.41 1.0.1

25 May 2022

- Fixed bug in contracted systems symmetries.
- Fixed bug in AlgorithmVQE property.
- Fixed time-limit issue on decryption.

CHAPTER
THIRTY

BIBLIOGRAPHY

CHAPTER
THIRTYONE

SUPPORT

Having issues using InQuanto or found a bug? Inquanto-support@quantinuum.com

CHAPTER
THIRTYTWO

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\~noz 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 and Anderson, Robert J and Bhasker, Vardhini},  
    title={{InQuanto: Quantum Computational Chemistry}},  
    url={https://www.quantinuum.com/products-solutions/inquanto},  
    journal={Quantinuum}  
}
```

CHAPTER
THIRTYTHREE

SOFTWARE LICENCE

For enquiries regarding access to InQuanto, please contact inquanto@quantinuum.com.

33.1 Notices

Quantinuum is a trademark name of Quantinuum LLC (or its affiliates) registered in France, Germany, Israel, Japan, Mauritius, Mexico, Russia, United Arab Emirates and Taiwan and unregistered elsewhere.

TKET is a trademark name of Quantinuum Ltd (or its affiliates).

InQuanto is a trademark name of Quantinuum Ltd (or its affiliates) registered in the United Kingdom and European Union and unregistered elsewhere.

CHAPTER
THIRTYFOUR

OPEN-SOURCE ATTRIBUTION

Name	Ver- sion	License	URL	Description
cloud- pickle	3.0.0	BSD License	https://github.com/cloudpipe/cloudpickle	Extended pickling support for Python objects
h5py	3.10.0	BSD License	http://www.h5py.org	Read and write HDF5 files from Python
jax	0.4.23	Apache 2	https://github.com/google/jax	High-performance numerical computing
keyring	24.3.1	MIT License	https://github.com/jaraco/keyring	Store and access your passwords safely.
multipledispatch	1.0.0	BSD	http://github.com/mrocklin/multipledispatch/	Multiple dispatch
nglview	3.0.8	MIT License	https://github.com/arose/nglview	IPython widget to interactively view molecular structures and trajectories.
numpy	1.26.0	BSD License	https://www.numpy.org	NumPy is the fundamental package for array computing with Python.
open- fermion	1.5.1	Apache 2	http://www.openfermion.org	The electronic structure package for quantum computers.
pandas	1.5.3	BSD License	https://pandas.pydata.org	Powerful data structures for data analysis, time series, and statistics
phayes	0.1.1	Apache 2	https://github.com/CQCL/phayes	Bayesian phase and amplitude estimation
pyscf	2.4.0	Apache Software License	http://www.pyscf.org	PySCF: Python-based Simulations of Chemistry Framework
pytket	1.25.0	Apache Software License	https://github.com/CQCL/tket	Python module for interfacing with the CQC tket library of quantum software
pytket- extensions	1.X.X	Apache Software License	https://github.com/CQCL/pytket-extensions	Python module for interfacing quantum backends
qermit	0.5.0	Other/Proprietary License	https://github.com/CQCL/Qermit	error-mitigation framework, an extension to pytket
scipy	1.11.4	BSD License	https://scipy.org/	Fundamental algorithms for scientific computing in Python
sympy	1.12	BSD License	https://sympy.org	Computer algebra system (CAS) in Python
uncer- tainties	3.1.7	BSD License	http://uncertainties-python-package.readthedocs.io/	Transparent calculations with uncertainties
xxhash	3.4.1	BSD License	https://github.com/ifduyue/python-xxhash	Python binding for xxHash
licensing	0.43	MIT License	https://github.com/Cryptolens/cryptolens-python	Python for Cryptolens

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] 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.
- [9] 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.
- [10] 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.
- [11] A Yu Kitaev. Quantum measurements and the abelian stabilizer problem. 1995. arXiv:quant-ph/9511026.
- [12] Richard Cleve, Artur Ekert, Chiara Macchiavello, and Michele Mosca. Quantum algorithms revisited. *Proc. Roy. Soc. Lond. A*, 454:339, 1998. doi:10.1098/rspa.1998.0164.
- [13] Daniel S. Abrams and Seth Lloyd. Quantum algorithm providing exponential speed increase for finding eigenvalues and eigenvectors. *Phys. Rev. Lett.*, 83:5162–5165, Dec 1999. URL: <https://link.aps.org/doi/10.1103/PhysRevLett.83.5162>, doi:10.1103/PhysRevLett.83.5162.

- [14] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, December 2010. ISBN 978-1-139-49548-6.
- [15] 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.
- [16] Ayush Asthana, Ashutosh Kumar, Vibin Abraham, Harper Grimsley, Yu Zhang, Lukasz Cincio, Sergei Tretyak, Pavel A. Dub, Sophia E. Economou, Edwin Barnes, and Nicholas J. Mayhall. Quantum self-consistent equation-of-motion method for computing molecular excitation energies, ionization potentials, and electron affinities on a quantum computer. *Chem. Sci.*, 14:2405–2418, 2023. URL: <http://dx.doi.org/10.1039/D2SC05371C>, doi:10.1039/D2SC05371C.
- [17] Andrew M Childs, Dmitri Maslov, Yunseong Nam, Neil J Ross, and Yuan Su. Toward the first quantum simulation with quantum speedup. *Proc. Natl. Acad. Sci. U. S. A.*, 115(38):9456–9461, September 2018. doi:10.1073/pnas.1801723115.
- [18] Alexei Yu Kitaev, Alexander Shen, and Mikhail N Vyalyi. *Classical and quantum computation*. Number 47. American Mathematical Soc., 2002.
- [19] Krysta M Svore, Matthew B Hastings, and Michael Freedman. Faster phase estimation. *arXiv:1304.0741*, 2013. arXiv:1304.0741.
- [20] Kentaro Yamamoto, Samuel Duffield, Yuta Kikuchi, and David Muñoz Ramo. Demonstrating bayesian quantum phase estimation with quantum error detection. *arXiv:2306.16608*, June 2023. arXiv:2306.16608.
- [21] Miroslav Dobšček, Göran Johansson, Vitaly Shumeiko, and Göran Wendin. Arbitrary accuracy iterative quantum phase estimation algorithm using a single ancillary qubit: a two-qubit benchmark. *Phys. Rev. A*, 76(3):030306, September 2007. doi:10.1103/PhysRevA.76.030306.
- [22] P J J O’Malley, R Babbush, I D Kivlichan, J Romero, J R McClean, R Barends, J Kelly, P Roushan, A Tranter, N Ding, B Campbell, Y Chen, Z Chen, B Chiaro, A Dunsworth, A G Fowler, E Jeffrey, E Lucero, A Megrant, J Y Mutus, M Neeley, C Neill, C Quintana, D Sank, A Vainsencher, J Wenner, T C White, P V Coveney, P J Love, H Neven, A Aspuru-Guzik, and J M Martinis. Scalable quantum simulation of molecular energies. *Phys. Rev. X.*, July 2016. doi:10.1103/physrevx.6.031007.
- [23] Nathan Wiebe and Chris Granade. Efficient bayesian phase estimation. *Phys. Rev. Lett.*, 117:010503, Jun 2016. doi:10.1103/PhysRevLett.117.010503.
- [24] Thomas E O’Brien, Brian Tarasinski, and Barbara M Terhal. Quantum phase estimation of multiple eigenvalues for small-scale (noisy) experiments. *New J. Phys.*, 21(2):023022, feb 2019. doi:10.1088/1367-2630/aafb8e.
- [25] Ewout van den Berg. Efficient Bayesian phase estimation using mixed priors. *Quantum*, 5:469, June 2021. doi:10.22331/q-2021-06-07-469.
- [26] 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.
- [27] 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.
- [28] Adriano Barenco, André Berthiaume, David Deutsch, Artur Ekert, Richard Jozsa, and Chiara Macchiavello. Stabilization of quantum computations by symmetrization. *SIAM Journal on Computing*, 26(5):1541–1557, 1997. URL: <https://doi.org/10.1137/S0097539796302452>.
- [29] Juan Carlos Garcia-Escartin and Pedro Chamorro-Posada. Swap test and hong-ou-mandel effect are equivalent. *Phys. Rev. A*, 87:052330, May 2013. doi:10.1103/PhysRevA.87.052330.
- [30] Maria Schuld, Ville Bergholm, Christian Gogolin, Josh Izaac, and Nathan Killoran. Evaluating analytic gradients on quantum hardware. *Physical Review A*, 99(3):032331, 2019. doi:<https://doi.org/10.1103/PhysRevA.99.032331>.

- [31] William J Huggins, Joonho Lee, Unpil Baek, Bryan O’Gorman, and K Birgitta Whaley. A non-orthogonal variational quantum eigensolver. *New Journal of Physics*, 22(7):073009, jul 2020. doi:10.1088/1367-2630/ab867b.
- [32] Unpil Baek, Diptarka Hait, James Shee, Oskar Leimkuhler, William J. Huggins, Torin F. Stetina, Martin Head-Gordon, and K. Birgitta Whaley. Say no to optimization: a nonorthogonal quantum eigensolver. *PRX Quantum*, 4:030307, Jul 2023. URL: <https://link.aps.org/doi/10.1103/PRXQuantum.4.030307>, doi:10.1103/PRXQuantum.4.030307.
- [33] Gian Giacomo Guerreschi and Mikhail Smelyanskiy. Practical optimization for hybrid quantum-classical algorithms. 2017. arXiv:1701.01450.
- [34] Ying Li and Simon C. Benjamin. Efficient variational quantum simulator incorporating active error minimization. *Phys. Rev. X*, 7:021050, Jun 2017. doi:10.1103/PhysRevX.7.021050.
- [35] Chris N Self, Marcello Benedetti, and David Amaro. Protecting expressive circuits with a quantum error detection code. arXiv:2211.06703, November 2022. arXiv:2211.06703.
- [36] Cristina Cirstoiu, Silas Dilkes, Daniel Mills, Seyond Sivarajah, and Ross Duncan. Volumetric Benchmarking of Error Mitigation with Qermit. *Quantum*, 7:1059, July 2023. URL: <https://doi.org/10.22331/q-2023-07-13-1059>, doi:10.22331/q-2023-07-13-1059.
- [37] Kentaro Yamamoto, David Zsolt Manrique, Irfan T. Khan, Hideaki Sawada, and David Muñoz Ramo. Quantum hardware calculations of periodic systems with partition-measurement symmetry verification: simplified models of hydrogen chain and iron crystals. *Phys. Rev. Res.*, 4:033110, Aug 2022. URL: <https://link.aps.org/doi/10.1103/PhysRevResearch.4.033110>, doi:10.1103/PhysRevResearch.4.033110.
- [38] 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.
- [39] Henrik Koch, Alfredo Sánchez de Merás, and Thomas Bondo Pedersen. Reduced scaling in electronic structure calculations using Cholesky decompositions. *The Journal of Chemical Physics*, 118(21):9481–9484, 06 2003. URL: <https://doi.org/10.1063/1.1578621>, arXiv:https://pubs.aip.org/aip/jcp/article-pdf/118/21/9481/19024657/9481_1_online.pdf, doi:10.1063/1.1578621.
- [40] Evgeny Epifanovsky, Dmitry Zuev, Xintian Feng, Kirill Khistyayev, Yihan Shao, and Anna I. Krylov. General implementation of the resolution-of-the-identity and Cholesky representations of electron repulsion integrals within coupled-cluster and equation-of-motion methods: Theory and benchmarks. *The Journal of Chemical Physics*, 139(13):134105, 10 2013. URL: <https://doi.org/10.1063/1.4820484>, arXiv:https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/1.4820484/15465582/134105_1_online.pdf, doi:10.1063/1.4820484.
- [41] Mario Motta, Erika Ye, Jarrod R McClean, Zhendong Li, Austin J Minnich, Ryan Babbush, and Garnet Kin-Lic Chan. Low rank representations for quantum simulation of electronic structure. *npj Quantum Information*, 7(1):83, 2021.
- [42] David J Thouless. Stability conditions and nuclear rotations in the hartree-fock theory. *Nuclear Physics*, 21:225–232, 1960.
- [43] Ian D Kivlichan, Jarrod McClean, Nathan Wiebe, Craig Gidney, Alán Aspuru-Guzik, Garnet Kin-Lic Chan, and Ryan Babbush. Quantum simulation of electronic structure with linear depth and connectivity. *Physical review letters*, 120(11):110501, 2018.
- [44] Bo Peng and Karol Kowalski. Highly efficient and scalable compound decomposition of two-electron integral tensor and its application in coupled cluster calculations. *Journal of chemical theory and computation*, 13(9):4179–4192, 2017.
- [45] 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.
- [46] 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 Comput-

- ing View on Unitary Coupled Cluster Theory. *arXiv:2109.15176 [physics, physics:quant-ph]*, September 2021. [arXiv:2109.15176](https://arxiv.org/abs/2109.15176).
- [47] I. T. Khan, M. Tudorovskaya, J. J. M. Kirsopp, D. Muñoz Ramo, P. Warrier, D. K. Papanastasiou, and R. Singh. Chemically aware unitary coupled cluster with ab initio calculations on an ion trap quantum computer: A refrigerant chemicals' application. *The Journal of Chemical Physics*, 158(21):214114, 06 2023. URL: <https://doi.org/10.1063/5.0144680>, doi:10.1063/5.0144680.
- [48] 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.
- [49] 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.
- [50] Gian-Luca R Anselmetti, David Wierichs, Christian Gogolin, and Robert M Parrish. Local, expressive, quantum-number-preserving vqe ansätze for fermionic systems. *New Journal of Physics*, 23(11):113010, nov 2021. URL: <https://dx.doi.org/10.1088/1367-2630/ac2cb3>, doi:10.1088/1367-2630/ac2cb3.
- [51] 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.
- [52] 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.
- [53] Philipp Niemann, Rhitam Datta, and Robert Wille. Logic synthesis for quantum state generation. In *2016 IEEE 46th International Symposium on Multiple-Valued Logic (ISMVL)*, volume, 247–252. 2016. doi:10.1109/ISMVL.2016.30.
- [54] 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](https://arxiv.org/abs/1701.08213).
- [55] 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.
- [56] 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.
- [57] James C Spall. An overview of the simultaneous perturbation method for efficient optimization. *Johns Hopkins apl technical digest*, 19(4):482–492, 1998.
- [58] 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.
- [59] 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.
- [60] Sebastian Wouters, Carlos A Jiménez-Hoyos, Qiming Sun, and Garnet K-L Chan. A practical guide to density matrix embedding theory in quantum chemistry. *Journal of chemical theory and computation*, 12(6):2706–2719, 2016.
- [61] M. Krompiec and D. Muñoz Ramo. Strongly contracted n-electron valence state perturbation theory using reduced density matrices from a quantum computer. *arXiv preprint*, pages 2210.05702, 2022. [arXiv:2210.05702](https://arxiv.org/abs/2210.05702), doi:10.48550/arXiv.2210.05702.
- [62] Yang Guo, Kantharuban Sivalingam, and Frank Neese. Approximations of density matrices in n-electron valence state second-order perturbation theory (nevpT2). i. revisiting the nevpT2 construction. *J. Chem. Phys.*, 2021.
- [63] Ewa Pastoreczak and Katarzyna Pernal. Correlation energy from the adiabatic connection formalism for complete active space wave functions. *J. Chem. Theory Comput.*, 14(7):3493–3503, 2018.

- [64] 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](https://doi.org/10.1002/anie.200802019).
- [65] Lili Cao and Ulf Ryde. On the Difference Between Additive and Subtractive QM/MM Calculations. *Frontiers in Chemistry*, 2018.
- [66] 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](https://doi.org/10.1039/P29930000799).
- [67] 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](https://doi.org/10.1063/1.4901304).

PYTHON MODULE INDEX

i

inquanto.computables.atomic, 462
inquanto.computables.composite, 532
inquanto.computables.primitive, 485
inquanto.core, 584
inquanto.embeddings.dmet, 585
inquanto.express, 600
inquanto.extensions.nglview, 1264
inquanto.extensions.phayes, 1266
inquanto.extensions.pyscf, 1104
inquanto.extensions.pyscf.fmo, 1258
inquanto.minimizers, 644
inquanto.operators, 652
inquanto.spaces, 1017
inquanto.states, 1055
inquanto.symmetry, 1099

INDEX

Non-alphabetical

`_init_()` (*QCM4Computable method*), 553
`_MAPPING_FLAGS` (*QubitMapping attribute*), 626
`_MAPPING_FLAGS` (*QubitMappingBravyiKitaev attribute*), 633
`_MAPPING_FLAGS` (*QubitMappingJordanWigner attribute*), 630
`_MAPPING_FLAGS` (*QubitMappingParaparticular attribute*), 640
`_MAPPING_FLAGS` (*QubitMappingParity attribute*), 637

A

`add()` (*SymbolSet method*), 575
`add_atom()` (*GeometryMolecular method*), 604
`add_atom()` (*GeometryPeriodic method*), 615
`add_label()` (*CommutatorComputable method*), 532
`add_label()` (*ComputableFunction method*), 485
`add_label()` (*ComputableInt method*), 487
`add_label()` (*ComputableList method*), 489
`add_label()` (*ComputableNDArray method*), 492
`add_label()` (*ComputableNode method*), 527
`add_label()` (*ComputableSingleChild method*), 529
`add_label()` (*ExpectationValue method*), 462
`add_label()` (*ExpectationValueBraDerivativeImag method*), 464
`add_label()` (*ExpectationValueBraDerivativeReal method*), 466
`add_label()` (*ExpectationValueDerivative method*), 468
`add_label()` (*ExpectationValueKetDerivativeImag method*), 469
`add_label()` (*ExpectationValueKetDerivativeReal method*), 471
`add_label()` (*ExpectationValueNonHermitian method*), 473
`add_label()` (*ExpectationValueSumComputable method*), 534
`add_label()` (*HoleGFComputable method*), 535
`add_label()` (*KrylovSubspaceComputable method*), 539
`add_label()` (*LanczosCoefficientsComputable method*), 541
`add_label()` (*LanczosMatrixComputable method*), 542
`add_label()` (*ManyBodyGFComputable method*), 544

`add_label()` (*MetricTensorImag method*), 475
`add_label()` (*MetricTensorReal method*), 476
`add_label()` (*NonOrthogonalMatricesComputable method*), 546
`add_label()` (*Overlap method*), 478
`add_label()` (*OverlapImag method*), 480
`add_label()` (*OverlapMatrixComputable method*), 548
`add_label()` (*OverlapReal method*), 481
`add_label()` (*OverlapSquared method*), 483
`add_label()` (*ParticleGFComputable method*), 552
`add_label()` (*PDM1234RealComputable method*), 550
`add_label()` (*QCM4Computable method*), 554
`add_label()` (*QSEMatricesComputable method*), 555
`add_label()` (*RDM1234RealComputable method*), 557
`add_label()` (*RestrictedOneBodyRDMComputable method*), 559
`add_label()` (*RestrictedOneBodyRDMRealComputable method*), 561
`add_label()` (*SCEOMMatrixComputable method*), 562
`add_label()` (*SpinlessNBodyPDMArrayRealComputable method*), 565
`add_label()` (*SpinlessNBodyRDMArrayRealComputable method*), 567
`add_label()` (*UnrestrictedOneBodyRDMComputable method*), 569
`add_label()` (*UnrestrictedOneBodyRDMRealComputable method*), 570
`AlgorithmAdaptVQE` (class in *in quanto.algorithms.adapt*), 314
`AlgorithmBayesianQPE` (class in *in quanto.extensions.phayes*), 1266
`AlgorithmDeterministicQPE` (class in *in quanto.algorithms.phase_estimation*), 321
`AlgorithmFermionicAdaptVQE` (class in *in quanto.algorithms.adapt*), 313
`AlgorithmInfoTheoryQPE` (class in *in quanto.algorithms.phase_estimation*), 323
`AlgorithmIQEB` (class in *in quanto.algorithms.adapt*), 315
`AlgorithmKitaevQPE` (class in *in quanto.algorithms.phase_estimation*), 324
`AlgorithmMcLachlanImagTime` (class in *in quanto.algorithms.time_evolution*), 328

AlgorithmMcLachlanRealTime (class in *in quanto.algorithms.time_evolution*), 326
 AlgorithmQSE (class in *inquanto.algorithms.qse*), 319
 AlgorithmSCEOM (class in *inquanto.algorithms.sceom*), 320
 AlgorithmVQD (class in *inquanto.algorithms.vqd*), 318
 AlgorithmVQE (class in *inquanto.algorithms.vqe*), 317
 AlgorithmVQS (class in *in quanto.algorithms.time_evolution*), 325
 align_bond_to_axis() (*GeometryMolecular* method), 604
 align_bond_to_axis() (*GeometryPeriodic* method), 615
 align_bond_to_vector() (*GeometryMolecular* method), 604
 align_bond_to_vector() (*GeometryPeriodic* method), 616
 align_to_plane() (*GeometryMolecular* method), 604
 align_to_plane() (*GeometryPeriodic* method), 616
 align_to_xy_plane() (*GeometryMolecular* method), 605
 align_to_xy_plane() (*GeometryPeriodic* method), 616
 align_to_xz_plane() (*GeometryMolecular* method), 605
 align_to_xz_plane() (*GeometryPeriodic* method), 616
 align_to_yz_plane() (*GeometryMolecular* method), 605
 align_to_yz_plane() (*GeometryPeriodic* method), 616
 ALIGNED (*ComputableNDArray* attribute), 503
 ALL (*FermionOperatorList.CompressScalarsBehavior* attribute), 702
 ALL (*QubitOperatorList.CompressScalarsBehavior* attribute), 757
 ALL (*SymmetryOperatorFermionicFactorised.CompressScalarsBehavior* attribute), 821
 ALL (*SymmetryOperatorPauliFactorised.CompressScalarsBehavior* attribute), 870
 all() (*ComputableNDArray* method), 493
 all_modes (*FermionStateString* property), 1067
 all_modes (*QubitStateString* property), 1083
 all_modes (*StateString* property), 1096
 all_nontrivial_qubits (*QubitOperator* property), 735
 all_nontrivial_qubits (*QubitOperatorList* property), 771
 all_nontrivial_qubits (*QubitOperatorString* property), 788
 all_nontrivial_qubits (*SymmetryOperatorPauli* property), 848
 all_nontrivial_qubits (*SymmetryOperatorPauliFactorised* property), 884
 all_qubits (*QubitOperator* property), 735
 all_qubits (*QubitOperatorList* property), 771
 all_qubits (*SymmetryOperatorPauli* property), 848
 all_qubits (*SymmetryOperatorPauliFactorised* property), 884
 alpha_f() (*KrylovSubspace* method), 537
 ansatz_parameters_from_unitary() (*RealGeneralizedBasisRotationAnsatz* method), 428
 ansatz_parameters_from_unitary() (*RealRestrictedBasisRotationAnsatz* method), 434
 ansatz_parameters_from_unitary() (*RealUnrestrictedBasisRotationAnsatz* method), 439
 anticommutator() (*QubitOperator* method), 735
 anticommutator() (*QubitOperatorString* method), 788
 anticommutator() (*SymmetryOperatorPauli* method), 848
 anticommutes_with() (*QubitOperator* method), 735
 anticommutes_with() (*QubitOperatorString* method), 788
 anticommutes_with() (*SymmetryOperatorPauli* method), 848
 antihermitian_part() (*QubitOperator* method), 736
 antihermitian_part() (*SymmetryOperatorPauli* method), 848
 any() (*ComputableNDArray* method), 493
 ao_mask_2_atom_mask() (*FMO* static method), 1258
 append() (*ComputableList* method), 489
 append() (*ProtocolList* method), 1005
 apply_bra() (*FermionOperator* method), 684
 apply_bra() (*FermionOperatorString* method), 723
 apply_bra() (*SymmetryOperatorFermionic* method), 803
 apply_ket() (*FermionOperator* method), 684
 apply_ket() (*FermionOperatorString* method), 724
 apply_ket() (*SymmetryOperatorFermionic* method), 803
 apply_state() (*FermionOperatorString* method), 724
 approx_equal() (*ChemistryRestrictedIntegralOperator* method), 652
 approx_equal() (*ChemistryRestrictedIntegralOperatorCompact* method), 658
 approx_equal() (*ChemistryUnrestrictedIntegralOperator* method), 663
 approx_equal() (*ChemistryUnrestrictedIntegralOperatorCompact* method), 668
 approx_equal() (*PySCFChemistryRestrictedIntegralOperator* method), 1249
 approx_equal() (*PySCFChemistryUnrestrictedIntegralOperator* method), 1253
 approx_equal_to() (*FermionOperator* method), 684
 approx_equal_to() (*FermionState* method), 1056
 approx_equal_to() (*QubitOperator* method), 736
 approx_equal_to() (*QubitState* method), 1070
 approx_equal_to() (*State* method), 1086
 approx_equal_to() (*SymmetryOperatorFermionic* method), 803
 approx_equal_to() (*SymmetryOperatorPauli* method),

849
approx_equal_to_by_random_subs() (*FermionOperator method*), 685
approx_equal_to_by_random_subs() (*FermionState method*), 1056
approx_equal_to_by_random_subs() (*QubitOperator method*), 736
approx_equal_to_by_random_subs() (*QubitState method*), 1071
approx_equal_to_by_random_subs() (*State method*), 1086
approx_equal_to_by_random_subs() (*Symmetry OperatorFermionic method*), 804
approx_equal_to_by_random_subs() (*Symmetry OperatorPauli method*), 849
argmax() (*ComputableNDArray method*), 493
argmin() (*ComputableNDArray method*), 493
argpartition() (*ComputableNDArray method*), 493
args (*ComputableFunction attribute*), 486
argsort() (*ComputableNDArray method*), 494
as_integer_ratio() (*CacheLevels method*), 580
as_integer_ratio() (*CacheSizeUnit method*), 581
as_integer_ratio() (*CompilationLevel method*), 1014
as_integer_ratio() (*CtrluStrat method*), 1016
as_scalar() (*FermionOperator method*), 685
as_scalar() (*QubitOperator method*), 736
as_scalar() (*SymmetryOperatorFermionic method*), 804
as_scalar() (*SymmetryOperatorPauli method*), 849
astype() (*ChemistryRestrictedIntegralOperator method*), 653
astype() (*ChemistryRestrictedIntegralOperatorCompact method*), 658
astype() (*CompactTwoBodyIntegralsS4 method*), 672
astype() (*CompactTwoBodyIntegralsS8 method*), 674
astype() (*ComputableNDArray method*), 494
atom_mask_2_ao_mask() (*FMO static method*), 1259
atomic_coordinates (*GeometryMolecular property*), 605
atomic_coordinates (*GeometryPeriodic property*), 617
AVAS (*class in inquanto.extensions.pyscf*), 1104

B

backend (*IterativePhaseEstimation property*), 987
backend (*IterativePhaseEstimationQuantinuum property*), 992
BackendStatevectorProtocol (*class in inquanto.protocols*), 964
base (*ComputableNDArray attribute*), 495
base (*InQuantoContext property*), 584
basis_states (*FermionState property*), 1056
basis_states (*QubitState property*), 1071
basis_states (*State property*), 1086
BEHAVED (*ComputableNDArray attribute*), 503
beta_f() (*KrylovSubspace method*), 537
beta_iqpe (*IterativePhaseEstimation property*), 988
beta_iqpe (*IterativePhaseEstimationQuantinuum property*), 992
bit_count() (*CacheLevels method*), 580
bit_count() (*CacheSizeUnit method*), 581
bit_count() (*CompilationLevel method*), 1014
bit_count() (*CtrluStrat method*), 1016
bit_length() (*CacheLevels method*), 580
bit_length() (*CacheSizeUnit method*), 582
bit_length() (*CompilationLevel method*), 1014
bit_length() (*CtrluStrat method*), 1016
block_counter (*TimerWith attribute*), 583
bond_angle() (*GeometryMolecular method*), 605
bond_angle() (*GeometryPeriodic method*), 617
bond_length() (*GeometryMolecular method*), 606
bond_length() (*GeometryPeriodic method*), 617
bra_state (*Overlap attribute*), 478
bra_state (*OverlapImag attribute*), 480
bra_state (*OverlapReal attribute*), 482
bra_state (*OverlapSquared attribute*), 484
BRING_INTO_OPERATOR (*QubitOperatorList.ExpandExponentialProductCoefficientsBehavior attribute*), 762
BRING_INTO_OPERATOR (*SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior attribute*), 874
build() (*AlgorithmAdaptVQE method*), 315
build() (*AlgorithmBayesianQPE method*), 1266
build() (*AlgorithmDeterministicQPE method*), 322
build() (*AlgorithmFermionicAdaptVQE method*), 313
build() (*AlgorithmInfoTheoryQPE method*), 324
build() (*AlgorithmIQEB method*), 316
build() (*AlgorithmKitaevQPE method*), 325
build() (*AlgorithmMcLachlanImagTime method*), 328
build() (*AlgorithmMcLachlanRealTime method*), 327
build() (*AlgorithmQSE method*), 319
build() (*AlgorithmSCEOM method*), 320
build() (*AlgorithmVQD method*), 318
build() (*AlgorithmVQE method*), 317
build() (*AlgorithmVQS method*), 326
build() (*ComputeUncompute method*), 919
build() (*ComputeUncomputeFactorizedOverlap method*), 956
build() (*DestructiveSwapTest method*), 925
build() (*FactorizedOverlap method*), 944
build() (*HadamardTest method*), 912
build() (*HadamardTestDerivative method*), 973
build() (*HadamardTestDerivativeOverlap method*), 967
build() (*HadamardTestOverlap method*), 938
build() (*IterativePhaseEstimation method*), 988
build() (*IterativePhaseEstimationQuantinuum method*), 992
build() (*IterativePhaseEstimationStatevector method*), 996

build() (*PauliAveraging method*), 905
 build() (*PhaseShift method*), 980
 build() (*ProjectiveMeasurements method*), 999
 build() (*SwapFactorizedOverlap method*), 950
 build() (*SwapTest method*), 931
 build_2atom_chain() (*GeometryMolecular method*), 606
 build_2atom_chain() (*GeometryPeriodic method*), 617
 build_alternating_ring() (*GeometryMolecular method*), 606
 build_alternating_ring() (*GeometryPeriodic method*), 617
 build_from() (*ComputeUncompute method*), 919
 build_from() (*ComputeUncomputeFactorizedOverlap method*), 957
 build_from() (*DestructiveSwapTest method*), 926
 build_from() (*FactorizedOverlap method*), 944
 build_from() (*HadamardTest method*), 913
 build_from() (*HadamardTestDerivative method*), 974
 build_from() (*HadamardTestDerivativeOverlap method*), 967
 build_from() (*HadamardTestOverlap method*), 938
 build_from() (*PauliAveraging method*), 906
 build_from() (*PhaseShift method*), 980
 build_from() (*SwapFactorizedOverlap method*), 951
 build_from() (*SwapTest method*), 932
 build_from_circuit() (*IterativePhaseEstimation method*), 988
 build_from_circuit() (*IterativePhaseEstimation-Quantinuum method*), 992
 build_from_circuit() (*IterativePhaseEstimationStatevector method*), 997
 build_mm_charges() (*ChemistryDriverPySCFMolecularRHFQMMMCOSMO static method*), 1188
 build_mm_charges() (*ChemistryDriverPySCFMolecularROHFQMMMCOSMO static method*), 1206
 build_mm_charges() (*ChemistryDriverPySCFMolecularUHFQMMMCOSMO static method*), 1225
 build_protocols_from() (*ComputeUncompute class method*), 920
 build_protocols_from() (*ComputeUncomputeFactorizedOverlap class method*), 957
 build_protocols_from() (*DestructiveSwapTest class method*), 926
 build_protocols_from() (*FactorizedOverlap class method*), 945
 build_protocols_from() (*HadamardTest class method*), 913
 build_protocols_from() (*HadamardTestDerivative class method*), 974
 build_protocols_from() (*HadamardTestDerivativeOverlap class method*), 968
 build_protocols_from() (*HadamardTestOverlap class method*), 938
 build_protocols_from() (*PauliAveraging class method*), 906
 build_protocols_from() (*PhaseShift class method*), 981
 build_protocols_from() (*SwapFactorizedOverlap class method*), 951
 build_protocols_from() (*SwapTest class method*), 932
 build_rectangle() (*GeometryMolecular method*), 606
 build_rectangle() (*GeometryPeriodic method*), 618
 build_ring() (*GeometryMolecular method*), 606
 build_ring() (*GeometryPeriodic method*), 618
 build_subset() (*QubitOperatorList method*), 771
 build_subset() (*SymmetryOperatorPauliFactorised method*), 884
 build_supercell() (*GeometryPeriodic method*), 618
 BYTES (*CacheSizeUnit attribute*), 581
 byteswap() (*ComputableNDArray method*), 495

C

C_CONTIGUOUS (*ComputableNDArray attribute*), 503
 ca() (*FermionOperator class method*), 685
 ca() (*SymmetryOperatorFermionic class method*), 804
 caca() (*FermionOperator class method*), 686
 caca() (*SymmetryOperatorFermionic class method*), 804
 cache (*Cache property*), 578
 Cache (*class in inquanto.core*), 578
 cache (*ProtocolCache property*), 1011
 cache (*SparseStatevectorProtocol property*), 962
 cache_hit_report() (*SparseStatevectorProtocol method*), 962
 CacheLevels (*class in inquanto.core*), 579
 CacheSizeUnit (*class in inquanto.core*), 581
 calibrate() (*SPAM method*), 1009
 calibrate_process() (*SPAM method*), 1009
 calibrate_retrieve() (*SPAM method*), 1010
 CanonicalPhaseEstimation (*class in inquanto.protocols*), 986
 capitalize() (*FermionOperatorList.CompressScalarsBehavior method*), 703
 capitalize() (*FermionOperatorList.FactoryCoefficientsLocation method*), 707
 capitalize() (*FermionOperator.TrotterizeCoefficientsLocation method*), 680
 capitalize() (*QubitOperatorList.CompressScalarsBehavior method*), 757
 capitalize() (*QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method*), 762

capitalize()	(QubitOperatorFactoryCoefficientsLocation method),	826
767		
capitalize()	(QubitOperatorTrotterizeCoefficientsLocation method),	798
730		
capitalize()	(SymmetryOperatorFermionicFactoryCoefficientsLocation method),	821
821		
capitalize()	(SymmetryOperatorFermionicFactoryCoefficientsLocation method),	826
826		
capitalize()	(SymmetryOperatorFermionicFactoryCoefficientsLocation method),	798
798		
capitalize()	(SymmetryOperatorPauliFactoredCompressScalarsBehavior method),	870
870		
capitalize()	(SymmetryOperatorPauliFactoredExpandExponentialProductCoefficientsBehavior method),	875
875		
capitalize()	(SymmetryOperatorPauliFactoredFactoryCoefficientsLocation method),	880
880		
capitalize()	(SymmetryOperatorPauliFactoredTrotterizeCoefficientsLocation method),	843
843		
CARRAY (ComputableNDArray attribute),	503	
casefold()	(FermionOperatorList.CompressScalarsBehavior method),	703
703		
casefold()	(FermionOperatorList.FactoryCoefficientsLocation method),	707
707		
casefold()	(FermionOperatorList.TrotterizeCoefficientsLocation method),	680
680		
casefold()	(QubitOperatorList.CompressScalarsBehavior method),	757
757		
casefold()	(QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method),	762
762		
casefold()	(QubitOperatorList.FactoryCoefficientsLocation method),	767
767		
casefold()	(QubitOperatorList.TrotterizeCoefficientsLocation method),	730
730		
casefold()	(SymmetryOperatorFermionicFactoryCoefficientsLocation method),	822
822		
casefold()	(SymmetryOperatorFermionicFactoryCoefficientsLocation method),	826
826		
casefold()	(SymmetryOperatorFermionicTrotterizeCoefficientsLocation method),	798
798		
casefold()	(SymmetryOperatorPauliFactoredCompressScalarsBehavior method),	870
870		
casefold()	(SymmetryOperatorPauliFactoredExpandExponentialProductCoefficientsBehavior method),	875
875		
casefold()	(SymmetryOperatorPauliFactoredFactoryCoefficientsLocation method),	880
880		

880
 center() (SymmetryOperator-
PauliTrotterizeCoefficientsLocation method),
 843
 chain_filters() (*in module inquanto.spaces*), 1055
 check_energies() (*SCEOMMatrixComputable method*),
 563
 check_mem (*Cache property*), 578
 check_mem (*ProtocolCache property*), 1012
 check_s4_symmetry() (*CompactTwoBodyIntegralsS4 static method*), 673
 check_s8_symmetry() (*CompactTwoBodyIntegralsS8 static method*), 674
 check_translation_invariance() (*FermionSpace-Supercell method*), 1041
 ChemistryDriverPySCFEmbeddingGammaRHF (*class in inquanto.extensions.pyscf*), 1107
 ChemistryDriverPySCFEmbeddingGammaROHF_UHF (*class in inquanto.extensions.pyscf*), 1116
 ChemistryDriverPySCFEmbeddingRHF (*class in inquanto.extensions.pyscf*), 1126
 ChemistryDriverPySCFEmbeddingROHF (*class in inquanto.extensions.pyscf*), 1135
 ChemistryDriverPySCFEmbeddingROHF_UHF (*class in inquanto.extensions.pyscf*), 1144
 ChemistryDriverPySCFGammaRHF (*class in inquanto.extensions.pyscf*), 1153
 ChemistryDriverPySCFGammaROHF (*class in inquanto.extensions.pyscf*), 1162
 ChemistryDriverPySCFIntegrals (*class in inquanto.extensions.pyscf*), 1171
 ChemistryDriverPySCFMolecularRHF (*class in inquanto.extensions.pyscf*), 1178
 ChemistryDriverPySCFMolecularRHFQMMMCOSMO (*class in inquanto.extensions.pyscf*), 1187
 ChemistryDriverPySCFMolecularROHF (*class in inquanto.extensions.pyscf*), 1196
 ChemistryDriverPySCFMolecularROHFQMMMCOSMO (*class in inquanto.extensions.pyscf*), 1205
 ChemistryDriverPySCFMolecularUHF (*class in inquanto.extensions.pyscf*), 1215
 ChemistryDriverPySCFMolecularUHFQMMMCOSMO (*class in inquanto.extensions.pyscf*), 1224
 ChemistryDriverPySCFMomentumRHF (*class in inquanto.extensions.pyscf*), 1233
 ChemistryDriverPySCFMomentumROHF (*class in inquanto.extensions.pyscf*), 1236
 ChemistryRestrictedIntegralOperator (*class in inquanto.operators*), 652
 ChemistryRestrictedIntegralOperatorCompact (*class in inquanto.operators*), 658
 ChemistryUnrestrictedIntegralOperator (*class in inquanto.operators*), 663
 ChemistryUnrestrictedIntegralOperatorCompact
 (*class in inquanto.operators*), 667
 children() (*CommutatorComputable method*), 532
 children() (*ComputableFunction method*), 486
 children() (*ComputableInt method*), 487
 children() (*ComputableList method*), 489
 children() (*ComputableNDArray method*), 496
 children() (*ComputableNode method*), 528
 children() (*ComputableSingleChild method*), 529
 children() (*ComputableTuple method*), 531
 children() (*ExpectationValue method*), 463
 children() (*ExpectationValueBraDerivativeImag method*), 464
 children() (*ExpectationValueBraDerivativeReal method*), 466
 children() (*ExpectationValueDerivative method*), 468
 children() (*ExpectationValueKetDerivativeImag method*), 470
 children() (*ExpectationValueKetDerivativeReal method*), 471
 children() (*ExpectationValueNonHermitian method*), 473
 children() (*ExpectationValueSumComputable method*), 534
 children() (*HoleGFComputable method*), 536
 children() (*KrylovSubspaceComputable method*), 539
 children() (*LanczosCoefficientsComputable method*), 541
 children() (*LanczosMatrixComputable method*), 543
 children() (*ManyBodyGFComputable method*), 545
 children() (*MetricTensorImag method*), 475
 children() (*MetricTensorReal method*), 476
 children() (*NonOrthogonalMatricesComputable method*), 547
 children() (*Overlap method*), 478
 children() (*OverlapImag method*), 480
 children() (*OverlapMatrixComputable method*), 548
 children() (*OverlapReal method*), 482
 children() (*OverlapSquared method*), 484
 children() (*ParticleGFComputable method*), 552
 children() (*PDM1234RealComputable method*), 550
 children() (*QCM4Computable method*), 554
 children() (*QSEMatricesComputable method*), 556
 children() (*RDM1234RealComputable method*), 558
 children() (*RestrictedOneBodyRDMComputable method*), 559
 children() (*RestrictedOneBodyRDMRealComputable method*), 561
 children() (*SCEOMMatrixComputable method*), 563
 children() (*SpinlessNBodyPDMArrayRealComputable method*), 565
 children() (*SpinlessNBodyRDMArrayRealComputable method*), 567
 children() (*UnrestrictedOneBodyRDMComputable method*), 569

children() (*UnrestrictedOneBodyRDMRealComputable method*), 571
CHOLESKY (*DecompositionMethod attribute*), 676
choose() (*ComputableNDArray method*), 496
circuit (*ComputeUncompute attribute*), 920
circuit (*DestructiveSwapTest attribute*), 926
circuit (*ProjectiveMeasurements attribute*), 1000
circuit (*SwapTest attribute*), 933
CircuitAnsatz (*class in inquanto.ansatzes*), 334
CircuitEncoderQuantinuum (*class in quanto.protocols*), 1013
circuits (*ComputeUncomputeFactorizedOverlap attribute*), 957
circuits (*FactorizedOverlap attribute*), 945
circuits (*HadamardTestOverlap attribute*), 939
circuits (*SwapFactorizedOverlap attribute*), 952
clear() (*Cache method*), 579
clear() (*ComputableList method*), 489
clear() (*ComputeUncompute method*), 920
clear() (*ComputeUncomputeFactorizedOverlap method*), 958
clear() (*DestructiveSwapTest method*), 926
clear() (*FactorizedOverlap method*), 945
clear() (*HadamardTest method*), 914
clear() (*HadamardTestDerivative method*), 975
clear() (*HadamardTestDerivativeOverlap method*), 968
clear() (*HadamardTestOverlap method*), 939
clear() (*IterativePhaseEstimation method*), 988
clear() (*IterativePhaseEstimationQuantinuum method*), 993
clear() (*IterativePhaseEstimationStatevector method*), 997
clear() (*PauliAveraging method*), 907
clear() (*PhaseShift method*), 981
clear() (*ProjectiveMeasurements method*), 1000
clear() (*ProtocolCache method*), 1012
clear() (*SparseStatevectorProtocol method*), 963
clear() (*SwapFactorizedOverlap method*), 952
clear() (*SwapTest method*), 933
clear() (*SymbolDict method*), 572
clear() (*SymbolicProtocol method*), 965
clear() (*SymbolSet method*), 575
clear_cache() (*IterativePhaseEstimation method*), 988
clear_cache() (*IterativePhaseEstimationQuantinuum method*), 993
clip() (*ComputableNDArray method*), 496
clone() (*CircuitAnsatz method*), 334
clone() (*ComposedAnsatz method*), 340
clone() (*FermionOperator method*), 686
clone() (*FermionOperatorList method*), 712
clone() (*FermionSpaceAnsatzChemicallyAwareUCCSD method*), 373
clone() (*FermionSpaceAnsatzkUpCCGD method*), 379
clone() (*FermionSpaceAnsatzkUpCCGSD method*), 384
clone() (*FermionSpaceAnsatzkUpCCGSDSinglet method*), 390
clone() (*FermionSpaceAnsatzUCCD method*), 363
clone() (*FermionSpaceAnsatzUCCGD method*), 395
clone() (*FermionSpaceAnsatzUCCGSD method*), 401
clone() (*FermionSpaceAnsatzUCCSD method*), 357
clone() (*FermionSpaceAnsatzUCCSDSinglet method*), 407
clone() (*FermionSpaceStateExp method*), 352
clone() (*FermionSpaceStateExpChemicallyAware method*), 368
clone() (*FermionState method*), 1056
clone() (*GeneralAnsatz method*), 329
clone() (*HamiltonianVariationalAnsatz method*), 446
clone() (*HardwareEfficientAnsatz method*), 457
clone() (*LayeredAnsatz method*), 452
clone() (*MultiConfigurationAnsatz method*), 412
clone() (*MultiConfigurationState method*), 418
clone() (*MultiConfigurationStateBox method*), 423
clone() (*QubitOperator method*), 737
clone() (*QubitOperatorList method*), 772
clone() (*QubitState method*), 1071
clone() (*RealGeneralizedBasisRotationAnsatz method*), 428
clone() (*RealRestrictedBasisRotationAnsatz method*), 434
clone() (*RealUnrestrictedBasisRotationAnsatz method*), 439
clone() (*State method*), 1086
clone() (*SymmetryOperatorFermionic method*), 805
clone() (*SymmetryOperatorFermionicFactorised method*), 831
clone() (*SymmetryOperatorPauli method*), 849
clone() (*SymmetryOperatorPauliFactorised method*), 884
clone() (*TrotterAnsatz method*), 345
coefficients (*FermionOperator property*), 687
coefficients (*FermionState property*), 1057
coefficients (*QubitOperator property*), 737
coefficients (*QubitState property*), 1071
coefficients (*State property*), 1086
coefficients (*SymmetryOperatorFermionic property*), 805
coefficients (*SymmetryOperatorPauli property*), 849
collapse_as_linear_combination() (*FermionOperatorList method*), 712
collapse_as_linear_combination() (*QubitOperatorList method*), 772
collapse_as_linear_combination() (*SymmetryOperatorFermionicFactorised method*), 831
collapse_as_linear_combination() (*SymmetryOperatorPauliFactorised method*), 885
collapse_as_product() (*FermionOperatorList method*), 712
collapse_as_product() (*QubitOperatorList method*), 772

collapse_as_product() (*SymmetryOperatorFermion-icFactorised method*), 831
collapse_as_product() (*SymmetryOperatorPauliFactorised method*), 885
COLUMN_KP (*FermionSpaceBrillouin attribute*), 1034
COLUMN_ORB (*FermionSpace attribute*), 1017
COLUMN_ORB (*FermionSpaceBrillouin attribute*), 1034
COLUMN_ORB (*FermionSpaceSupercell attribute*), 1041
COLUMN_RP (*FermionSpaceSupercell attribute*), 1041
COLUMN_SPIN (*FermionSpace attribute*), 1017
COLUMN_SPIN (*FermionSpaceBrillouin attribute*), 1034
COLUMN_SPIN (*FermionSpaceSupercell attribute*), 1041
CombinedMitigation (*class in inquanto.protocols*), 1010
commutator() (*FermionOperator method*), 687
commutator() (*QubitOperator method*), 737
commutator() (*QubitOperatorString method*), 789
commutator() (*SymmetryOperatorFermionic method*), 805
commutator() (*SymmetryOperatorPauli method*), 849
CommutatorComputable (*class in inquanto.computables.composite*), 532
commutes_with() (*FermionOperator method*), 687
commutes_with() (*QubitOperator method*), 737
commutes_with() (*QubitOperatorString method*), 789
commutes_with() (*SymmetryOperatorFermionic method*), 805
commutes_with() (*SymmetryOperatorPauli method*), 850
CompactTwoBodyIntegralsS4 (*class in inquanto.operators*), 672
CompactTwoBodyIntegralsS8 (*class in inquanto.operators*), 674
compatibility_matrix() (*QubitOperatorList method*), 772
compatibility_matrix() (*SymmetryOperatorPauliFactorised method*), 885
compilation_level (*IterativePhaseEstimation property*), 988
compilation_level (*IterativePhaseEstimationQuantum property*), 993
CompilationLevel (*class in inquanto.protocols*), 1014
compile_circuits() (*ProjectiveMeasurements method*), 1000
COMPILED (*CompilationLevel attribute*), 1014
compose_fragments() (*FMOFragment class method*), 1259
compose_fragments() (*FMOFragmentPySCFActive class method*), 1260
compose_fragments() (*FMOFragmentPySCFCCSD class method*), 1262
compose_fragments() (*FMOFragmentPySCFMP2 class method*), 1263
compose_fragments() (*FMOFragmentPySCFRHF class method*), 1263
ComposedAnsatz (*class in inquanto.ansatzes*), 339
compress() (*ComputableNDArray method*), 497
compress() (*FermionOperator method*), 687
compress() (*FermionState method*), 1057
compress() (*QubitOperator method*), 737
compress() (*QubitOperatorString method*), 789
compress() (*QubitState method*), 1071
compress() (*State method*), 1086
compress() (*SymmetryOperatorFermionic method*), 806
compress() (*SymmetryOperatorPauli method*), 850
compress_scalars_as_product() (*FermionOperatorList method*), 713
compress_scalars_as_product() (*QubitOperatorList method*), 773
compress_scalars_as_product() (*SymmetryOperatorFermionicFactorised method*), 832
compress_scalars_as_product() (*SymmetryOperatorPauliFactorised method*), 886
ComputableFunction (*class in inquanto.computables.primitive*), 485
ComputableInt (*class in inquanto.computables.primitive*), 487
ComputableList (*class in inquanto.computables.primitive*), 488
ComputableNDArray (*class in inquanto.computables.primitive*), 491
ComputableNode (*class in inquanto.computables.primitive*), 527
ComputableSingleChild (*class in inquanto.computables.primitive*), 529
ComputableTuple (*class in inquanto.computables.primitive*), 530
compute_distance_matrix() (*GeometryMolecular method*), 607
compute_distance_matrix() (*GeometryPeriodic method*), 618
compute_fragment_energy() (*DMETRHFFragment static method*), 589
compute_fragment_energy() (*DMETRHFFragment-PySCFCCSD static method*), 1241
compute_fragment_energy() (*DMETRHFFragment-PySCFFCI static method*), 1242
compute_fragment_energy() (*DMETRHFFragment-PySCFMP2 static method*), 1242
compute_fragment_energy() (*DMETRHFFragment-PySCFRHF static method*), 1243
compute_jacobian() (*QubitOperatorList method*), 774
compute_jacobian() (*SymmetryOperatorPauliFactorised method*), 887
compute_nuclear_dipole() (*ChemistryDriver-PySCFEmbeddingGammaRHF method*), 1108
compute_nuclear_dipole() (*ChemistryDriver-PySCFEmbeddingGammaROHF_UHF method*), 1108

1118
`compute_nuclear_dipole()` (*ChemistryDriver-PySCFEmbeddingRHF method*), 1127
`compute_nuclear_dipole()` (*ChemistryDriver-PySCFEmbeddingROHF method*), 1135
`compute_nuclear_dipole()` (*ChemistryDriver-PySCFEmbeddingROHF_UHF method*), 1145
`compute_nuclear_dipole()` (*ChemistryDriver-PySCFGammaRHF method*), 1153
`compute_nuclear_dipole()` (*ChemistryDriver-PySCFGammaROHF method*), 1162
`compute_nuclear_dipole()` (*ChemistryDriverPySCF-MolecularRHF method*), 1179
`compute_nuclear_dipole()` (*ChemistryDriverPySCF-MolecularRHFQMMMCOSMO method*), 1188
`compute_nuclear_dipole()` (*ChemistryDriverPySCF-MolecularROHF method*), 1197
`compute_nuclear_dipole()` (*ChemistryDriverPySCF-MolecularROHFQMMMCOSMO method*), 1206
`compute_nuclear_dipole()` (*ChemistryDriverPySCF-MolecularUHF method*), 1216
`compute_nuclear_dipole()` (*ChemistryDriverPySCF-MolecularUHFQMMMCOSMO method*), 1225
`compute_one_electron_operator()` (*ChemistryDriverPySCFEmbeddingGammaRHF method*), 1108
`compute_one_electron_operator()` (*ChemistryDriverPySCFEmbeddingGammaROHF_UHF method*), 1118
`compute_one_electron_operator()` (*ChemistryDriverPySCFEmbeddingRHF method*), 1127
`compute_one_electron_operator()` (*ChemistryDriverPySCFEmbeddingROHF method*), 1136
`compute_one_electron_operator()` (*ChemistryDriverPySCFEmbeddingROHF_UHF method*), 1145
`compute_one_electron_operator()` (*ChemistryDriverPySCFGammaRHF method*), 1153
`compute_one_electron_operator()` (*ChemistryDriverPySCFGammaROHF method*), 1162
`compute_one_electron_operator()` (*ChemistryDriverPySCFMolecularRHF method*), 1179
`compute_one_electron_operator()` (*ChemistryDriverPySCFMolecularRHFQMMMCOSMO method*), 1188
`compute_one_electron_operator()` (*ChemistryDriverPySCFMolecularROHF method*), 1197
`compute_one_electron_operator()` (*ChemistryDriverPySCFMolecularROHFQMMMCOSMO method*), 1207
`compute_one_electron_operator()` (*ChemistryDriverPySCFMolecularUHF method*), 1216
`compute_one_electron_operator()` (*ChemistryDriverPySCFMolecularUHFQMMMCOSMO method*), 1225
`compute_representation_components()` (*Point-Group method*), 1099
`compute_unitary()` (*AVAS method*), 1105
`compute_unitary()` (*CASSCF method*), 1107
`compute_unitary()` (*OrbitalOptimizer static method*), 726
`compute_unitary()` (*OrbitalTransformer static method*), 728
`ComputeUncompute` (*class in inquanto.protocols*), 919
`ComputeUncomputeFactorizedOverlap` (*class in inquanto.protocols*), 956
`conditional_exit` (*IcebergOptions attribute*), 1013
`conj()` (*ComputableNDArray method*), 497
`conjugate()` (*CacheLevels method*), 580
`conjugate()` (*CacheSizeUnit method*), 582
`conjugate()` (*CompilationLevel method*), 1014
`conjugate()` (*ComputableNDArray method*), 497
`conjugate()` (*CtrluStrat method*), 1016
`constant()` (*FermionOperator class method*), 687
`constant()` (*SymmetryOperatorFermionic class method*), 806
`construct_contraction_mask_from_operators()` (*FermionSpace method*), 1018
`construct_contraction_mask_from_operators()` (*FermionSpaceBrillouin method*), 1034
`construct_contraction_mask_from_operators()` (*FermionSpaceSupercell method*), 1041
`construct_double_excitation_operators()` (*FermionSpace method*), 1018
`construct_double_qubit_excitation_operators()` (*ParaFermionSpace method*), 1050
`construct_double_ucc_operators()` (*FermionSpace method*), 1018
`construct_fragment_energy_operator()` (*DMETRHFFragmentActive static method*), 590
`construct_fragment_energy_operator()` (*DMETRHFFragmentDirect static method*), 591
`construct_fragment_energy_operator()` (*DMETRHFFragmentPySCFActive static method*), 1239
`construct_fragment_energy_operator()` (*DMETRHFFragmentUCCSDVQE static method*), 592
`construct_from_array()` (*SymbolSet method*), 575
`construct_from_dict()` (*SymbolSet method*), 576
`construct_generalised_double_excitation_operators()` (*FermionSpace method*), 1018
`construct_generalised_double_ucc_operators()` (*FermionSpace method*), 1018
`construct_generalised_pair_double_excitation_operators()` (*FermionSpace method*), 1019

```

construct_generalised_pair_double_ucc_operators()   method), 1021
    (FermionSpace method), 1019
construct_generalised_single_excitation_operators() louin static method), 1035
    (FermionSpace method), 1019
construct_generalised_single_ucc_operators()   method), 1019
    (FermionSpace method), 1019
construct_imag_pauli_exponent_operators()   (ParaFermionSpace method), 1051
construct_imag_pauli_exponent_operators()   (QubitSpace method), 1054
construct_n_body_spinless_pdm_operators()   (FermionSpace method), 1020
construct_n_body_spinless_rdm_operators()   (FermionSpace method), 1020
construct_number_alpha_operator()   (FermionSpace method), 1020
construct_number_beta_operator()   (FermionSpace method), 1020
construct_number_operator()   (FermionSpace method), 1020
construct_number_operator()   (FermionSpaceBrillouin method), 1034
construct_number_operator()   (FermionSpaceSuperCell method), 1041
construct_one_body_operator_from_big_integral()   (FermionSpace method), 1021
construct_one_body_operator_from_integral()   (FermionSpace method), 1035
construct_one_body_spatial_rdm_operators()   (FermionSpace method), 1021
construct_operator_from_string()   (FermionSpace static method), 1021
construct_operator_from_string()   (ParaFermionSpace static method), 1051
construct_operator_from_string()   (QubitSpace static method), 1054
construct_orbital_number_operators()   (FermionSpace method), 1021
construct_permutation_operator()   (FermionSpace SuperCell method), 1042
construct_random() (SymbolSet method), 576
construct_random_parameters() (DMETRHF static method), 586
construct_random_variables()   (OrbitalOptimizer method), 727
construct_real_pauli_exponent_operators()   (ParaFermionSpace method), 1051
construct_real_pauli_exponent_operators()   (QubitSpace method), 1054
construct_reverse_rp_permutation_operator()   (FermionSpace SuperCell method), 1042
construct_scalar_operator()   (FermionSpace static
construct_scalar_operator()   (FermionSpaceBrillouin method), 1035
construct_scalar_operator()   (FermionSpaceSuperCell method), 1042
construct_scalar_operator()   (ParaFermionSpace static method), 1051
construct_shift_rp_permutation_operator()   (FermionSpace SuperCell method), 1042
construct_single_excitation_operators()   (FermionSpace method), 1022
construct_single_qubit_excitation_operators()   (ParaFermionSpace method), 1051
construct_single_ucc_operators()   (FermionSpace method), 1022
construct_singlet_double_excitation_operators()   (FermionSpace method), 1022
construct_singlet_double_ucc_operators()   (FermionSpace method), 1022
construct_singlet_generalised_double_excitation_operator()   (FermionSpace method), 1023
construct_singlet_generalised_single_excitation_operator()   (FermionSpace method), 1023
construct_singlet_generalised_single_ucc_operators()   (FermionSpace method), 1023
construct_singlet_single_excitation_operators()   (FermionSpace method), 1023
construct_singlet_single_ucc_operators()   (FermionSpace method), 1023
construct_spin_operator()   (FermionSpace method), 1024
construct_swap_rp_permutation_operator()   (FermionSpace SuperCell method), 1042
construct_symbolic_recursive_gf()   (KrylovSubspace method), 537
construct_symbolic_recursive_gf_h()   (KrylovSubspace method), 537
construct_symbolic_recursive_gf_p()   (KrylovSubspace method), 538
construct_symbolic_recursive_lanczos_gf00()   (KrylovSubspace method), 538
construct_sz_operator()   (FermionSpace method), 1024
construct_tridiagonal_representation()   (KrylovSubspace method), 538
construct_triplet_generalised_single_excitation_operator()   (FermionSpace method), 1024
construct_triplet_generalised_single_ucc_operators()   (FermionSpace method), 1024
construct_two_body_operator_from_big_integral()   (FermionSpace SuperCell method), 1043
construct_two_body_operator_from_integral()   (FermionSpace method), 1024
construct_two_body_operator_from_integral()

```

(*FermionSpaceBrillouin method*), 1035
`construct_two_body_operator_from_tensor()`
 (*FermionSpace method*), 1025
`construct_two_body_spatial_rdm_operators()`
 (*FermionSpace method*), 1025
`construct_zeros()` (*SymbolSet method*), 576
`contract_occupation_space()` (*FermionSpace method*), 1026
`contract_occupation_space()` (*FermionSpaceBrillouin method*), 1035
`contract_occupation_state()` (*FermionSpace static method*), 1026
`contract_occupation_state()` (*FermionSpaceBrillouin static method*), 1036
`contract_occupation_state()` (*FermionSpaceSupercell static method*), 1043
`contract_operator()` (*FermionSpace static method*), 1026
`contract_operator()` (*FermionSpaceBrillouin static method*), 1036
`contract_operator()` (*FermionSpaceSupercell static method*), 1043
`contract_state_mask()` (*FermionSpace static method*), 1026
`contract_state_mask()` (*FermionSpaceBrillouin static method*), 1036
`contract_state_mask()` (*FermionSpaceSupercell static method*), 1044
`contracted_system()` (*FermionSpace method*), 1026
`convert_mask_to_index_map()` (*FermionSpace static method*), 1027
`convert_mask_to_index_map()` (*FermionSpaceBrillouin static method*), 1036
`convert_mask_to_index_map()` (*FermionSpaceSupercell static method*), 1044
`copy()` (*BackendStatevectorProtocol method*), 964
`copy()` (*CanonicalPhaseEstimation method*), 986
`copy()` (*ChemistryRestrictedIntegralOperator method*), 653
`copy()` (*ChemistryRestrictedIntegralOperatorCompact method*), 658
`copy()` (*ChemistryUnrestrictedIntegralOperator method*), 664
`copy()` (*ChemistryUnrestrictedIntegralOperatorCompact method*), 668
`copy()` (*CircuitAnsatz method*), 334
`copy()` (*ComposedAnsatz method*), 340
`copy()` (*ComputableList method*), 489
`copy()` (*ComputableNDArray method*), 497
`copy()` (*FermionOperator method*), 688
`copy()` (*FermionOperatorList method*), 714
`copy()` (*FermionSpaceAnsatzChemicallyAwareUCCSD method*), 373
`copy()` (*FermionSpaceAnsatzkUpCCGD method*), 379
`copy()` (*FermionSpaceAnsatzkUpCCGSD method*), 384
`copy()` (*FermionSpaceAnsatzkUpCCGSDSinglet method*), 390
`copy()` (*FermionSpaceAnsatzUCCD method*), 363
`copy()` (*FermionSpaceAnsatzUCCGD method*), 396
`copy()` (*FermionSpaceAnsatzUCCGSD method*), 401
`copy()` (*FermionSpaceAnsatzUCCSD method*), 357
`copy()` (*FermionSpaceAnsatzUCCSDSinglet method*), 407
`copy()` (*FermionSpaceStateExp method*), 352
`copy()` (*FermionSpaceStateExpChemicallyAware method*), 368
`copy()` (*FermionState method*), 1057
`copy()` (*GeneralAnsatz method*), 329
`copy()` (*HamiltonianVariationalAnsatz method*), 446
`copy()` (*HardwareEfficientAnsatz method*), 458
`copy()` (*IterativePhaseEstimationSingleCircuit method*), 987
`copy()` (*LayeredAnsatz method*), 452
`copy()` (*MultiConfigurationAnsatz method*), 413
`copy()` (*MultiConfigurationState method*), 418
`copy()` (*MultiConfigurationStateBox method*), 423
`copy()` (*PySCFChemistryRestrictedIntegralOperator method*), 1249
`copy()` (*PySCFChemistryUnrestrictedIntegralOperator method*), 1253
`copy()` (*QubitOperator method*), 738
`copy()` (*QubitOperatorList method*), 775
`copy()` (*QubitState method*), 1072
`copy()` (*RealGeneralizedBasisRotationAnsatz method*), 428
`copy()` (*RealRestrictedBasisRotationAnsatz method*), 434
`copy()` (*RealUnrestrictedBasisRotationAnsatz method*), 439
`copy()` (*RestrictedOneBodyRDM method*), 795
`copy()` (*RestrictedTwoBodyRDM method*), 797
`copy()` (*SparseStatevectorProtocol method*), 963
`copy()` (*State method*), 1087
`copy()` (*SymbolDict method*), 572
`copy()` (*SymbolSet method*), 576
`copy()` (*SymmetryOperatorFermionic method*), 806
`copy()` (*SymmetryOperatorFermionicFactorised method*), 833
`copy()` (*SymmetryOperatorPauli method*), 850
`copy()` (*SymmetryOperatorPauliFactorised method*), 888
`copy()` (*TrotterAnsatz method*), 345
`copy()` (*UnrestrictedOneBodyRDM method*), 902
`copy()` (*UnrestrictedTwoBodyRDM method*), 904
`correlation_potential_pattern()` (*DMETRHF static method*), 586
`cost()` (*ComputeUncompute method*), 920
`cost()` (*ComputeUncomputeFactorizedOverlap method*), 958
`cost()` (*DestructiveSwapTest method*), 926
`cost()` (*FactorizedOverlap method*), 945

cost () (*HadamardTest method*), 914
 cost () (*HadamardTestDerivative method*), 975
 cost () (*HadamardTestDerivativeOverlap method*), 968
 cost () (*HadamardTestOverlap method*), 939
 cost () (*IterativePhaseEstimation method*), 988
 cost () (*IterativePhaseEstimationQuantinuum method*), 993
 cost () (*PauliAveraging method*), 907
 cost () (*PhaseShift method*), 981
 cost () (*ProjectiveMeasurements method*), 1000
 cost () (*ProtocolList method*), 1005
 cost () (*SwapFactorizedOverlap method*), 952
 cost () (*SwapTest method*), 933
 count () (*ComputableList method*), 489
 count () (*FermionOperatorList.CompressScalarsBehavior method*), 703
 count () (*FermionOperatorList.FactoryCoefficientsLocation method*), 708
 count () (*FermionOperator.TrotterizeCoefficientsLocation method*), 680
 count () (*FermionSpace method*), 1027
 count () (*FermionSpaceBrillouin method*), 1037
 count () (*FermionSpaceSupercell method*), 1044
 count () (*QubitOperatorList.CompressScalarsBehavior method*), 757
 count () (*QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method*), 762
 count () (*QubitOperatorList.FactoryCoefficientsLocation method*), 767
 count () (*QubitOperator.TrotterizeCoefficientsLocation method*), 731
 count () (*SymmetryOperatorFermionicFactored.CompressScalarsBehavior method*), 822
 count () (*SymmetryOperatorFermionicFactored.FactoryCoefficientsLocation method*), 826
 count () (*SymmetryOperatorFermionic.TrotterizeCoefficientsLocation method*), 798
 count () (*SymmetryOperatorPauliFactored.CompressScalarsBehavior method*), 870
 count () (*SymmetryOperatorPauliFactored.ExpandExponentialProductCoefficientsBehavior method*), 875
 count () (*SymmetryOperatorPauliFactored.FactoryCoefficientsLocation method*), 880
 count () (*SymmetryOperatorPauli.TrotterizeCoefficientsLocation method*), 843
 count_spin () (*ParaFermionSpace static method*), 1051
 counts (*ProjectiveMeasurements attribute*), 1000
 credits () (*ComputeUncompute method*), 920
 credits () (*ComputeUncomputeFactorizedOverlap method*), 958
 credits () (*DestructiveSwapTest method*), 927
 credits () (*FactorizedOverlap method*), 946
 credits () (*HadamardTest method*), 914
 credits () (*HadamardTestDerivative method*), 975
 credits () (*HadamardTestDerivativeOverlap method*), 969
 credits () (*HadamardTestOverlap method*), 939
 credits () (*IterativePhaseEstimation method*), 988
 credits () (*IterativePhaseEstimationQuantinuum method*), 993
 credits () (*PauliAveraging method*), 907
 credits () (*PhaseShift method*), 982
 credits () (*ProjectiveMeasurements method*), 1000
 credits () (*ProtocolList method*), 1006
 credits () (*SwapFactorizedOverlap method*), 952
 credits () (*SwapTest method*), 933
 CtrluStrat (*class in inquanto.protocols*), 1015
 ctypes (*ComputableNDArray attribute*), 498
 cumprod () (*ComputableNDArray method*), 500
 cumsum () (*ComputableNDArray method*), 500

D

dagger () (*FermionOperator method*), 688
 dagger () (*FermionOperatorString method*), 724
 dagger () (*QubitOperator method*), 738
 dagger () (*SymmetryOperatorFermionic method*), 806
 dagger () (*SymmetryOperatorPauli method*), 850
 data (*ComputableNDArray attribute*), 500
 dataframe (*GeometryMolecular property*), 607
 dataframe (*GeometryPeriodic property*), 618
 dataframe_circuit_shot () (*ComputeUncompute method*), 921
 dataframe_circuit_shot () (*ComputeUncomputeFactorizedOverlap method*), 958
 dataframe_circuit_shot () (*DestructiveSwapTest method*), 927
 dataframe_circuit_shot () (*FactorizedOverlap method*), 946
 dataframe_circuit_shot () (*HadamardTest method*), 914
 dataframe_circuit_shot () (*HadamardTestDerivative method*), 975
 dataframe_circuit_shot () (*HadamardTestDerivativeOverlap method*), 969
 dataframe_circuit_shot () (*HadamardTestOverlap method*), 940
 dataframe_circuit_shot () (*IterativePhaseEstimation method*), 989
 dataframe_circuit_shot () (*IterativePhaseEstimationQuantinuum method*), 994

defaultframe_circuit_shot()	(<i>PauliAveraging method</i>), 907	(<i>method</i>), 543
defaultframe_circuit_shot()	(<i>PhaseShift method</i>), 982	(<i>ManyBodyGFComputable method</i>), 545
defaultframe_circuit_shot()	(<i>ProjectiveMeasurements method</i>), 1001	(<i>MetricTensorImag method</i>), 475
defaultframe_circuit_shot()	(<i>ProtocolList method</i>), 1006	(<i>MetricTensorReal method</i>), 477
defaultframe_circuit_shot()	(<i>SwapFactorizedOverlap method</i>), 952	(<i>NonOrthogonalMatricesComputable method</i>), 547
defaultframe_circuit_shot()	(<i>SwapTest method</i>), 933	(<i>Overlap method</i>), 478
defaultframe_measurements()	(<i>HadamardTest method</i>), 915	(<i>OverlapImag method</i>), 480
defaultframe_measurements()	(<i>PauliAveraging method</i>), 907	(<i>OverlapMatrixComputable method</i>), 549
defaultframe_partitioning()	(<i>PauliAveraging method</i>), 907	(<i>OverlapReal method</i>), 482
defaultframe_protocol_circuit()	(<i>ProtocolList method</i>), 1006	(<i>OverlapSquared method</i>), 484
DecompositionMethod (class in <i>inquanto.operators</i>), 676		(<i>ParticleGFComputable method</i>), 552
default_evaluate()	(<i>CommutatorComputable method</i>), 532	(<i>PDM1234RealComputable method</i>), 550
default_evaluate()	(<i>ComputableFunction method</i>), 486	(<i>QCM4Computable method</i>), 554
default_evaluate()	(<i>ComputableInt method</i>), 487	(<i>QSEMatricesComputable method</i>), 556
default_evaluate()	(<i>ComputableList method</i>), 489	(<i>RDM1234RealComputable method</i>), 558
default_evaluate()	(<i>ComputableNDArray method</i>), 500	(<i>RestrictedOneBodyRDMComputable method</i>), 559
default_evaluate()	(<i>ComputableNode method</i>), 528	(<i>RestrictedOneBodyRDMRealComputable method</i>), 561
default_evaluate()	(<i>ComputableSingleChild method</i>), 529	(<i>SCEOMMatrixComputable method</i>), 563
default_evaluate()	(<i>Evaluatable method</i>), 531	(<i>SpinlessNBodyPDMArrayRealComputable method</i>), 565
default_evaluate()	(<i>ExpectationValue method</i>), 463	(<i>SpinlessNBodyRDMArrayRealComputable method</i>), 567
default_evaluate()	(<i>ExpectationValueBraDerivativeImag method</i>), 465	(<i>UnrestrictedOneBodyRDMComputable method</i>), 569
default_evaluate()	(<i>ExpectationValueBraDerivativeReal method</i>), 466	(<i>UnrestrictedOneBodyRDMRealComputable method</i>), 571
default_evaluate()	(<i>ExpectationValueDerivative method</i>), 468	default_evaluate_as_function() (<i>ManyBodyGFComputable method</i>), 545
default_evaluate()	(<i>ExpectationValueKetDerivativeImag method</i>), 470	default_pass() (<i>CircuitAnsatz method</i>), 335
default_evaluate()	(<i>ExpectationValueKetDerivativeReal method</i>), 471	default_pass() (<i>ComposedAnsatz method</i>), 340
default_evaluate()	(<i>ExpectationValueNonHermitian method</i>), 473	default_pass() (<i>FermionSpaceAnsatzChemicallyAwareUCCSD method</i>), 373
default_evaluate()	(<i>ExpectationValueSumComputable method</i>), 534	default_pass() (<i>FermionSpaceAnsatzkUpCCGD method</i>), 379
default_evaluate()	(<i>HoleGFComputable method</i>), 536	default_pass() (<i>FermionSpaceAnsatzkUpCCGSD method</i>), 384
default_evaluate()	(<i>KrylovSubspaceComputable method</i>), 539	default_pass() (<i>FermionSpaceAnsatzkUpCCGDSinglet method</i>), 390
default_evaluate()	(<i>LanczosCoefficientsComputable method</i>), 541	default_pass() (<i>FermionSpaceAnsatzUCCD method</i>), 363
default_evaluate()	(<i>LanczosMatrixComputable</i>	default_pass() (<i>FermionSpaceAnsatzUCCGD method</i>), 396

default_pass() (*FermionSpaceAnsatzUCCSD method*),
 357
 default_pass() (*FermionSpaceAnsatzUCCSDSinglet method*), 407
 default_pass() (*FermionSpaceStateExp method*), 352
 default_pass() (*FermionSpaceStateExpChemicallyAware method*), 368
 default_pass() (*GeneralAnsatz method*), 330
 default_pass() (*HamiltonianVariationalAnsatz method*), 446
 default_pass() (*HardwareEfficientAnsatz method*),
 458
 default_pass() (*LayeredAnsatz method*), 452
 default_pass() (*MultiConfigurationAnsatz method*),
 413
 default_pass() (*MultiConfigurationState method*), 418
 default_pass() (*MultiConfigurationStateBox method*),
 423
 default_pass() (*RealGeneralizedBasisRotationAnsatz method*), 429
 default_pass() (*RealRestrictedBasisRotationAnsatz method*), 434
 default_pass() (*RealUnrestrictedBasisRotationAnsatz method*), 440
 default_pass() (*TrotterAnsatz method*), 346
 delete_atom() (*GeometryMolecular method*), 607
 delete_atom() (*GeometryPeriodic method*), 619
 denominator (*CacheLevels attribute*), 580
 denominator (*CacheSizeUnit attribute*), 582
 denominator (*CompilationLevel attribute*), 1015
 denominator (*CtrluStrat attribute*), 1016
 DestructiveSwapTest (*class in inquanto.protocols*),
 925
 df() (*ChemistryRestrictedIntegralOperator method*), 653
 df() (*ChemistryRestrictedIntegralOperatorCompact method*), 659
 df() (*ChemistryUnrestrictedIntegralOperator method*), 664
 df() (*ChemistryUnrestrictedIntegralOperatorCompact method*), 669
 df() (*FermionOperator method*), 688
 df() (*FermionOperatorList method*), 714
 df() (*FermionState method*), 1057
 df() (*PySCFChemistryRestrictedIntegralOperator method*),
 1249
 df() (*PySCFChemistryUnrestrictedIntegralOperator method*), 1254
 df() (*QubitOperator method*), 738
 df() (*QubitOperatorList method*), 775
 df() (*QubitState method*), 1072
 df() (*State method*), 1087
 df() (*SymbolDict method*), 573
 df() (*SymbolSet method*), 577
 df() (*SymmetryOperatorFermionic method*), 807
 df() (*SymmetryOperatorFermionicFactorised method*),
 833
 df() (*SymmetryOperatorPauli method*), 851
 df() (*SymmetryOperatorPauliFactorised method*), 888
 df_numeric() (*CircuitAnsatz method*), 335
 df_numeric() (*ComposedAnsatz method*), 340
 df_numeric() (*FermionSpaceAnsatzChemicallyAwareUCCSD method*), 374
 df_numeric() (*FermionSpaceAnsatzkUpCCGD method*),
 379
 df_numeric() (*FermionSpaceAnsatzkUpCCGSD method*),
 385
 df_numeric() (*FermionSpaceAnsatzkUpCCGSDSinglet method*), 390
 df_numeric() (*FermionSpaceAnsatzUCCD method*), 363
 df_numeric() (*FermionSpaceAnsatzUCCGD method*),
 396
 df_numeric() (*FermionSpaceAnsatzUCCGSD method*),
 401
 df_numeric() (*FermionSpaceAnsatzUCCSD method*),
 357
 df_numeric() (*FermionSpaceAnsatzUCCSDSinglet method*), 407
 df_numeric() (*FermionSpaceStateExp method*), 352
 df_numeric() (*FermionSpaceStateExpChemicallyAware method*), 368
 df_numeric() (*GeneralAnsatz method*), 330
 df_numeric() (*HamiltonianVariationalAnsatz method*),
 446
 df_numeric() (*HardwareEfficientAnsatz method*), 458
 df_numeric() (*LayeredAnsatz method*), 453
 df_numeric() (*MultiConfigurationAnsatz method*), 413
 df_numeric() (*MultiConfigurationState method*), 418
 df_numeric() (*MultiConfigurationStateBox method*), 423
 df_numeric() (*RealGeneralizedBasisRotationAnsatz method*), 429
 df_numeric() (*RealRestrictedBasisRotationAnsatz method*), 434
 df_numeric() (*RealUnrestrictedBasisRotationAnsatz method*), 440
 df_numeric() (*TrotterAnsatz method*), 346
 df_symbolic() (*CircuitAnsatz method*), 335
 df_symbolic() (*ComposedAnsatz method*), 341
 df_symbolic() (*FermionSpaceAnsatzChemicallyAwareUCCSD method*), 374
 df_symbolic() (*FermionSpaceAnsatzkUpCCGD method*), 380
 df_symbolic() (*FermionSpaceAnsatzkUpCCGSD method*), 385
 df_symbolic() (*FermionSpaceAnsatzkUpCCGSDSinglet method*), 391
 df_symbolic() (*FermionSpaceAnsatzUCCD method*),
 364
 df_symbolic() (*FermionSpaceAnsatzUCCGD method*),
 396

df_symbolic() (*FermionSpaceAnsatzUCCGSD method*),
 402
 df_symbolic() (*FermionSpaceAnsatzUCCSD method*),
 358
 df_symbolic() (*FermionSpaceAnsatzUCCSDSinglet method*), 408
 df_symbolic() (*FermionSpaceStateExp method*), 353
 df_symbolic() (*FermionSpaceStateExpChemicallyAware method*), 369
 df_symbolic() (*GeneralAnsatz method*), 330
 df_symbolic() (*HamiltonianVariationalAnsatz method*),
 447
 df_symbolic() (*HardwareEfficientAnsatz method*), 458
 df_symbolic() (*LayeredAnsatz method*), 453
 df_symbolic() (*MultiConfigurationAnsatz method*), 413
 df_symbolic() (*MultiConfigurationState method*), 419
 df_symbolic() (*MultiConfigurationStateBox method*),
 424
 df_symbolic() (*RealGeneralizedBasisRotationAnsatz method*), 429
 df_symbolic() (*RealRestrictedBasisRotationAnsatz method*), 435
 df_symbolic() (*RealUnrestrictedBasisRotationAnsatz method*), 440
 df_symbolic() (*TrotterAnsatz method*), 346
 df_to_xyz() (*GeometryMolecular method*), 607
 df_to_xyz() (*GeometryPeriodic method*), 619
 diagonal() (*ComputableNDArray method*), 500
 dict_to_matrix() (*in module inquanto.core*), 584
 dict_to_vector() (*in module inquanto.core*), 584
 dihedral_angle() (*GeometryMolecular method*), 607
 dihedral_angle() (*GeometryPeriodic method*), 619
 discard() (*SymbolDict method*), 573
 discard() (*SymbolSet method*), 577
 distributions (*HadamardTestDerivativeOverlap attribute*), 969
 DMETRHF (*class in inquanto.embeddings.dmet*), 585
 DMETRHFFragment (*class in inquanto.embeddings.dmet*),
 588
 DMETRHFFragmentActive (*class in inquanto.embeddings.dmet*), 589
 DMETRHFFragmentDirect (*class in inquanto.embeddings.dmet*), 591
 DMETRHFFragmentPySCFActive (*class in inquanto.extensions.pyscf*), 1239
 DMETRHFFragmentPySCFCCSD (*class in inquanto.extensions.pyscf*), 1240
 DMETRHFFragmentPySCFFCI (*class in inquanto.extensions.pyscf*), 1241
 DMETRHFFragmentPySCFMP2 (*class in inquanto.extensions.pyscf*), 1242
 DMETRHFFragmentPySCFRHF (*class in inquanto.extensions.pyscf*), 1243
 DMETRHFFragmentUCCSDVQE (*class in inquanto.embeddings.dmet*), 591
 dot() (*ComputableNDArray method*), 501
 dot_state() (*QubitOperator method*), 738
 dot_state() (*QubitOperatorString method*), 789
 dot_state() (*SymmetryOperatorPauli method*), 851
 dot_state_as_linear_combination() (*QubitOperatorList method*), 775
 dot_state_as_linear_combination() (*SymmetryOperatorPauliFactorised method*), 888
 dot_state_as_product() (*QubitOperatorList method*),
 776
 dot_state_as_product() (*SymmetryOperatorPauliFactorised method*), 889
 double_factorize() (*ChemistryRestrictedIntegralOperator method*), 653
 double_factorize() (*ChemistryRestrictedIntegralOperatorCompact method*), 659
 double_factorize() (*ChemistryUnrestrictedIntegralOperator method*), 664
 double_factorize() (*ChemistryUnrestrictedIntegralOperatorCompact method*), 669
 double_factorize() (*PySCFChemistryRestrictedIntegralOperator method*), 1249
 double_factorize() (*PySCFChemistryUnrestrictedIntegralOperator method*), 1254
 DoubleFactorizedHamiltonian (*class in inquanto.operators*), 676
 DriverGeneralizedHubbard (*class in inquanto.express*), 596
 DriverHubbardDimer (*class in inquanto.express*), 598
 DriverIsing1D (*class in inquanto.express*), 599
 DriverIsing1DRing (*class in inquanto.express*), 599
 DriverIsingCustomConnectivity (*class in inquanto.express*), 598
 dtype (*ChemistryRestrictedIntegralOperator property*), 654
 dtype (*ChemistryRestrictedIntegralOperatorCompact property*), 660
 dtype (*CompactTwoBodyIntegralsS4 property*), 673
 dtype (*CompactTwoBodyIntegralsS8 property*), 675
 dtype (*ComputableNDArray attribute*), 501
 dump() (*ComputableNDArray method*), 501
 dump() (*ComputeUncompute method*), 921
 dump() (*DestructiveSwapTest method*), 927
 dump() (*HadamardTest method*), 915
 dump() (*HadamardTestDerivativeOverlap method*), 969
 dump() (*PauliAveraging method*), 908
 dump() (*ProjectiveMeasurements method*), 1001
 dump() (*SwapTest method*), 933
 dump_flags() (*AVAS method*), 1105
 dumps() (*ComputableNDArray method*), 502
 dumps() (*ComputeUncompute method*), 921
 dumps() (*DestructiveSwapTest method*), 927
 dumps() (*HadamardTest method*), 915
 dumps() (*HadamardTestDerivativeOverlap method*), 969

dumps () (*PauliAveraging method*), 908
 dumps () (*ProjectiveMeasurements method*), 1001
 dumps () (*SwapTest method*), 933

E

effective_potential() (*ChemistryRestrictedIntegralOperator method*), 654
 effective_potential() (*ChemistryRestrictedIntegralOperatorCompact method*), 660
 effective_potential() (*ChemistryUnrestrictedIntegralOperator method*), 665
 effective_potential() (*ChemistryUnrestrictedIntegralOperatorCompact method*), 670
 effective_potential() (*PySCFChemistryRestrictedIntegralOperator method*), 1250
 effective_potential() (*PySCFChemistryUnrestrictedIntegralOperator method*), 1255
 effective_potential_spin() (*ChemistryRestrictedIntegralOperator method*), 654
 effective_potential_spin() (*ChemistryRestrictedIntegralOperatorCompact method*), 660
 effective_potential_spin() (*PySCFChemistryRestrictedIntegralOperator method*), 1250
 EIG (*DecompositionMethod attribute*), 676
 eigenspectrum() (*QubitOperator method*), 739
 eigenspectrum() (*SymmetryOperatorPauli method*), 851
 eigenvalues (*IterativePhaseEstimationStatevector property*), 997
 eigenvalues() (*KrylovSubspace method*), 538
 elements (*GeometryMolecular property*), 607
 elements (*GeometryPeriodic property*), 619
 empty() (*FermionOperator method*), 688
 empty() (*FermionOperatorList method*), 714
 empty() (*FermionState method*), 1057
 empty() (*QubitOperator method*), 739
 empty() (*QubitOperatorList method*), 776
 empty() (*QubitState method*), 1072
 empty() (*State method*), 1087
 empty() (*SymmetryOperatorFermionic method*), 807
 empty() (*SymmetryOperatorFermionicFactorised method*), 833
 empty() (*SymmetryOperatorPauli method*), 852
 empty() (*SymmetryOperatorPauliFactorised method*), 889
 encode() (*FermionOperatorList.CompressScalarsBehavior method*), 703
 encode() (*FermionOperatorList.FactoryCoefficientsLocation method*), 708
 encode() (*FermionOperator.TrotterizeCoefficientsLocation method*), 680

encode() (*QubitOperatorList.CompressScalarsBehavior method*), 757
 encode() (*QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method*), 762
 encode() (*QubitOperatorList.FactoryCoefficientsLocation method*), 767
 encode() (*QubitOperator.TrotterizeCoefficientsLocation method*), 731
 encode() (*SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method*), 822
 encode() (*SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method*), 826
 encode() (*SymmetryOperatorFermionic.TrotterizeCoefficientsLocation method*), 799
 encode() (*SymmetryOperatorPauliFactorised.CompressScalarsBehavior method*), 870
 encode() (*SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method*), 875
 encode() (*SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method*), 880
 encode() (*SymmetryOperatorPauli.TrotterizeCoefficientsLocation method*), 843
 endswith() (*FermionOperatorList.CompressScalarsBehavior method*), 703
 endswith() (*FermionOperatorList.FactoryCoefficientsLocation method*), 708
 endswith() (*FermionOperator.TrotterizeCoefficientsLocation method*), 680
 endswith() (*QubitOperatorList.CompressScalarsBehavior method*), 757
 endswith() (*QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method*), 762
 endswith() (*QubitOperatorList.FactoryCoefficientsLocation method*), 767
 endswith() (*QubitOperator.TrotterizeCoefficientsLocation method*), 731
 endswith() (*SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method*), 822

```

endswith()           (SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation      method), 827
endswith()           (SymmetryOperatorFermionic.TrotterizeCoefficientsLocation      method), 799
endswith()           (SymmetryOperatorPauliFactorised.CompressScalarsBehavior      method), 870
endswith()           (SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior      method), 875
endswith()           (SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation      method), 880
endswith()           (SymmetryOperatorPauli.TrotterizeCoefficientsLocation      method), 843
energy()            (ChemistryRestrictedIntegralOperator      method), 655
energy()            (ChemistryRestrictedIntegralOperatorCompact      method), 660
energy()            (ChemistryUnrestrictedIntegralOperator      method), 665
energy()            (ChemistryUnrestrictedIntegralOperatorCompact      method), 670
energy()            (DMETRHF      method), 587
energy()            (FMO      method), 1259
energy()            (PySCFChemistryRestrictedIntegralOperator      method), 1250
energy()            (PySCFChemistryUnrestrictedIntegralOperator      method), 1255
energy_electron_mean_field()    (ChemistryRestrictedIntegralOperator      method), 655
energy_electron_mean_field()    (ChemistryRestrictedIntegralOperatorCompact      method), 660
energy_electron_mean_field()    (ChemistryUnrestrictedIntegralOperator      method), 665
energy_electron_mean_field()    (ChemistryUnrestrictedIntegralOperatorCompact      method), 670
energy_electron_mean_field()    (PySCFChemistryRestrictedIntegralOperator      method), 1251
energy_electron_mean_field()    (PySCFChemistryUnrestrictedIntegralOperator      method), 1255
ensure_hermitian()        (QubitOperator      method), 739
ensure_hermitian()        (SymmetryOperatorPauli      method), 852
equality_matrix()         (QubitOperatorList      method), 776
equality_matrix()         (SymmetryOperatorPauliFactorised      method), 889
evalf()                (FermionOperator      method), 688
evalf()                (FermionOperatorList      method), 714
evalf()                (FermionState      method), 1057
evalf()                (QubitOperator      method), 740
evalf()                (QubitOperatorList      method), 777
evalf()                (QubitState      method), 1072
evalf()                (State      method), 1087
evalf()                (SymmetryOperatorFermionic      method), 807
evalf()                (SymmetryOperatorFermionicFactorised      method), 833
evalf()                (SymmetryOperatorPauli      method), 852
evalf()                (SymmetryOperatorPauliFactorised      method), 890
for evaluatable (class in inquanto.computables.primitive), 531
evaluate()             (CommutatorComputable      method), 532
evaluate()             (ComputableFunction      method), 486
evaluate()             (ComputableInt      method), 488
evaluate()             (ComputableList      method), 490
evaluate()             (ComputableNDArray      method), 502
evaluate()             (ComputableNode      method), 528
evaluate()             (ComputableSingleChild      method), 530
evaluate()             (ComputableTuple      method), 531
evaluate()             (Evaluatable      method), 531
evaluate()             (ExpectationValue      method), 463
evaluate()             (ExpectationValueBraDerivativeImag      method), 465
evaluate()             (ExpectationValueBraDerivativeReal      method), 467
evaluate()             (ExpectationValueDerivative      method), 468
evaluate()             (ExpectationValueKetDerivativeImag      method), 470
evaluate()             (ExpectationValueKetDerivativeReal      method), 472
evaluate()             (ExpectationValueNonHermitian      method), 473
evaluate()             (ExpectationValueSumComputable      method), 534
evaluate()             (HoleGFComputable      method), 536
evaluate()             (KrylovSubspaceComputable      method), 540
evaluate()             (LanczosCoefficientsComputable      method), 541
evaluate()             (LanczosMatrixComputable      method), 543
evaluate()             (ManyBodyGFComputable      method), 545
evaluate()             (MetricTensorImag      method), 475
evaluate()             (MetricTensorReal      method), 477
evaluate()             (NonOrthogonalMatricesComputable      method), 547
evaluate()             (Overlap      method), 479
evaluate()             (OverlapImag      method), 480
evaluate()             (OverlapMatrixComputable      method), 549
evaluate()             (OverlapReal      method), 482
evaluate()             (OverlapSquared      method), 484
evaluate()             (ParticleGFComputable      method), 553
evaluate()             (PDM1234RealComputable      method), 551
evaluate()             (QCM4Computable      method), 554
evaluate()             (QSEMatricesComputable      method), 556

```

evaluate() (*RDM1234RealComputable method*), 558
 evaluate() (*RestrictedOneBodyRDMComputable method*), 560
 evaluate() (*RestrictedOneBodyRDMRealComputable method*), 561
 evaluate() (*SCEOMMatrixComputable method*), 563
 evaluate() (*SpinlessNBodyPDMArrayRealComputable method*), 566
 evaluate() (*SpinlessNBodyRDMArrayRealComputable method*), 568
 evaluate() (*UnrestrictedOneBodyRDMComputable method*), 569
 evaluate() (*UnrestrictedOneBodyRDMRealComputable method*), 571
 evaluate_as_function() (*ManyBodyGFComputable method*), 545
 evaluate_dbra() (*HadamardTestDerivative method*), 975
 evaluate_dbra() (*PhaseShift method*), 982
 evaluate_derivative_overlap() (*HadamardTest-DerivativeOverlap method*), 969
 evaluate_dket() (*HadamardTestDerivative method*), 976
 evaluate_dket() (*PhaseShift method*), 982
 evaluate_expectation_uvalue() (*PauliAveraging method*), 908
 evaluate_expectation_value() (*HadamardTest method*), 915
 evaluate_expectation_value() (*PauliAveraging method*), 908
 evaluate_gradient() (*HadamardTestDerivative method*), 976
 evaluate_gradient() (*PhaseShift method*), 983
 evaluate_overlap() (*ComputeUncomputeFactorizedOverlap method*), 958
 evaluate_overlap() (*FactorizedOverlap method*), 946
 evaluate_overlap() (*HadamardTestOverlap method*), 940
 evaluate_overlap() (*SwapFactorizedOverlap method*), 952
 evaluate_overlap_imag() (*ComputeUncomputeFactorizedOverlap method*), 959
 evaluate_overlap_imag() (*FactorizedOverlap method*), 946
 evaluate_overlap_imag() (*HadamardTestOverlap method*), 940
 evaluate_overlap_imag() (*SwapFactorizedOverlap method*), 953
 evaluate_overlap_real() (*ComputeUncomputeFactorizedOverlap method*), 959
 evaluate_overlap_real() (*FactorizedOverlap method*), 947
 evaluate_overlap_real() (*HadamardTestOverlap method*), 940
 evaluate_overlap_real() (*SwapFactorizedOverlap method*), 953
 evaluate_overlap_squared() (*ComputeUncompute method*), 921
 evaluate_overlap_squared() (*DestructiveSwapTest method*), 927
 evaluate_overlap_squared() (*SwapTest method*), 934
 expand_exponential_product_commuting_operators() (*QubitOperatorList method*), 777
 expand_exponential_product_commuting_operators() (*SymmetryOperatorPauliFactorised method*), 890
 expandtabs() (*FermionOperatorList.CompressScalarsBehavior method*), 703
 expandtabs() (*FermionOperatorList.FactoryCoefficientsLocation method*), 708
 expandtabs() (*FermionOperator.TrotterizeCoefficientsLocation method*), 680
 expandtabs() (*QubitOperatorList.CompressScalarsBehavior method*), 757
 expandtabs() (*QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method*), 762
 expandtabs() (*QubitOperatorList.FactoryCoefficientsLocation method*), 767
 expandtabs() (*QubitOperator.TrotterizeCoefficientsLocation method*), 731
 expandtabs() (*SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method*), 822
 expandtabs() (*SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method*), 827
 expandtabs() (*SymmetryOperatorFermionic.TrotterizeCoefficientsLocation method*), 799
 expandtabs() (*SymmetryOperatorPauliFactorised.CompressScalarsBehavior method*), 870
 expandtabs() (*SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method*), 875
 expandtabs() (*SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method*), 880
 expandtabs() (*SymmetryOperatorPauli.TrotterizeCoefficientsLocation method*), 844

ExpectationValue (class in *inquito.computables.atomic*), 462
 ExpectationValueBraDerivativeImag (class in *inquito.computables.atomic*), 464
 ExpectationValueBraDerivativeReal (class in *inquito.computables.atomic*), 466
 ExpectationValueDerivative (class in *inquito.computables.atomic*), 467
 ExpectationValueKetDerivativeImag (class in *inquito.computables.atomic*), 469
 ExpectationValueKetDerivativeReal (class in *inquito.computables.atomic*), 471
 ExpectationValueNonHermitian (class in *inquito.computables.atomic*), 473
 ExpectationValueSumComputable (class in *inquito.computables.composite*), 533
 exponentiate_commuting_operator() (*QubitOperator* method), 740
 exponentiate_commuting_operator() (*SymmetryOperatorPauli* method), 852
 exponentiate_single_term() (*QubitOperator* method), 740
 exponentiate_single_term() (*SymmetryOperatorPauli* method), 853
 exponents (*FermionSpaceAnsatzkUpCCGD* property), 380
 exponents (*FermionSpaceAnsatzkUpCCGSD* property), 386
 exponents (*FermionSpaceAnsatzkUpCCGSDSinglet* property), 391
 exponents (*FermionSpaceAnsatzUCCD* property), 364
 exponents (*FermionSpaceAnsatzUCCGD* property), 397
 exponents (*FermionSpaceAnsatzUCCGSD* property), 403
 exponents (*FermionSpaceAnsatzUCCSD* property), 359
 exponents (*FermionSpaceAnsatzUCCSDSinglet* property), 408
 exponents (*FermionSpaceStateExp* property), 353
 exponents (*HamiltonianVariationalAnsatz* property), 448
 exponents (*TrotterAnsatz* property), 347
 extend() (*ComputableList* method), 490
 extract_point_group_information() (*ChemistryDriverPySCFEmbeddingGammaRHF* method), 1109
 extract_point_group_information() (*ChemistryDriverPySCFEmbeddingGammaROHF_UHF* method), 1118
 extract_point_group_information() (*ChemistryDriverPySCFEmbeddingRHF* method), 1127
 extract_point_group_information() (*ChemistryDriverPySCFEmbeddingROHF* method), 1136
 extract_point_group_information() (*ChemistryDriverPySCFEmbeddingROHF_UHF* method), 1145
 extract_point_group_information() (*ChemistryDriverPySCFGammaRHF* method), 1154
 extract_point_group_information() (*ChemistryDriverPySCFGammaROHF* method), 1163
 extract_point_group_information() (*ChemistryDriverPySCFMolecularRHF* method), 1179
 extract_point_group_information() (*ChemistryDriverPySCFMolecularRHFQMMMCOSMO* method), 1189
 extract_point_group_information() (*ChemistryDriverPySCFMolecularROHF* method), 1198
 extract_point_group_information() (*ChemistryDriverPySCFMolecularROHFQMMMCOSMO* method), 1207
 extract_point_group_information() (*ChemistryDriverPySCFMolecularUHF* method), 1216
 extract_point_group_information() (*ChemistryDriverPySCFMolecularUHFQMMMCOSMO* method), 1225

F

F_CONTIGUOUS (*ComputableNDArray* attribute), 503
 FactorizedOverlap (class in *inquito.protocols*), 944
 factors() (*KrylovSubspace* method), 538
 FARRAY (*ComputableNDArray* attribute), 503
 FCIDumpRestricted (class in *inquito.operators*), 677
 FERMION_ANNIHILATION (*FermionOperatorString* attribute), 723
 FERMION_CREATION (*FermionOperatorString* attribute), 723
 fermion_operator_exponents (*FermionSpaceAnsatzChemicallyAwareUCCSD* property), 375
 fermion_operator_exponents (*FermionSpaceAnsatzkUpCCGD* property), 380
 fermion_operator_exponents (*FermionSpaceAnsatzkUpCCGSD* property), 386
 fermion_operator_exponents (*FermionSpaceAnsatzkUpCCGSDSinglet* property), 391
 fermion_operator_exponents (*FermionSpaceAnsatzUCCD* property), 364
 fermion_operator_exponents (*FermionSpaceAnsatzUCCGD* property), 397
 fermion_operator_exponents (*FermionSpaceAnsatzUCCGSD* property), 403
 fermion_operator_exponents (*FermionSpaceAnsatzUCCSD* property), 359
 fermion_operator_exponents (*FermionSpaceAnsatzUCCSDSinglet* property), 408
 fermion_operator_exponents (*FermionSpaceStateExp* property), 353
 fermion_operator_exponents (*FermionSpaceStateExpChemicallyAware* property), 370
 FermionOperator (class in *inquito.operators*), 679

FermionOperatorList (*class in inquanto.operators*),
702
 FermionOperatorList.CompressScalarsBehavior
(*class in inquanto.operators*), 702
 FermionOperatorList.FactoryCoefficientsLocation
(*class in inquanto.operators*), 707
 FermionOperatorString (*class in inquanto.operators*),
723
 FermionOperator.TrotterizeCoefficientsLocation
(*class in inquanto.operators*), 679
 FermionSpace (*class in inquanto.spaces*), 1017
 FermionSpaceAnsatzChemicallyAwareUCCSD (*class
in inquanto.ansatzes*), 373
 FermionSpaceAnsatzkUpCCGD (*class in in-
quanto.ansatzes*), 378
 FermionSpaceAnsatzkUpCCGSD (*class in in-
quanto.ansatzes*), 384
 FermionSpaceAnsatzkUpCCGSDSinglet (*class in in-
quanto.ansatzes*), 389
 FermionSpaceAnsatzUCCD (*class in inquanto.ansatzes*),
362
 FermionSpaceAnsatzUCCGD (*class in in-
quanto.ansatzes*), 395
 FermionSpaceAnsatzUCCGSD (*class in in-
quanto.ansatzes*), 401
 FermionSpaceAnsatzUCCSD (*class in in-
quanto.ansatzes*), 357
 FermionSpaceAnsatzUCCSDSinglet (*class in in-
quanto.ansatzes*), 406
 FermionSpaceBrillouin (*class in inquanto.spaces*),
1034
 FermionSpaceStateExp (*class in inquanto.ansatzes*),
351
 FermionSpaceStateExpChemicallyAware (*class in
inquanto.ansatzes*), 368
 FermionSpaceSupercell (*class in inquanto.spaces*),
1040
 FermionState (*class in inquanto.states*), 1055
 FermionStateString (*class in inquanto.states*), 1067
 fill () (*ComputableNDArray method*), 502
 final_energy () (*AlgorithmDeterministicQPE method*),
322
 final_evaluated_auxiliary_expression (*Algo-
rithmVQE property*), 317
 final_evaluated_objective_expression (*Algo-
rithmVQE property*), 317
 final_parameters (*AlgorithmAdaptVQE property*), 315
 final_parameters (*AlgorithmFermionicAdaptVQE
property*), 314
 final_parameters (*AlgorithmIQEB property*), 316
 final_parameters (*AlgorithmMcLachlanImagTime
property*), 328
 final_parameters (*AlgorithmMcLachlanRealTime
property*), 327
 final_parameters (*AlgorithmVQD property*), 319
 final_parameters (*AlgorithmVQE property*), 317
 final_parameters (*AlgorithmVQS property*), 326
 final_pdf () (*AlgorithmBayesianQPE method*), 1266
 final_pdf () (*AlgorithmInfoTheoryQPE method*), 324
 final_phase () (*AlgorithmDeterministicQPE method*),
322
 final_propagation_evaluation () (*AlgorithmM-
cLachlanImagTime method*), 328
 final_propagation_evaluation () (*AlgorithmM-
cLachlanRealTime method*), 327
 final_propagation_evaluation () (*AlgorithmVQS
method*), 326
 final_states (*AlgorithmQSE property*), 319
 final_states (*AlgorithmSCEOM property*), 320
 final_value (*AlgorithmVQE property*), 318
 final_value () (*AlgorithmBayesianQPE method*), 1266
 final_value () (*AlgorithmInfoTheoryQPE method*), 324
 final_value () (*AlgorithmKitaevQPE method*), 325
 final_values (*AlgorithmQSE property*), 319
 final_values (*AlgorithmSCEOM property*), 320
 final_values (*AlgorithmVQD property*), 319
 find () (*FermionOperatorList.CompressScalarsBehavior
method*), 703
 find () (*FermionOperatorList.FactoryCoefficientsLocation
method*), 708
 find () (*FermionOperator.TrotterizeCoefficientsLocation
method*), 680
 find () (*QubitOperatorList.CompressScalarsBehavior
method*), 758
 find () (*QubitOperatorList.ExpandExponentialProductCoefficientsBehavior
method*), 762
 find () (*QubitOperatorList.FactoryCoefficientsLocation
method*), 767
 find () (*QubitOperator.TrotterizeCoefficientsLocation
method*), 731
 find () (*SymmetryOperatorFermionicFac-
torised.CompressScalarsBehavior method*),
822
 find () (*SymmetryOperatorFermionicFac-
torised.FactoryCoefficientsLocation method*),
827
 find () (*SymmetryOperator-
Fermionic.TrotterizeCoefficientsLocation
method*), 799
 find () (*SymmetryOperatorPauliFac-
torised.CompressScalarsBehavior method*),
870
 find () (*SymmetryOperatorPauliFac-
torised.ExpandExponentialProductCoefficientsBehavior
method*), 875
 find () (*SymmetryOperatorPauliFac-
torised.FactoryCoefficientsLocation method*),
880

```

find()                               (SymmetryOperator-
    PauliTrotterizeCoefficientsLocation   method), 799
    844
find_x_operators() (TapererZ2 method), 1101
flags (ComputableNDArray attribute), 503
flat (ComputableNDArray attribute), 504
flatten() (ComputableNDArray method), 504
flip_set() (QubitMapping class method), 626
flip_set() (QubitMappingBravyiKitaev class method),
    633
flip_set() (QubitMappingJordanWigner class method),
    630
flip_set() (QubitMappingParaparticular class method),
    640
flip_set() (QubitMappingParity class method), 637
FMO (class in inquanto.extensions.pyscf.fmo), 1258
FMOFragment (class in inquanto.extensions.pyscf.fmo),
    1259
FMOFragmentPySCFActive (class in in-
    quanto.extensions.pyscf.fmo), 1260
FMOFragmentPySCFCCSD (class in in-
    quanto.extensions.pyscf.fmo), 1261
FMOFragmentPySCFMP2 (class in in-
    quanto.extensions.pyscf.fmo), 1262
FMOFragmentPySCFRHF (class in in-
    quanto.extensions.pyscf.fmo), 1263
FNC (ComputableNDArray attribute), 503
FORC (ComputableNDArray attribute), 503
format()                               (FermionOpera-
    torList.CompressScalarsBehavior       method), 703
format()                               (FermionOpera-
    torList.FactoryCoefficientsLocation method),
    708
format()                               (FermionOpera-
    tor.TrotterizeCoefficientsLocation method),
    680
format_map()                          (QubitOpera-
    torList.CompressScalarsBehavior       method),
    758
format_map()                          (QubitOpera-
    torList.ExpandExponentialProductCoefficientsBehavior
    method), 763
format_map()                          (QubitOpera-
    torList.FactoryCoefficientsLocation method),
    767
format_map()                          (QubitOpera-
    tor.TrotterizeCoefficientsLocation method),
    731
format_map()                          (SymmetryOperatorFermionicFac-
    torised.CompressScalarsBehavior       method),
    822
format_map()                          (SymmetryOperatorFermionicFac-
    torised.FactoryCoefficientsLocation method),
    827
format_map()                          (SymmetryOperatorPauliFac-
    torised.CompressScalarsBehavior       method),
    871
format_map()                          (SymmetryOperatorPauliFac-
    torised.ExpandExponentialProductCoefficientsBehavior
    method), 876
format_map()                          (SymmetryOperatorPauliFac-
    torised.FactoryCoefficientsLocation method),
    880
format_map()                          (SymmetryOperator-
    PauliTrotterizeCoefficientsLocation method),

```

844
`free_symbols()` (*CircuitAnsatz method*), 336
`free_symbols()` (*CommutatorComputable method*), 533
`free_symbols()` (*ComposedAnsatz method*), 341
`free_symbols()` (*ComputableFunction method*), 486
`free_symbols()` (*ComputableInt method*), 488
`free_symbols()` (*ComputableList method*), 490
`free_symbols()` (*ComputableNDArray method*), 505
`free_symbols()` (*ComputableNode method*), 528
`free_symbols()` (*ComputableSingleChild method*), 530
`free_symbols()` (*ExpectationValue method*), 463
`free_symbols()` (*ExpectationValueBraDerivativeImag method*), 465
`free_symbols()` (*ExpectationValueBraDerivativeReal method*), 467
`free_symbols()` (*ExpectationValueDerivative method*), 468
`free_symbols()` (*ExpectationValueKetDerivativeImag method*), 470
`free_symbols()` (*ExpectationValueKetDerivativeReal method*), 472
`free_symbols()` (*ExpectationValueNonHermitian method*), 474
`free_symbols()` (*ExpectationValueSumComputable method*), 534
`free_symbols()` (*FermionOperator method*), 688
`free_symbols()` (*FermionOperatorList method*), 715
`free_symbols()` (*FermionSpaceAnsatzChemicallyAwareUCCSD method*), 375
`free_symbols()` (*FermionSpaceAnsatzkUpCCGD method*), 380
`free_symbols()` (*FermionSpaceAnsatzkUpCCGSD method*), 386
`free_symbols()` (*FermionSpaceAnsatzkUpCCGSDSinglet method*), 392
`free_symbols()` (*FermionSpaceAnsatzUCCD method*), 364
`free_symbols()` (*FermionSpaceAnsatzUCCGD method*), 397
`free_symbols()` (*FermionSpaceAnsatzUCCGSD method*), 403
`free_symbols()` (*FermionSpaceAnsatzUCCSD method*), 359
`free_symbols()` (*FermionSpaceAnsatzUCCSDSinglet method*), 408
`free_symbols()` (*FermionSpaceStateExp method*), 353
`free_symbols()` (*FermionSpaceStateExpChemicallyAware method*), 370
`free_symbols()` (*FermionState method*), 1058
`free_symbols()` (*GeneralAnsatz method*), 331
`free_symbols()` (*HamiltonianVariationalAnsatz method*), 448
`free_symbols()` (*HardwareEfficientAnsatz method*), 459
`free_symbols()` (*HoleGFComputable method*), 536
`free_symbols()` (*KrylovSubspaceComputable method*), 540
`free_symbols()` (*LanczosCoefficientsComputable method*), 541
`free_symbols()` (*LanczosMatrixComputable method*), 543
`free_symbols()` (*LayeredAnsatz method*), 454
`free_symbols()` (*ManyBodyGFComputable method*), 545
`free_symbols()` (*MetricTensorImag method*), 475
`free_symbols()` (*MetricTensorReal method*), 477
`free_symbols()` (*MultiConfigurationAnsatz method*), 414
`free_symbols()` (*MultiConfigurationState method*), 419
`free_symbols()` (*MultiConfigurationStateBox method*), 424
`free_symbols()` (*NonOrthogonalMatricesComputable method*), 547
`free_symbols()` (*Overlap method*), 479
`free_symbols()` (*OverlapImag method*), 481
`free_symbols()` (*OverlapMatrixComputable method*), 549
`free_symbols()` (*OverlapReal method*), 482
`free_symbols()` (*OverlapSquared method*), 484
`free_symbols()` (*ParticleGFComputable method*), 553
`free_symbols()` (*PDM1234RealComputable method*), 551
`free_symbols()` (*QCM4Computable method*), 554
`free_symbols()` (*QSEMatricesComputable method*), 556
`free_symbols()` (*QubitOperator method*), 741
`free_symbols()` (*QubitOperatorList method*), 778
`free_symbols()` (*QubitState method*), 1072
`free_symbols()` (*RDM1234RealComputable method*), 558
`free_symbols()` (*RealGeneralizedBasisRotationAnsatz method*), 430
`free_symbols()` (*RealRestrictedBasisRotationAnsatz method*), 435
`free_symbols()` (*RealUnrestrictedBasisRotationAnsatz method*), 441
`free_symbols()` (*RestrictedOneBodyRDMComputable method*), 560
`free_symbols()` (*RestrictedOneBodyRDMRealComputable method*), 561
`free_symbols()` (*SCEOMMatrixComputable method*), 564
`free_symbols()` (*SpinlessNBodyPDMArrayRealComputable method*), 566
`free_symbols()` (*SpinlessNBodyRDMArrayRealComputable method*), 568
`free_symbols()` (*State method*), 1087
`free_symbols()` (*SymmetryOperatorFermionic method*), 807

free_symbols() (*SymmetryOperatorFermionicFactorised method*), 833
 free_symbols() (*SymmetryOperatorPauli method*), 853
 free_symbols() (*SymmetryOperatorPauliFactorised method*), 891
 free_symbols() (*TrotterAnsatz method*), 347
 free_symbols() (*UnrestrictedOneBodyRDMComputable method*), 569
 free_symbols() (*UnrestrictedOneBodyRDMRealComputable method*), 571
 free_symbols_ordered() (*CircuitAnsatz method*), 336
 free_symbols_ordered() (*CommutatorComputable method*), 533
 free_symbols_ordered() (*ComposedAnsatz method*), 341
 free_symbols_ordered() (*ComputableFunction method*), 486
 free_symbols_ordered() (*ComputableInt method*), 488
 free_symbols_ordered() (*ComputableList method*), 490
 free_symbols_ordered() (*ComputableNDArray method*), 505
 free_symbols_ordered() (*ComputableNode method*), 528
 free_symbols_ordered() (*ComputableSingleChild method*), 530
 free_symbols_ordered() (*ExpectationValue method*), 463
 free_symbols_ordered() (*ExpectationValue-BraDerivativeImag method*), 465
 free_symbols_ordered() (*ExpectationValue-BraDerivativeReal method*), 467
 free_symbols_ordered() (*ExpectationValueDerivative method*), 469
 free_symbols_ordered() (*ExpectationValueKet-DerivativeImag method*), 470
 free_symbols_ordered() (*ExpectationValueKet-DerivativeReal method*), 472
 free_symbols_ordered() (*ExpectationValueNonHermitian method*), 474
 free_symbols_ordered() (*ExpectationValueSumComputable method*), 534
 free_symbols_ordered() (*FermionOperator method*), 688
 free_symbols_ordered() (*FermionOperatorList method*), 715
 free_symbols_ordered() (*FermionSpaceAnsatz-ChemicallyAwareUCCSD method*), 375
 free_symbols_ordered() (*Fermion-SpaceAnsatzUpCCGD method*), 380
 free_symbols_ordered() (*Fermion-SpaceAnsatzUpCCGSD method*), 386
 free_symbols_ordered() (*Fermion-SpaceAnsatzkUpCCGSDSinglet method*), 392
 free_symbols_ordered() (*Fermion-SpaceAnsatzUCCD method*), 364
 free_symbols_ordered() (*FermionSpaceAnsatzUC-CGD method*), 397
 free_symbols_ordered() (*FermionSpaceAnsatzUC-CGSD method*), 403
 free_symbols_ordered() (*Fermion-SpaceAnsatzUCCSD method*), 359
 free_symbols_ordered() (*Fermion-SpaceAnsatzUCCSDSinglet method*), 408
 free_symbols_ordered() (*FermionSpaceStateExp method*), 353
 free_symbols_ordered() (*FermionSpaceStateExp-ChemicallyAware method*), 370
 free_symbols_ordered() (*FermionState method*), 1058
 free_symbols_ordered() (*GeneralAnsatz method*), 331
 free_symbols_ordered() (*HamiltonianVariationalAnsatz method*), 448
 free_symbols_ordered() (*HardwareEfficientAnsatz method*), 459
 free_symbols_ordered() (*HoleGFComputable method*), 536
 free_symbols_ordered() (*KrylovSubspaceComputable method*), 540
 free_symbols_ordered() (*LanczosCoefficientsComputable method*), 542
 free_symbols_ordered() (*LanczosMatrixComputable method*), 543
 free_symbols_ordered() (*LayeredAnsatz method*), 454
 free_symbols_ordered() (*ManyBodyGFComputable method*), 546
 free_symbols_ordered() (*MetricTensorImag method*), 475
 free_symbols_ordered() (*MetricTensorReal method*), 477
 free_symbols_ordered() (*MultiConfigurationAnsatz method*), 414
 free_symbols_ordered() (*MultiConfigurationState method*), 419
 free_symbols_ordered() (*MultiConfigurationState-Box method*), 424
 free_symbols_ordered() (*NonOrthogonalMatricesComputable method*), 547
 free_symbols_ordered() (*Overlap method*), 479
 free_symbols_ordered() (*OverlapImag method*), 481
 free_symbols_ordered() (*OverlapMatrixComputable method*), 549
 free_symbols_ordered() (*OverlapReal method*), 482
 free_symbols_ordered() (*OverlapSquared method*), 484

free_symbols_ordered() (*ParticleGFComputable method*), 553
 free_symbols_ordered() (*PDM1234RealComputable method*), 551
 free_symbols_ordered() (*QCM4Computable method*), 555
 free_symbols_ordered() (*QSEMatricesComputable method*), 556
 free_symbols_ordered() (*QubitOperator method*), 741
 free_symbols_ordered() (*QubitOperatorList method*), 778
 free_symbols_ordered() (*QubitState method*), 1072
 free_symbols_ordered() (*RDM1234RealComputable method*), 558
 free_symbols_ordered() (*RealGeneralizedBasisRotationAnsatz method*), 430
 free_symbols_ordered() (*RealRestrictedBasisRotationAnsatz method*), 435
 free_symbols_ordered() (*RealUnrestrictedBasisRotationAnsatz method*), 441
 free_symbols_ordered() (*RestrictedOneBodyRDM-Computable method*), 560
 free_symbols_ordered() (*RestrictedOneBodyRDM-RealComputable method*), 561
 free_symbols_ordered() (*SCEOMMatrixComputable method*), 564
 free_symbols_ordered() (*SpinlessNBodyPDMArrayRealComputable method*), 566
 free_symbols_ordered() (*SpinlessNBodyRDMArrrayRealComputable method*), 568
 free_symbols_ordered() (*State method*), 1087
 free_symbols_ordered() (*SymmetryOperator-Fermionic method*), 807
 free_symbols_ordered() (*SymmetryOperator-FermionicFactorised method*), 834
 free_symbols_ordered() (*SymmetryOperatorPauli method*), 853
 free_symbols_ordered() (*SymmetryOperatorPauli-Factorised method*), 891
 free_symbols_ordered() (*TrotterAnsatz method*), 347
 free_symbols_ordered() (*UnrestrictedOne-BodyRDMComputable method*), 570
 free_symbols_ordered() (*UnrestrictedOne-BodyRDMRealComputable method*), 571
 freeze() (*FermionOperator method*), 688
 freeze() (*SymmetryOperatorFermionic method*), 807
 from_array() (*SymbolDict method*), 573
 from_bytes() (*CacheLevels method*), 580
 from_bytes() (*CacheSizeUnit method*), 582
 from_bytes() (*CompilationLevel method*), 1015
 from_bytes() (*CtrluStrat method*), 1016
 from_circuit() (*SymbolDict class method*), 573
 from_circuit() (*SymbolSet class method*), 577
 from_fcidump() (*ChemistryRestrictedIntegralOperator static method*), 655
 from_FermionOperator() (*ChemistryRestrictedIntegralOperator class method*), 655
 from_FermionOperator() (*ChemistryRestrictedIntegralOperatorCompact class method*), 660
 from_FermionOperator() (*ChemistryUnrestrictedIntegralOperator class method*), 665
 from_FermionOperator() (*ChemistryUnrestrictedIntegralOperatorCompact class method*), 670
 from_index() (*FermionStateString class method*), 1067
 from_index() (*QubitStateString class method*), 1083
 from_index() (*StateString class method*), 1097
 from_integral_operator() (*ChemistryDriver-PySCFIntegrals class method*), 1171
 from_list() (*FermionOperator class method*), 689
 from_list() (*QubitOperator class method*), 741
 from_list() (*QubitOperatorList class method*), 779
 from_list() (*QubitOperatorString class method*), 790
 from_list() (*SymmetryOperatorFermionic class method*), 807
 from_list() (*SymmetryOperatorPauli class method*), 854
 from_list() (*SymmetryOperatorPauliFactorised class method*), 892
 from_list_int() (*FermionStateString class method*), 1068
 from_list_int() (*QubitStateString class method*), 1083
 from_list_int() (*StateString class method*), 1097
 from_mf() (*ChemistryDriverPySCFEmbeddingGammaRHF class method*), 1109
 from_mf() (*ChemistryDriverPySCFEmbeddingGammaROHF_UHF class method*), 1118
 from_mf() (*ChemistryDriverPySCFEmbeddingRHF class method*), 1128
 from_mf() (*ChemistryDriverPySCFEmbeddingROHF class method*), 1136
 from_mf() (*ChemistryDriverPySCFEmbeddingROHF_UHF class method*), 1145
 from_mf() (*ChemistryDriverPySCFGammaRHF class method*), 1154
 from_mf() (*ChemistryDriverPySCFGammaROHF class method*), 1163
 from_mf() (*ChemistryDriverPySCFMolecularRHF class method*), 1179
 from_mf() (*ChemistryDriverPySCFMolecularRHFQM-MMCOSMO class method*), 1189
 from_mf() (*ChemistryDriverPySCFMolecularROHFQM-MMCOSMO class method*), 1207
 from_mf() (*ChemistryDriverPySCFMolecularUHF class method*), 1216
 from_mf() (*ChemistryDriverPySCFMolecularUHFQM-MMCOSMO class method*), 1223

MMCOSMO class method), 1226
from_mf() (ChemistryDriverPySCFMomentumRHF class method), 1234
from_mf() (ChemistryDriverPySCFMomentumROHF class method), 1237
from_ndarray() (FermionState class method), 1058
from_ndarray() (QubitState class method), 1072
from_ndarray() (State class method), 1087
from_Operator() (FermionOperatorList class method), 715
from_Operator() (QubitOperatorList class method), 779
from_Operator() (SymmetryOperatorFermionicFactorised class method), 834
from_Operator() (SymmetryOperatorPauliFactorised class method), 891
from_QubitPauliOperator() (QubitOperator class method), 741
from_QubitPauliOperator() (SymmetryOperatorPauli class method), 853
from_QubitPauliString() (QubitOperatorString class method), 790
from_sparray() (FermionState class method), 1058
from_sparray() (QubitState class method), 1073
from_sparray() (State class method), 1088
from_state() (FermionSpace static method), 1027
from_string() (FermionOperator class method), 689
from_string() (FermionOperatorList class method), 715
from_string() (FermionOperatorString class method), 725
from_string() (FermionState class method), 1059
from_string() (FermionStateString class method), 1068
from_string() (QubitOperator class method), 741
from_string() (QubitOperatorList class method), 780
from_string() (QubitOperatorString class method), 790
from_string() (QubitState class method), 1073
from_string() (QubitStateString class method), 1083
from_string() (State class method), 1088
from_string() (StateString class method), 1097
from_string() (SymmetryOperatorFermionic class method), 808
from_string() (SymmetryOperatorFermionicFactorised class method), 834
from_string() (SymmetryOperatorPauli class method), 854
from_string() (SymmetryOperatorPauliFactorised class method), 892
from_symplectic_row() (QubitOperatorString class method), 791
from_tuple() (FermionOperator class method), 689
from_tuple() (QubitOperatorString class method), 791
from_tuple() (SymmetryOperatorFermionic class method), 808
from_uncompacted_integrals() (CompactTwo-BodyIntegralsS4 class method), 673
from_uncompacted_integrals() (CompactTwo-BodyIntegralsS8 class method), 675
from_xyz_string() (GeometryMolecular static method), 607
from_xyz_string() (GeometryPeriodic static method), 619
FromActiveOrbitals (class in inquanto.extensions.pyscf), 1244
FromActiveSpace (class in inquanto.extensions.pyscf), 1244
frozen (ChemistryDriverPySCFEmbeddingGammaRHF property), 1110
frozen (ChemistryDriverPySCFEmbeddingGammaROHF_UHF property), 1119
frozen (ChemistryDriverPySCFEmbeddingRHF property), 1128
frozen (ChemistryDriverPySCFEmbeddingROHF property), 1137
frozen (ChemistryDriverPySCFEmbeddingROHF_UHF property), 1146
frozen (ChemistryDriverPySCFGammaRHF property), 1154
frozen (ChemistryDriverPySCFGammaROHF property), 1163
frozen (ChemistryDriverPySCFIntegrals property), 1171
frozen (ChemistryDriverPySCFMolecularRHF property), 1180
frozen (ChemistryDriverPySCFMolecularRHFQMMM-COSMO property), 1189
frozen (ChemistryDriverPySCFMolecularROHF property), 1198
frozen (ChemistryDriverPySCFMolecularROHFQMMM-COSMO property), 1208
frozen (ChemistryDriverPySCFMolecularUHF property), 1217
frozen (ChemistryDriverPySCFMolecularUHFQMMM-COSMO property), 1226
frozen (ChemistryDriverPySCFMomentumRHF property), 1234
frozen (ChemistryDriverPySCFMomentumROHF property), 1237
FrozenCore (class in inquanto.extensions.pyscf), 1245
frozenf() (AVAS method), 1105
func (ComputableFunction attribute), 486

G

GB (*CacheSizeUnit attribute*), 581
GeneralAnsatz (class in inquanto.ansatzes), 329
generalized_basis_rotation_to_circuit() (in module inquanto.ansatzes), 444
generate_chain() (DriverGeneralizedHubbard static method), 597
generate_cyclic_masks() (FermionSpace method), 1027

generate_cyclic_masks() (FermionSpaceSupercell method),	1044	generate_report() (ChemistryDriverPySCFMolecular-RHFQMMMCOSMO method),	1189
generate_cyclic_window_mask() (FermionSpace method),	1028	generate_report() (ChemistryDriverPySCFMolecular-ROHF method),	1198
generate_cyclic_window_mask() (FermionSpaceSupercell method),	1045	generate_report() (ChemistryDriverPySCFMolecular-ROHFQMMMCOSMO method),	1208
generate_fock_state_from_list() (FermionSpaceSupercell method),	1045	generate_report() (ChemistryDriverPySCFMolecularUHF method),	1217
generate_fock_state_from_spatial_big_occupation()	generate_report() (ChemistryDriverPySCFMolecularUHFQMMMCOSMO method),	1226	
generate_fock_state_from_spatial_occupation()	generate_report() (ChemistryDriverPySCFMomentumRHF method),	1234	
generate_occupation_state() (FermionSpace method),	1028	generate_report() (ChemistryDriverPySCFMomentumROHF method),	1237
generate_occupation_state() (FermionSpaceBrillouin method),	1037	generate_report() (CircuitAnsatz method),	336
generate_occupation_state() (FermionSpaceSupercell method),	1046	generate_report() (ComposedAnsatz method),	341
generate_occupation_state_from_list() (FermionSpace method),	1028	generate_report() (DriverGeneralizedHubbard method),	597
generate_occupation_state_from_list() (FermionSpaceBrillouin method),	1037	generate_report() (DriverHubbardDimer method),	598
generate_occupation_state_from_spatial_occupation()	generate_report() (DriverIsing1D method),	599	
(FermionSpace method),	1029	generate_report() (DriverIsing1DRing method),	599
generate_occupation_state_from_spatial_occupation()	generate_report() (DriverIsingCustomConnectivity method),	598	
(FermionSpaceBrillouin method),	1037	generate_report() (FermionSpaceAnsatzChemicallyAwareUCCSD method),	375
generate_report() (AlgorithmAdaptVQE method),	315	generate_report() (FermionSpaceAnsatzkUpCCGD method),	380
generate_report() (AlgorithmDeterministicQPE method),	322	generate_report() (FermionSpaceAnsatzkUpCCGSD method),	386
generate_report() (AlgorithmFermionicAdaptVQE method),	314	generate_report() (FermionSpaceAnsatzkUpCCGS-DSinglet method),	392
generate_report() (AlgorithmIQEB method),	316	generate_report() (FermionSpaceAnsatzUCCD method),	364
generate_report() (AlgorithmQSE method),	319	generate_report() (FermionSpaceAnsatzUCCGD method),	397
generate_report() (AlgorithmSCEOM method),	320	generate_report() (FermionSpaceAnsatzUCCGSD method),	403
generate_report() (AlgorithmVQD method),	319	generate_report() (FermionSpaceAnsatzUCCSD method),	359
generate_report() (AlgorithmVQE method),	318	generate_report() (FermionSpaceAnsatzUCCSDSinglet method),	408
generate_report() (ChemistryDriverPySCFEmbeddingGammaRHF method),	1110	generate_report() (FermionSpaceStateExp method),	353
generate_report() (ChemistryDriverPySCFEmbeddingGammaROHF_UHF method),	1119	generate_report() (FermionSpaceStateExpChemicallyAware method),	370
generate_report() (ChemistryDriverPySCFEmbeddingRHF method),	1128	generate_report() (GeneralAnsatz method),	331
generate_report() (ChemistryDriverPySCFEmbeddinggROHF method),	1137	generate_report() (HamiltonianVariationalAnsatz method),	448
generate_report() (ChemistryDriverPySCFEmbeddinggROHF_UHF method),	1146	generate_report() (HardwareEfficientAnsatz method),	459
generate_report() (ChemistryDriverPySCFGammaRHF method),	1154	generate_report() (LayeredAnsatz method),	454
generate_report() (ChemistryDriverPySCFGammaROHF method),	1164	generate_report() (MinimizerRotosolve method),	644
generate_report() (ChemistryDriverPySCFIntegrals method),	1171	generate_report() (MinimizerScipy method),	646
generate_report() (ChemistryDriverPySCFMolecular-RHF method),	1180		

generate_report() (*MinimizerSGD method*), 645
 generate_report() (*MinimizerSPSA method*), 645
 generate_report() (*MultiConfigurationAnsatz method*), 414
 generate_report() (*MultiConfigurationState method*), 419
 generate_report() (*MultiConfigurationStateBox method*), 425
 generate_report() (*OrbitalOptimizer method*), 727
 generate_report() (*RealGeneralizedBasisRotationAnsatz method*), 430
 generate_report() (*RealRestrictedBasisRotationAnsatz method*), 435
 generate_report() (*RealUnrestrictedBasisRotationAnsatz method*), 441
 generate_report() (*SymbolDict method*), 573
 generate_report() (*TrotterAnsatz method*), 347
 generate_ring() (*DriverGeneralizedHubbard static method*), 597
 generate_subspace_singles() (*FermionSpace method*), 1029
 generate_subspace_singlet_singles() (*FermionSpace method*), 1029
 generate_subspace_triplet_singles() (*FermionSpace method*), 1029
GeometryMolecular (*class in inquanto.geometries*), 603
GeometryPeriodic (*class in inquanto.geometries*), 615
 get() (*QubitOperator method*), 741
 get() (*SymmetryOperatorPauli method*), 854
 get_ac0_correction() (*ChemistryDriverPySCFEmbeddingGammaRHF method*), 1110
 get_ac0_correction() (*ChemistryDriverPySCFEmbeddingGammaROHF_UHF method*), 1119
 get_ac0_correction() (*ChemistryDriverPySCFEmbeddingRHF method*), 1128
 get_ac0_correction() (*ChemistryDriverPySCFEmbeddingROHF method*), 1137
 get_ac0_correction() (*ChemistryDriverPySCFEmbeddingROHF_UHF method*), 1146
 get_ac0_correction() (*ChemistryDriverPySCFGammaRHF method*), 1155
 get_ac0_correction() (*ChemistryDriverPySCFGammaROHF method*), 1164
 get_ac0_correction() (*ChemistryDriverPySCFIntegrals method*), 1171
 get_ac0_correction() (*ChemistryDriverPySCFMolecularRHF method*), 1180
 get_ac0_correction() (*ChemistryDriverPySCFMolecularRHFQMMMCOSMO method*), 1189
 get_ac0_correction() (*ChemistryDriverPySCFMolecularROHF method*), 1198
 get_ac0_correction() (*ChemistryDriverPySCFMolecularROHFQMMMCOSMO method*), 1208
 get_ac0_correction() (*ChemistryDriverPySCFMolec-*
ularUHF method), 1217
 get_ac0_correction() (*ChemistryDriverPySCFMolecularUHFQMMMCOSMO method*), 1226
 get_ansatz() (*AlgorithmFermionicAdaptVQE method*), 314
 get_block() (*RestrictedOneBodyRDM method*), 795
 get_block() (*UnrestrictedOneBodyRDM method*), 902
 get_casci_12rdms() (*ChemistryDriverPySCFEmbeddingGammaRHF method*), 1110
 get_casci_12rdms() (*ChemistryDriverPySCFEmbeddingGammaROHF_UHF method*), 1119
 get_casci_12rdms() (*ChemistryDriverPySCFEmbeddingRHF method*), 1128
 get_casci_12rdms() (*ChemistryDriverPySCFEmbeddingROHF method*), 1137
 get_casci_12rdms() (*ChemistryDriverPySCFEmbeddingROHF_UHF method*), 1146
 get_casci_12rdms() (*ChemistryDriverPySCFGammaRHF method*), 1155
 get_casci_12rdms() (*ChemistryDriverPySCFGammaROHF method*), 1164
 get_casci_12rdms() (*ChemistryDriverPySCFIntegrals method*), 1172
 get_casci_12rdms() (*ChemistryDriverPySCFMolecularRHF method*), 1180
 get_casci_12rdms() (*ChemistryDriverPySCFMolecularRHFQMMMCOSMO method*), 1190
 get_casci_12rdms() (*ChemistryDriverPySCFMolecularROHF method*), 1199
 get_casci_12rdms() (*ChemistryDriverPySCFMolecularROHFQMMMCOSMO method*), 1208
 get_casci_12rdms() (*ChemistryDriverPySCFMolecularUHF method*), 1217
 get_casci_12rdms() (*ChemistryDriverPySCFMolecularUHFQMMMCOSMO method*), 1227
 get_casci_12rdms() (*ChemistryDriverPySCFMomentumRHF method*), 1234
 get_casci_12rdms() (*ChemistryDriverPySCFMomentumROHF method*), 1237
 get_casci_1234pdms() (*ChemistryDriverPySCFEmbeddingGammaRHF method*), 1110
 get_casci_1234pdms() (*ChemistryDriverPySCFEmbeddingGammaROHF_UHF method*), 1119
 get_casci_1234pdms() (*ChemistryDriverPySCFEmbeddingRHF method*), 1128
 get_casci_1234pdms() (*ChemistryDriverPySCFEmbeddingROHF method*), 1137
 get_casci_1234pdms() (*ChemistryDriverPySCFEmbeddingROHF_UHF method*), 1146
 get_casci_1234pdms() (*ChemistryDriverPySCFGammaRHF method*), 1155
 get_casci_1234pdms() (*ChemistryDriverPySCFGammaROHF method*), 1164
 get_casci_1234pdms() (*ChemistryDriverPySCFInte-*

grals method), 1172
`get_cascl_1234pdms () (ChemistryDriverPySCFMolecularRHF method), 1180`
`get_cascl_1234pdms () (ChemistryDriverPySCFMolecularRHFQMMMCOSMO method), 1190`
`get_cascl_1234pdms () (ChemistryDriverPySCFMolecularROHF method), 1199`
`get_cascl_1234pdms () (ChemistryDriverPySCFMolecularROHFQMMMCOSMO method), 1208`
`get_cascl_1234pdms () (ChemistryDriverPySCFMolecularUHF method), 1217`
`get_cascl_1234pdms () (ChemistryDriverPySCFMolecularUHFQMMMCOSMO method), 1227`
`get_cascl_1234pdms () (ChemistryDriverPySCFMomentumRHF method), 1234`
`get_cascl_1234pdms () (ChemistryDriverPySCFMomentumROHF method), 1237`
`get_circuit () (CircuitAnsatz method), 336`
`get_circuit () (ComposedAnsatz method), 342`
`get_circuit () (FermionSpaceAnsatzChemicallyAwareUCCSD method), 375`
`get_circuit () (FermionSpaceAnsatzkUpCCGD method), 380`
`get_circuit () (FermionSpaceAnsatzkUpCCGSD method), 386`
`get_circuit () (FermionSpaceAnsatzkUpCCGSDSinglet method), 392`
`get_circuit () (FermionSpaceAnsatzUCCD method), 364`
`get_circuit () (FermionSpaceAnsatzUCCGD method), 397`
`get_circuit () (FermionSpaceAnsatzUCCGSD method), 403`
`get_circuit () (FermionSpaceAnsatzUCCSD method), 359`
`get_circuit () (FermionSpaceAnsatzUCCSDSinglet method), 408`
`get_circuit () (FermionSpaceStateExp method), 353`
`get_circuit () (FermionSpaceStateExpChemicallyAware method), 370`
`get_circuit () (GeneralAnsatz method), 331`
`get_circuit () (HamiltonianVariationalAnsatz method), 448`
`get_circuit () (HardwareEfficientAnsatz method), 459`
`get_circuit () (LayeredAnsatz method), 454`
`get_circuit () (MultiConfigurationAnsatz method), 414`
`get_circuit () (MultiConfigurationState method), 420`
`get_circuit () (MultiConfigurationStateBox method), 425`
`get_circuit () (RealGeneralizedBasisRotationAnsatz method), 430`
`get_circuit () (RealRestrictedBasisRotationAnsatz method), 436`
`get_circuit () (RealUnrestrictedBasisRotationAnsatz method), 441`
`get_circuit_no_ref () (TrotterAnsatz method), 347`
`get_circuits () (ComputeUncompute method), 921`
`get_circuits () (ComputeUncomputeFactorizedOverlap method), 959`
`get_circuits () (DestructiveSwapTest method), 928`
`get_circuits () (FactorizedOverlap method), 947`
`get_circuits () (HadamardTest method), 915`
`get_circuits () (HadamardTestDerivative method), 977`

get_circuits() (*HadamardTestDerivativeOverlap method*), 970
 get_circuits() (*HadamardTestOverlap method*), 941
 get_circuits() (*IterativePhaseEstimation method*), 989
 get_circuits() (*IterativePhaseEstimationQuantinuum method*), 994
 get_circuits() (*PauliAveraging method*), 908
 get_circuits() (*PhaseShift method*), 983
 get_circuits() (*ProjectiveMeasurements method*), 1001
 get_circuits() (*ProtocolList method*), 1006
 get_circuits() (*SwapFactorizedOverlap method*), 953
 get_circuits() (*SwapTest method*), 934
 get_circuitshots() (*ComputeUncompute method*), 922
 get_circuitshots() (*ComputeUncomputeFactorizedOverlap method*), 959
 get_circuitshots() (*DestructiveSwapTest method*), 928
 get_circuitshots() (*FactorizedOverlap method*), 947
 get_circuitshots() (*HadamardTest method*), 915
 get_circuitshots() (*HadamardTestDerivative method*), 977
 get_circuitshots() (*HadamardTestDerivativeOverlap method*), 970
 get_circuitshots() (*HadamardTestOverlap method*), 941
 get_circuitshots() (*IterativePhaseEstimation method*), 989
 get_circuitshots() (*IterativePhaseEstimationQuantinuum method*), 994
 get_circuitshots() (*PauliAveraging method*), 909
 get_circuitshots() (*PhaseShift method*), 983
 get_circuitshots() (*ProjectiveMeasurements method*), 1001
 get_circuitshots() (*ProtocolList method*), 1006
 get_circuitshots() (*SwapFactorizedOverlap method*), 953
 get_circuitshots() (*SwapTest method*), 934
 get_correlation_potential_pattern() (in module `inquanto.extensions.pyscf`), 1257
 get_cube_density() (*ChemistryDriverPySCFEmbeddingGammaRHF method*), 1110
 get_cube_density() (*ChemistryDriverPySCFEmbeddingGammaROHF_UHF method*), 1120
 get_cube_density() (*ChemistryDriverPySCFEmbeddingRHF method*), 1129
 get_cube_density() (*ChemistryDriverPySCFEmbeddingROHF method*), 1138
 get_cube_density() (*ChemistryDriverPySCFEmbeddingROHF_UHF method*), 1147
 get_cube_density() (*ChemistryDriverPySCFGammaRHF method*), 1155
 get_cube_density() (*ChemistryDriverPySCFGam-*
maRHF method), 1164
 get_cube_density() (*ChemistryDriverPySCFMolecularRHF method*), 1180
 get_cube_density() (*ChemistryDriverPySCFMolecularRHFQMMMCOSMO method*), 1190
 get_cube_density() (*ChemistryDriverPySCFMolecularROHF method*), 1199
 get_cube_density() (*ChemistryDriverPySCFMolecularROHFQMMMCOSMO method*), 1208
 get_cube_density() (*ChemistryDriverPySCFMolecularUHF method*), 1217
 get_cube_density() (*ChemistryDriverPySCFMolecularUHFQMMMCOSMO method*), 1227
 get_cube_orbitals() (*ChemistryDriverPySCFEmbeddingGammaRHF method*), 1111
 get_cube_orbitals() (*ChemistryDriverPySCFEmbeddingGammaROHF_UHF method*), 1120
 get_cube_orbitals() (*ChemistryDriverPySCFEmbeddingRHF method*), 1129
 get_cube_orbitals() (*ChemistryDriverPySCFEmbeddingROHF_UHF method*), 1147
 get_cube_orbitals() (*ChemistryDriverPySCFGammaRHF method*), 1155
 get_cube_orbitals() (*ChemistryDriverPySCFGammaROHF method*), 1164
 get_cube_orbitals() (*ChemistryDriverPySCFMolecularRHF method*), 1181
 get_cube_orbitals() (*ChemistryDriverPySCFMolecularRHFQMMMCOSMO method*), 1190
 get_cube_orbitals() (*ChemistryDriverPySCFMolecularROHF method*), 1199
 get_cube_orbitals() (*ChemistryDriverPySCFMolecularROHFQMMMCOSMO method*), 1209
 get_cube_orbitals() (*ChemistryDriverPySCFMolecularUHF method*), 1218
 get_cube_orbitals() (*ChemistryDriverPySCFMolecularUHFQMMMCOSMO method*), 1227
 get_dataframe_basis_states() (*ProjectiveMeasurements method*), 1001
 get_dataframe_derivative_overlap() (*HadamardTestDerivativeOverlap method*), 970
 get_dataframe_sceom_analysis() (*Algorithm-SCEOM method*), 320
 get_distribuition() (*IterativePhaseEstimation method*), 989
 get_distribuition() (*IterativePhaseEstimationQuantinuum method*), 994
 get_distribuition() (*IterativePhaseEstimationStat-evector method*), 997
 get_distribution() (*IterativePhaseEstimationStat-evector method*), 997

get_distribution()	(<i>ProjectiveMeasurements method</i>), 1001	get_evaluator()	(<i>SparseStatevectorProtocol method</i>), 963
get_dominant_basis_states()	(<i>ProjectiveMeasurements method</i>), 1001	get_evaluator()	(<i>SwapFactorizedOverlap method</i>), 953
get_double_factorized_system()	(<i>ChemistryDriverPySCFEmbeddingGammaRHF method</i>), 1111	get_evaluator()	(<i>SwapTest method</i>), 934
get_double_factorized_system()	(<i>ChemistryDriverPySCFEmbeddingGammaROHF_UHF method</i>), 1120	get_evaluator()	(<i>SymbolicProtocol method</i>), 965
get_double_factorized_system()	(<i>ChemistryDriverPySCFEmbeddingRHF method</i>), 1129	get_excitation_amplitudes()	(<i>ChemistryDriverPySCFEmbeddingGammaRHF method</i>), 1112
get_double_factorized_system()	(<i>ChemistryDriverPySCFEmbeddingROHF method</i>), 1138	get_excitation_amplitudes()	(<i>ChemistryDriverPySCFEmbeddingGammaROHF_UHF method</i>), 1121
get_double_factorized_system()	(<i>ChemistryDriverPySCFEmbeddingROHF_UHF method</i>), 1147	get_excitation_amplitudes()	(<i>ChemistryDriverPySCFEmbeddingRHF method</i>), 1130
get_double_factorized_system()	(<i>ChemistryDriverPySCFGammaRHF method</i>), 1156	get_excitation_amplitudes()	(<i>ChemistryDriverPySCFEmbeddingROHF method</i>), 1139
get_double_factorized_system()	(<i>ChemistryDriverPySCFGammaROHF method</i>), 1165	get_excitation_amplitudes()	(<i>ChemistryDriverPySCFEmbeddingROHF_UHF method</i>), 1148
get_double_factorized_system()	(<i>ChemistryDriverPySCFIntegrals method</i>), 1172	get_excitation_amplitudes()	(<i>ChemistryDriverPySCFGammaRHF method</i>), 1157
get_double_factorized_system()	(<i>ChemistryDriverPySCFMolecularRHF method</i>), 1181	get_excitation_amplitudes()	(<i>ChemistryDriverPySCFGammaROHF method</i>), 1166
get_double_factorized_system()	(<i>ChemistryDriverPySCFMolecularRHFQMMMCOSMO method</i>), 1190	get_excitation_amplitudes()	(<i>ChemistryDriverPySCFIntegrals method</i>), 1173
get_double_factorized_system()	(<i>ChemistryDriverPySCFMolecularROHF method</i>), 1199	get_excitation_amplitudes()	(<i>ChemistryDriverPySCFMolecularRHF method</i>), 1182
get_double_factorized_system()	(<i>ChemistryDriverPySCFMolecularROHFQMMMCOSMO method</i>), 1209	get_excitation_amplitudes()	(<i>ChemistryDriverPySCFMolecularRHFQMMMCOSMO method</i>), 1192
get_double_factorized_system()	(<i>ChemistryDriverPySCFMolecularUHF method</i>), 1218	get_excitation_amplitudes()	(<i>ChemistryDriverPySCFMolecularROHF method</i>), 1201
get_double_factorized_system()	(<i>ChemistryDriverPySCFMolecularUHFQMMMCOSMO method</i>), 1227	get_excitation_amplitudes()	(<i>ChemistryDriverPySCFMolecularROHFQMMMCOSMO method</i>), 1210
get_evaluator()	(<i>BackendStatevectorProtocol method</i>), 964	get_excitation_amplitudes()	(<i>ChemistryDriverPySCFMolecularUHF method</i>), 1219
get_evaluator()	(<i>ComputeUncompute method</i>), 922	get_excitation_amplitudes()	(<i>ChemistryDriverPySCFMolecularUHFQMMMCOSMO method</i>), 1229
get_evaluator()	(<i>ComputeUncomputeFactorizedOverlap method</i>), 959	get_excitation_amplitudes()	(<i>ChemistryDriverPySCFMomentumRHF method</i>), 1234
get_evaluator()	(<i>DestructiveSwapTest method</i>), 928	get_excitation_amplitudes()	(<i>ChemistryDriverPySCFMomentumROHF method</i>), 1237
get_evaluator()	(<i>FactorizedOverlap method</i>), 947	get_excitation_operators()	(<i>ChemistryDriverPySCFEmbeddingGammaRHF method</i>), 1112
get_evaluator()	(<i>HadamardTest method</i>), 915	get_excitation_operators()	(<i>ChemistryDriverPySCFEmbeddingGammaROHF_UHF method</i>), 1122
get_evaluator()	(<i>HadamardTestDerivative method</i>), 977	get_excitation_operators()	(<i>ChemistryDriverPySCFEmbeddingRHF method</i>), 1131
get_evaluator()	(<i>HadamardTestDerivativeOverlap method</i>), 970	get_excitation_operators()	(<i>ChemistryDriverPySCFEmbeddingROHF method</i>), 1140
get_evaluator()	(<i>HadamardTestOverlap method</i>), 941	get_excitation_operators()	(<i>ChemistryDriverPySCFEmbeddingROHF_UHF method</i>), 1149
get_evaluator()	(<i>PauliAveraging method</i>), 909	get_excitation_operators()	(<i>ChemistryDriverPySCFEmbeddingGammaRHF method</i>), 1112
get_evaluator()	(<i>PhaseShift method</i>), 983	get_excitation_operators()	(<i>ChemistryDriverPySCFEmbeddingGammaROHF_UHF method</i>), 1122
get_evaluator()	(<i>ProtocolList method</i>), 1006	get_excitation_operators()	(<i>ChemistryDriverPySCFEmbeddingROHF method</i>), 1130

PySCFGammaRHF method), 1157
`get_excitation_operators()` (*ChemistryDriver-PySCFGammaROHF method), 1166*
`get_excitation_operators()` (*ChemistryDriver-PySCFIInternals method), 1174*
`get_excitation_operators()` (*ChemistryDriver-PySCFMolecularRHF method), 1183*
`get_excitation_operators()` (*ChemistryDriver-PySCFMolecularROHFQMMMCOSMO method), 1192*
`get_excitation_operators()` (*ChemistryDriver-PySCFMolecularROHF method), 1201*
`get_excitation_operators()` (*ChemistryDriver-PySCFMolecularROHFQMMMCOSMO method), 1210*
`get_excitation_operators()` (*ChemistryDriver-PySCFMolecularUHF method), 1219*
`get_excitation_operators()` (*ChemistryDriver-PySCFMolecularUHFQMMMCOSMO method), 1229*
`get_excitation_operators()` (*ChemistryDriver-PySCFMomentumRHF method), 1235*
`get_excitation_operators()` (*ChemistryDriver-PySCFMomentumROHF method), 1237*
`get_exponents_with_symbols()` (*Algorithm-FermionicAdaptVQE method), 314*
`get_fragment_orbital_masks()` (*in module in-quanto.extensions.pyscf), 1257*
`get_fragment_orbitals()` (*in module in-quanto.extensions.pyscf), 1258*
`get_generators_symbol2irrep_dict()` (*Point-Group static method), 1099*
`get_iqpe_circuit` (*IterativePhaseEstimation property), 989*
`get_iqpe_circuit` (*IterativePhaseEstimationQuantin-uum property), 994*
`get_irrep2symbol_dict()` (*PointGroup static method), 1099*
`get_lowdin_system()` (*ChemistryDriverPySCFEmbeddingGammaRHF method), 1113*
`get_lowdin_system()` (*ChemistryDriverPySCFEmbeddingGammaROHF_UHF method), 1122*
`get_lowdin_system()` (*ChemistryDriverPySCFEmbeddingRHF method), 1131*
`get_lowdin_system()` (*ChemistryDriverPySCFEmbeddingROHF method), 1140*
`get_lowdin_system()` (*ChemistryDriverPySCFEmbeddingROHF_UHF method), 1149*
`get_lowdin_system()` (*ChemistryDriverPySCFGam-maRHF method), 1158*
`get_lowdin_system()` (*ChemistryDriverPySCFGam-maROHF method), 1167*
`get_lowdin_system()` (*ChemistryDriverPySCFI-*
get_lowdin_system() (*ChemistryDriverPySCFMolecu-larRHF method), 1183*
get_lowdin_system() (*ChemistryDriverPySCFMolecu-larRHF method), 1183*
get_lowdin_system() (*ChemistryDriverPySCFMolecu-larRHFQMMMCOSMO method), 1192*
get_lowdin_system() (*ChemistryDriverPySCFMolecu-larROHF method), 1201*
get_lowdin_system() (*ChemistryDriverPySCFMolecu-larROHFQMMMCOSMO method), 1211*
get_lowdin_system() (*ChemistryDriverPySCFMolecu-larUHF method), 1220*
get_lowdin_system() (*ChemistryDriverPySCFMolecu-larUHFQMMMCOSMO method), 1229*
get_madelung_constant() (*ChemistryDriver-PySCFEmbeddingGammaRHF method), 1113*
get_madelung_constant() (*ChemistryDriver-PySCFEmbeddingGammaROHF_UHF method), 1122*
get_madelung_constant() (*ChemistryDriver-PySCFGammaRHF method), 1158*
get_madelung_constant() (*ChemistryDriver-PySCFGammaROHF method), 1167*
get_madelung_constant() (*ChemistryDriverPySCF-MomentumRHF method), 1235*
get_madelung_constant() (*ChemistryDriverPySCF-MomentumROHF method), 1238*
get_measurement_outcome() (*IterativePhaseEstima-tion method), 989*
get_measurement_outcome() (*IterativePhaseEstima-tionQuantinuum method), 994*
get_measurement_outcome() (*IterativePhaseEstima-tionStatevector method), 997*
get_mm_coulomb() (*ChemistryDriverPySCFMolecular-RHFQMMMCOSMO static method), 1193*
get_mm_coulomb() (*ChemistryDriverPySCFMolecular-ROHFQMMMCOSMO static method), 1211*
get_mm_coulomb() (*ChemistryDriverPySCFMolecu-larUHFQMMMCOSMO static method), 1230*
get_mulliken_pop() (*ChemistryDriverPySCFEmbed-dingGammaRHF method), 1113*
get_mulliken_pop() (*ChemistryDriverPySCFEmbed-dingGammaROHF_UHF method), 1122*
get_mulliken_pop() (*ChemistryDriverPySCFEmbed-dingRHF method), 1131*
get_mulliken_pop() (*ChemistryDriverPySCFEmbed-dingROHF method), 1140*
get_mulliken_pop() (*ChemistryDriverPySCFEmbed-dingROHF_UHF method), 1149*
get_mulliken_pop() (*ChemistryDriverPySCFGam-maRHF method), 1158*
get_mulliken_pop() (*ChemistryDriverPySCFGam-maROHF method), 1167*
get_mulliken_pop() (*ChemistryDriverPySCFMolecu-larRHF method), 1183*
get_mulliken_pop() (*ChemistryDriverPySCFMolecu-*

<i>larRHFQMMMCOSMO method), 1193</i>	<i>SpaceAnsatzUCCD method), 365</i>
<code>get_mulliken_pop()</code> (<i>ChemistryDriverPySCFMolecularROHF method), 1202</i>	<code>get_numeric_representation()</code> (<i>FermionSpaceAnsatzUCCG method), 398</i>
<code>get_mulliken_pop()</code> (<i>ChemistryDriverPySCFMolecularROHFQMMMCOSMO method), 1211</i>	<code>get_numeric_representation()</code> (<i>FermionSpaceAnsatzUCCGSD method), 403</i>
<code>get_mulliken_pop()</code> (<i>ChemistryDriverPySCFMolecularUHF method), 1220</i>	<code>get_numeric_representation()</code> (<i>FermionSpaceAnsatzUCCSD method), 359</i>
<code>get_mulliken_pop()</code> (<i>ChemistryDriverPySCFMolecularUHFQMMMCOSMO method), 1230</i>	<code>get_numeric_representation()</code> (<i>FermionSpaceAnsatzUCCSDSinglet method), 409</i>
<code>get_nevpt2_correction()</code> (<i>ChemistryDriverPySCFEmbeddingGammaRHF method), 1113</i>	<code>get_numeric_representation()</code> (<i>FermionSpaceStateExp method), 354</i>
<code>get_nevpt2_correction()</code> (<i>ChemistryDriverPySCFEmbeddingGammaROHF_UHF method), 1123</i>	<code>get_numeric_representation()</code> (<i>FermionSpaceStateExpChemicallyAware method), 370</i>
<code>get_nevpt2_correction()</code> (<i>ChemistryDriverPySCFEmbeddingRHF method), 1132</i>	<code>get_numeric_representation()</code> (<i>FermionState method), 1059</i>
<code>get_nevpt2_correction()</code> (<i>ChemistryDriverPySCFEmbeddingROHF method), 1140</i>	<code>get_numeric_representation()</code> (<i>GeneralAnsatz method), 332</i>
<code>get_nevpt2_correction()</code> (<i>ChemistryDriverPySCFEmbeddingROHF_UHF method), 1149</i>	<code>get_numeric_representation()</code> (<i>HamiltonianVariationalAnsatz method), 448</i>
<code>get_nevpt2_correction()</code> (<i>ChemistryDriverPySCFGammaRHF method), 1158</i>	<code>get_numeric_representation()</code> (<i>HardwareEfficientAnsatz method), 460</i>
<code>get_nevpt2_correction()</code> (<i>ChemistryDriverPySCFGammaROHF method), 1167</i>	<code>get_numeric_representation()</code> (<i>LayeredAnsatz method), 455</i>
<code>get_nevpt2_correction()</code> (<i>ChemistryDriverPySCFIintegrals method), 1174</i>	<code>get_numeric_representation()</code> (<i>MultiConfigurationAnsatz method), 415</i>
<code>get_nevpt2_correction()</code> (<i>ChemistryDriverPySCFMolecularRHF method), 1183</i>	<code>get_numeric_representation()</code> (<i>MultiConfigurationState method), 420</i>
<code>get_nevpt2_correction()</code> (<i>ChemistryDriverPySCFMolecularRHFQMMMCOSMO method), 1193</i>	<code>get_numeric_representation()</code> (<i>MultiConfigurationStateBox method), 425</i>
<code>get_nevpt2_correction()</code> (<i>ChemistryDriverPySCFMolecularROHF method), 1202</i>	<code>get_numeric_representation()</code> (<i>QubitState method), 1073</i>
<code>get_nevpt2_correction()</code> (<i>ChemistryDriverPySCFMolecularROHFQMMMCOSMO method), 1212</i>	<code>get_numeric_representation()</code> (<i>RealGeneralizedBasisRotationAnsatz method), 431</i>
<code>get_nevpt2_correction()</code> (<i>ChemistryDriverPySCFMolecularUHF method), 1220</i>	<code>get_numeric_representation()</code> (<i>RealRestrictedBasisRotationAnsatz method), 436</i>
<code>get_nevpt2_correction()</code> (<i>ChemistryDriverPySCFMolecularUHFQMMMCOSMO method), 1230</i>	<code>get_numeric_representation()</code> (<i>RealUnrestrictedBasisRotationAnsatz method), 442</i>
<code>get_noisy_backend()</code> (<i>in module inquanto.express), 600</i>	<code>get_numeric_representation()</code> (<i>State method), 1088</i>
<code>get_numeric_representation()</code> (<i>CircuitAnsatz method), 337</i>	<code>get_numeric_representation()</code> (<i>TrotterAnsatz method), 348</i>
<code>get_numeric_representation()</code> (<i>ComposedAnsatz method), 342</i>	<code>get_occupations()</code> (<i>RestrictedOneBodyRDM method), 796</i>
<code>get_numeric_representation()</code> (<i>FermionSpaceAnsatzChemicallyAwareUCCSD method), 376</i>	<code>get_one_body_rdm()</code> (<i>ChemistryDriverPySCFIintegrals method), 1175</i>
<code>get_numeric_representation()</code> (<i>FermionSpaceAnsatzUpCCGD method), 381</i>	<code>get_orb_irreps_dataframe()</code> (<i>FermionSpace method), 1029</i>
<code>get_numeric_representation()</code> (<i>FermionSpaceAnsatzUpCCGSD method), 387</i>	<code>get_orbital_coefficients()</code> (<i>ChemistryDriverPySCFEmbeddingGammaRHF method), 1113</i>
<code>get_numeric_representation()</code> (<i>FermionSpaceAnsatzUpCCGSDSinglet method), 392</i>	<code>get_orbital_coefficients()</code> (<i>ChemistryDriverPySCFEmbeddingGammaROHF_UHF method), 1123</i>
<code>get_numeric_representation()</code> (<i>Fermion</i>	<code>get_orbital_coefficients()</code> (<i>ChemistryDriverPySCFEmbeddingRHF method), 1132</i>

```

get_orbital_coefficients()      (ChemistryDriver-
    PySCFEmbeddingROHF method), 1141
get_orbital_coefficients()      (ChemistryDriver-
    PySCFEmbeddingROHF_UHF method), 1150
get_orbital_coefficients()      (ChemistryDriver-
    PySCFGammaRHF method), 1158
get_orbital_coefficients()      (ChemistryDriver-
    PySCFGammaROHF method), 1167
get_orbital_coefficients()      (ChemistryDriver-
    PySCFIntegrals method), 1175
get_orbital_coefficients()      (ChemistryDriver-
    PySCFMolecularRHF method), 1184
get_orbital_coefficients()      (ChemistryDriver-
    PySCFMolecularRHFQMMMCOSMO method), 1193
get_orbital_coefficients()      (ChemistryDriver-
    PySCFMolecularROHF method), 1202
get_orbital_coefficients()      (ChemistryDriver-
    PySCFMolecularROHFQMMMCOSMO method), 1212
get_orbital_coefficients()      (ChemistryDriver-
    PySCFMolecularUHF method), 1220
get_orbital_coefficients()      (ChemistryDriver-
    PySCFMolecularUHFQMMMCOSMO method), 1230
get_overlap_computables()      (SCEOMMatrixCom-
    putable method), 564
get_phaseless_qubit_state()     (ProjectiveMeasure-
    ments method), 1002
get_rdm1_ccsd()                (ChemistryDriverPySCFEmbed-
    dingGammaRHF method), 1113
get_rdm1_ccsd()                (ChemistryDriverPySCFEmbed-
    dingGammaROHF_UHF method), 1123
get_rdm1_ccsd()                (ChemistryDriverPySCFEmbeddin-
    gRHF method), 1132
get_rdm1_ccsd()                (ChemistryDriverPySCFEmbeddin-
    gROHF method), 1141
get_rdm1_ccsd()                (ChemistryDriverPySCFEmbeddin-
    gROHF_UHF method), 1150
get_rdm1_ccsd()                (ChemistryDriverPySCFGammaRHF
    method), 1158
get_rdm1_ccsd()                (ChemistryDriverPySCFGam-
    maROHF method), 1167
get_rdm1_ccsd()                (ChemistryDriverPySCFIntegrals
    method), 1175
get_rdm1_ccsd()                (ChemistryDriverPySCFMolecular-
    RHF method), 1184
get_rdm1_ccsd()                (ChemistryDriverPySCFMolecular-
    RHFQMMMCOSMO method), 1193
get_rdm1_ccsd()                (ChemistryDriverPySCFMolecular-
    ROHF method), 1202
get_rdm1_ccsd()                (ChemistryDriverPySCFMolecular-
    ROHFQMMMCOSMO method), 1212
get_rdm1_ccsd()                (ChemistryDriverPySCFMolecu-
    larUHF method), 1221
get_rdm1_ccsd()                (ChemistryDriverPySCFMolecu-
    larUHFQMMMCOSMO method), 1230
get_rdm2_ccsd()                (ChemistryDriverPySCFEmbed-
    dingGammaRHF method), 1114
get_rdm2_ccsd()                (ChemistryDriverPySCFEmbed-
    dingGammaROHF_UHF method), 1123
get_rdm2_ccsd()                (ChemistryDriverPySCFEmbeddin-
    gRHF method), 1132
get_rdm2_ccsd()                (ChemistryDriverPySCFEmbeddin-
    gROHF method), 1141
get_rdm2_ccsd()                (ChemistryDriverPySCFEmbeddin-
    gROHF_UHF method), 1150
get_rdm2_ccsd()                (ChemistryDriverPySCFGammaRHF
    method), 1159
get_rdm2_ccsd()                (ChemistryDriverPySCFGam-
    maROHF method), 1168
get_rdm2_ccsd()                (ChemistryDriverPySCFIntegrals
    method), 1175
get_rdm2_ccsd()                (ChemistryDriverPySCFMolecular-
    RHF method), 1184
get_rdm2_ccsd()                (ChemistryDriverPySCFMolecular-
    RHFQMMMCOSMO method), 1194
get_rdm2_ccsd()                (ChemistryDriverPySCFMolecular-
    ROHF method), 1202
get_rdm2_ccsd()                (ChemistryDriverPySCFMolecular-
    ROHFQMMMCOSMO method), 1212
get_rdm2_ccsd()                (ChemistryDriverPySCFMolecu-
    larUHF method), 1221
get_rdm2_ccsd()                (ChemistryDriverPySCFMolecu-
    larUHFQMMMCOSMO method), 1230
get_runner()                   (BackendStatevectorProtocol method), 965
get_runner()                   (ComputeUncompute method), 922
get_runner()                   (ComputeUncomputeFactorizedOverlap
    method), 960
get_runner()                   (DestructiveSwapTest method), 928
get_runner()                   (FactorizedOverlap method), 948
get_runner()                   (HadamardTest method), 916
get_runner()                   (HadamardTestDerivative method), 978
get_runner()                   (HadamardTestDerivativeOverlap
    method), 971
get_runner()                   (HadamardTestOverlap method), 941
get_runner()                   (PauliAveraging method), 909
get_runner()                   (PhaseShift method), 984
get_runner()                   (SparseStatevectorProtocol method), 964
get_runner()                   (SwapFactorizedOverlap method), 954
get_runner()                   (SwapTest method), 935
get_runner()                   (SymbolicProtocol method), 966
get_s2_computables()          (SCEOMMatrixComputable
    method), 564
get_shots()                    (ComputeUncompute method), 923
get_shots()                    (ComputeUncomputeFactorizedOverlap
    method), 961
get_shots()                    (DestructiveSwapTest method), 929

```

get_shots()	(FactorizedOverlap method),	948		
get_shots()	(HadamardTest method),	917		
get_shots()	(HadamardTestDerivative method),	978		
get_shots()	(HadamardTestDerivativeOverlap method),	971		
get_shots()	(HadamardTestOverlap method),	942		
get_shots()	(IterativePhaseEstimation method),	989		
get_shots()	(IterativePhaseEstimationQuantinuum method),	994		
get_shots()	(PauliAveraging method),	910		
get_shots()	(PhaseShift method),	984		
get_shots()	(ProjectiveMeasurements method),	1002		
get_shots()	(ProtocolList method),	1007		
get_shots()	(SwapFactorizedOverlap method),	955		
get_shots()	(SwapTest method),	935		
get_subsystem_driver()	(ChemistryDriver-PySCFEmbeddingGammaRHF method),	1114		
get_subsystem_driver()	(ChemistryDriver-PySCFEmbeddingGammaROHF_UHF method),	1123		
get_subsystem_driver()	(ChemistryDriver-PySCFEmbeddingRHF method),	1132		
get_subsystem_driver()	(ChemistryDriver-PySCFEmbeddingROHF method),	1141		
get_subsystem_driver()	(ChemistryDriver-PySCFEmbeddingROHF_UHF method),	1150		
get_subsystem_driver()	(ChemistryDriver-PySCFGammaRHF method),	1159		
get_subsystem_driver()	(ChemistryDriver-PySCFGammaROHF method),	1168		
get_subsystem_driver()	(ChemistryDriverPySCFIintegrals method),	1175		
get_subsystem_driver()	(ChemistryDriverPySCFMolecularRHF method),	1184		
get_subsystem_driver()	(ChemistryDriverPySCFMolecularRHFQmmmCOSMO method),	1194		
get_subsystem_driver()	(ChemistryDriverPySCFMolecularROHF method),	1202		
get_subsystem_driver()	(ChemistryDriverPySCFMolecularROHFQmmmCOSMO method),	1212		
get_subsystem_driver()	(ChemistryDriverPySCFMolecularUHF method),	1221		
get_subsystem_driver()	(ChemistryDriverPySCFMolecularUHFQmmmCOSMO method),	1231		
get_supported_point_group_dict()	(PointGroup static method),	1099		
get_symbol2irrep_dict()	(PointGroup static method),	1100		
get_symbolic_representation()	(CircuitAnsatz method),	337		
get_symbolic_representation()	(ComposedAnsatz method),	343		
get_symbolic_representation()	(FermionSpaceAnsatzChemicallyAwareUCCSD method),			
		376		
	get_symbolic_representation()		(FermionSpaceAnsatzkUpCCGD method),	381
	get_symbolic_representation()		(FermionSpaceAnsatzkUpCCGSD method),	387
	get_symbolic_representation()		(FermionSpaceAnsatzkUpCCGDSinglet method),	393
	get_symbolic_representation()		(FermionSpaceAnsatzUCCD method),	365
	get_symbolic_representation()		(FermionSpaceAnsatzUCCGD method),	398
	get_symbolic_representation()		(FermionSpaceAnsatzUCCGSD method),	404
	get_symbolic_representation()		(FermionSpaceAnsatzUCCSD method),	360
	get_symbolic_representation()		(FermionSpaceAnsatzUCCSDSinglet method),	409
	get_symbolic_representation()		(FermionSpaceStateExp method),	354
	get_symbolic_representation()		(FermionSpaceStateExpChemicallyAware method),	371
	get_symbolic_representation()		(FermionState method),	1059
	get_symbolic_representation()		(GeneralAnsatz method),	332
	get_symbolic_representation()		(HamiltonianVariationalAnsatz method),	449
	get_symbolic_representation()		(HardwareEfficientAnsatz method),	460
	get_symbolic_representation()		(LayeredAnsatz method),	455
	get_symbolic_representation()		(MultiConfigurationAnsatz method),	415
	get_symbolic_representation()		(MultiConfigurationState method),	421
	get_symbolic_representation()		(MultiConfigurationStateBox method),	426
	get_symbolic_representation()		(QubitState method),	1074
	get_symbolic_representation()		(RealGeneralizedBasisRotationAnsatz method),	431
	get_symbolic_representation()		(RealRestrictedBasisRotationAnsatz method),	437
	get_symbolic_representation()		(RealUnrestrictedBasisRotationAnsatz method),	442
	get_symbolic_representation()		(State method),	1088
	get_symbolic_representation()		(TrotterAnsatz method),	348
	get_system()		(ChemistryDriverPySCFEmbeddingGammaRHF method),	1114
	get_system()		(ChemistryDriverPySCFEmbeddingGammaROHF_UHF method),	1123
	get_system()		(ChemistryDriverPySCFEmbeddingRHF	

`method)`, 1132
`get_system()` (*ChemistryDriverPySCFEmbeddingROHF method*), 1141
`get_system()` (*ChemistryDriverPySCFEmbeddinggROHF_UHF method*), 1150
`get_system()` (*ChemistryDriverPySCFGammaRHF method*), 1159
`get_system()` (*ChemistryDriverPySCFGammaROHF method*), 1168
`get_system()` (*ChemistryDriverPySCFIintegrals method*), 1175
`get_system()` (*ChemistryDriverPySCFMolecularRHF method*), 1184
`get_system()` (*ChemistryDriverPySCFMolecularRHFQMMMCOSMO method*), 1194
`get_system()` (*ChemistryDriverPySCFMolecularROHF method*), 1203
`get_system()` (*ChemistryDriverPySCFMolecularROHFQMMMCOSMO method*), 1212
`get_system()` (*ChemistryDriverPySCFMolecularUHF method*), 1221
`get_system()` (*ChemistryDriverPySCFMolecularUHFQMMMCOSMO method*), 1231
`get_system()` (*ChemistryDriverPySCFMomentumRHF method*), 1235
`get_system()` (*ChemistryDriverPySCFMomentumROHF method*), 1238
`get_system()` (*DriverGeneralizedHubbard method*), 597
`get_system()` (*DriverHubbardDimer method*), 598
`get_system()` (*DriverIsing1D method*), 599
`get_system()` (*DriverIsing1DRing method*), 599
`get_system()` (*DriverIsingCustomConnectivity method*), 598
`get_system()` (*in module inquanto.express*), 600
`get_system_ao()` (*ChemistryDriverPySCFEmbeddingGammaRHF method*), 1114
`get_system_ao()` (*ChemistryDriverPySCFEmbeddingGammaROHF_UHF method*), 1124
`get_system_ao()` (*ChemistryDriverPySCFEmbeddinggRHF method*), 1132
`get_system_ao()` (*ChemistryDriverPySCFEmbeddinggROHF method*), 1141
`get_system_ao()` (*ChemistryDriverPySCFEmbeddinggROHF_UHF method*), 1150
`get_system_ao()` (*ChemistryDriverPySCFGammaRHF method*), 1159
`get_system_ao()` (*ChemistryDriverPySCFGammaROHF method*), 1168
`get_system_ao()` (*ChemistryDriverPySCFIintegrals method*), 1176
`get_system_ao()` (*ChemistryDriverPySCFMolecularRHF method*), 1185
`get_system_ao()` (*ChemistryDriverPySCFMolecularRHFQMMMCOSMO method*), 1194
`get_system_ao()` (*ChemistryDriverPySCFMolecular-ROHF method*), 1203
`get_system_ao()` (*ChemistryDriverPySCFMolecular-ROHFQMMMCOSMO method*), 1213
`get_system_ao()` (*ChemistryDriverPySCFMolecu-larUHF method*), 1222
`get_system_ao()` (*ChemistryDriverPySCFMolecu-larUHFQMMMCOSMO method*), 1231
`get_system_specification()` (*FCIDumpRestricted method*), 677
`get_sz_computables()` (*SCEOMMatrixComputable method*), 564
`get_zero_state_probability()` (*ProjectiveMea-surements method*), 1002
`get_zero_state_uncertainty()` (*ProjectiveMea-surements method*), 1002
`getfield()` (*ComputableNDArray method*), 505
`gram_schmidt()` (*OrbitalOptimizer static method*), 727
`gram_schmidt()` (*OrbitalTransformer static method*), 729

H

`HadamardTest` (*class in inquanto.protocols*), 912
`HadamardTestDerivative` (*class in in-quanto.protocols*), 973
`HadamardTestDerivativeOverlap` (*class in in-quanto.protocols*), 967
`HadamardTestOverlap` (*class in inquanto.protocols*), 937
`HamiltonianVariationalAnsatz` (*class in in-quanto.ansatzes*), 446
`hamming_weight` (*FermionStateString property*), 1068
`hamming_weight` (*QubitStateString property*), 1084
`hamming_weight` (*StateString property*), 1098
`HardwareEfficientAnsatz` (*class in in-quanto.ansatzes*), 457
`has_updated` (*AlgorithmBayesianQPE property*), 1266
`hashkey()` (*Cache method*), 579
`hashkey()` (*ProtocolCache method*), 1012
`hermitian_factorisation()` (*QubitOperator method*), 741
`hermitian_factorisation()` (*SymmetryOperator-Pauli method*), 854
`hermitian_part()` (*QubitOperator method*), 742
`hermitian_part()` (*SymmetryOperatorPauli method*), 855
`HoleGFComputable` (*class in in-quanto.computables.composite*), 535

I

`ICEBERG` (*CircuitEncoderQuantinuum attribute*), 1014
`IcebergOptions` (*class in inquanto.protocols*), 1013
`identity()` (*FermionOperator class method*), 690
`identity()` (*QubitOperator class method*), 743

identity() (*SymmetryOperatorFermionic* class method), 808
 identity() (*SymmetryOperatorPauli* class method), 855
 IGNORE (*QubitOperatorList.ExpandExponentialProductCoefficientsBehavior* attribute), 762
 IGNORE (*SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior* attribute), 875
 imag (*CacheLevels* attribute), 581
 imag (*CacheSizeUnit* attribute), 582
 imag (*ChemistryRestrictedIntegralOperator* property), 655
 imag (*ChemistryRestrictedIntegralOperatorCompact* property), 661
 imag (*CompactTwoBodyIntegralsS4* property), 673
 imag (*CompactTwoBodyIntegralsS8* property), 675
 imag (*CompilationLevel* attribute), 1015
 imag (*ComputableNDArray* attribute), 506
 imag (*CtrluStrat* attribute), 1016
 ImpurityDMETROHF (class in *in quanto.embeddings.dmet*), 593
 ImpurityDMETROHFFragment (class in *in quanto.embeddings.dmet*), 593
 ImpurityDMETROHFFragmentActive (class in *in quanto.embeddings.dmet*), 594
 ImpurityDMETROHFFragmentED (class in *in quanto.embeddings.dmet*), 595
 ImpurityDMETROHFFragmentPySCFActive (class in *inquanto.extensions.pyscf*), 1245
 ImpurityDMETROHFFragmentPySCFCCSD (class in *in quanto.extensions.pyscf*), 1246
 ImpurityDMETROHFFragmentPySCFFCI (class in *in quanto.extensions.pyscf*), 1247
 ImpurityDMETROHFFragmentPySCFMP2 (class in *in quanto.extensions.pyscf*), 1247
 ImpurityDMETROHFFragmentPySCFRHF (class in *in quanto.extensions.pyscf*), 1248
 ImpurityDMETROHFFragmentWithoutRDM (class in *in quanto.embeddings.dmet*), 596
 IN_EXPONENT (*QubitOperatorList.ExpandExponentialProductCoefficientsBehavior* attribute), 762
 IN_EXPONENT (*SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior* attribute), 875
 incompatibility_matrix() (*QubitOperatorList* method), 780
 incompatibility_matrix() (*SymmetryOperatorPauliFactorised* method), 893
 index() (*ComputableList* method), 490
 index() (*FermionOperatorList.CompressScalarsBehavior* method), 703
 index() (*FermionOperatorList.FactoryCoefficientsLocation* method), 708
 index() (*FermionOperator.TrotterizeCoefficientsLocation* method), 680
 index() (*FermionSpace* method), 1029
 index() (*FermionSpaceBrillouin* method), 1037
 index() (*FermionSpaceSupercell* method), 1046
 index() (*ParaFermionSpace* method), 1052
 index() (*QubitOperatorList.CompressScalarsBehavior* method), 758
 index() (*QubitOperatorList.ExpandExponentialProductCoefficientsBehavior* method), 763
 index() (*QubitOperatorList.FactoryCoefficientsLocation* method), 768
 index() (*QubitOperator.TrotterizeCoefficientsLocation* method), 731
 index() (*SymmetryOperatorFermionicFactorised.CompressScalarsBehavior* method), 822
 index() (*SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation* method), 827
 index() (*SymmetryOperatorFermionic.TrotterizeCoefficientsLocation* method), 799
 index() (*SymmetryOperatorPauliFactorised.CompressScalarsBehavior* method), 871
 index() (*SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior* method), 876
 index() (*SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation* method), 880
 index() (*SymmetryOperatorPauli.TrotterizeCoefficientsLocation* method), 844
 infer_num_spin_orbs() (*FermionOperator* method), 690
 infer_num_spin_orbs() (*FermionOperatorList* method), 716
 infer_num_spin_orbs() (*SymmetryOperatorFermionic* method), 808
 infer_num_spin_orbs() (*SymmetryOperatorFermionicFactorised* method), 835
 INNER (*FermionOperatorList.FactoryCoefficientsLocation* attribute), 707
 INNER (*FermionOperator.TrotterizeCoefficientsLocation* attribute), 679
 INNER (*QubitOperatorList.FactoryCoefficientsLocation* attribute), 767
 INNER (*QubitOperator.TrotterizeCoefficientsLocation* attribute), 730
 INNER (*SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation* attribute), 826

INNER (SymmetryOperator-
Fermionic.TrotterizeCoefficientsLocation attribute), 798

INNER (SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation attribute), 879

INNER (SymmetryOperatorPauli.TrotterizeCoefficientsLocation attribute), 843

inquito.computables.atomic module, 462

inquito.computables.composite module, 532

inquito.computables.primitive module, 485

InQuantoContext (class in `inquito.core`), 583

inquito.core module, 584

inquito.embeddings.dmet module, 585

inquito.express module, 600

inquito.extensions.nglview module, 1264

inquito.extensions.phayes module, 1266

inquito.extensions.pyscf module, 1104

inquito.extensions.pyscf.fmo module, 1258

inquito.minimizers module, 644

inquito.operators module, 652

inquito.spaces module, 1017

inquito.states module, 1055

inquito.symmetry module, 1099

insert() (ComputableList method), 490

IntegralType (class in `inquito.operators`), 726

irrep_direct_product() (PointGroup method), 1100

is_all_coeff_complex() (FermionOperator method), 690

is_all_coeff_complex() (FermionState method), 1059

is_all_coeff_complex() (QubitOperator method), 743

is_all_coeff_complex() (QubitState method), 1074

is_all_coeff_complex() (State method), 1088

is_all_coeff_complex() (SymmetryOperatorFermionic method), 809

is_all_coeff_complex() (SymmetryOperatorPauli method), 855

is_all_coeff_imag() (FermionOperator method), 690

is_all_coeff_imag() (FermionState method), 1059

is_all_coeff_imag() (QubitOperator method), 743

is_all_coeff_imag() (QubitState method), 1074

is_all_coeff_imag() (State method), 1089

is_all_coeff_imag() (SymmetryOperatorFermionic method), 809

is_all_coeff_imag() (SymmetryOperatorPauli method), 856

is_all_coeff_real() (FermionOperator method), 691

is_all_coeff_real() (FermionState method), 1059

is_all_coeff_real() (QubitOperator method), 743

is_all_coeff_real() (QubitState method), 1075

is_all_coeff_real() (State method), 1089

is_all_coeff_real() (SymmetryOperatorFermionic method), 809

is_all_coeff_real() (SymmetryOperatorPauli method), 856

is_all_coeff_symbolic() (FermionOperator method), 691

is_all_coeff_symbolic() (FermionState method), 1059

is_all_coeff_symbolic() (QubitOperator method), 743

is_all_coeff_symbolic() (QubitState method), 1075

is_all_coeff_symbolic() (State method), 1089

is_all_coeff_symbolic() (SymmetryOperatorFermionic method), 809

is_all_coeff_symbolic() (SymmetryOperatorPauli method), 856

is_antihermitian() (FermionOperator method), 691

is_antihermitian() (QubitOperator method), 744

is_antihermitian() (SymmetryOperatorFermionic method), 810

is_antihermitian() (SymmetryOperatorPauli method), 856

is_any_coeff_complex() (FermionOperator method), 691

is_any_coeff_complex() (FermionState method), 1060

is_any_coeff_complex() (QubitOperator method), 744

is_any_coeff_complex() (QubitState method), 1075

is_any_coeff_complex() (State method), 1089

is_any_coeff_complex() (SymmetryOperatorFermionic method), 810

is_any_coeff_complex() (SymmetryOperatorPauli method), 856

is_any_coeff_imag() (FermionOperator method), 691

is_any_coeff_imag() (FermionState method), 1060

is_any_coeff_imag() (QubitOperator method), 744

is_any_coeff_imag() (QubitState method), 1075

is_any_coeff_imag() (*State method*), 1089
 is_any_coeff_imag() (*SymmetryOperatorFermionic method*), 810
 is_any_coeff_imag() (*SymmetryOperatorPauli method*), 857
 is_any_coeff_real() (*FermionOperator method*), 692
 is_any_coeff_real() (*FermionState method*), 1060
 is_any_coeff_real() (*QubitOperator method*), 744
 is_any_coeff_real() (*QubitState method*), 1075
 is_any_coeff_real() (*State method*), 1090
 is_any_coeff_real() (*SymmetryOperatorFermionic method*), 810
 is_any_coeff_real() (*SymmetryOperatorPauli method*), 857
 is_any_coeff_symbolic() (*FermionOperator method*), 692
 is_any_coeff_symbolic() (*FermionState method*), 1060
 is_any_coeff_symbolic() (*QubitOperator method*), 744
 is_any_coeff_symbolic() (*QubitState method*), 1076
 is_any_coeff_symbolic() (*State method*), 1090
 is_any_coeff_symbolic() (*SymmetryOperatorFermionic method*), 810
 is_any_coeff_symbolic() (*SymmetryOperatorPauli method*), 857
 is_basis_state() (*FermionState method*), 1060
 is_basis_state() (*QubitState method*), 1076
 is_basis_state() (*State method*), 1090
 is_built (*ComputeUncompute property*), 923
 is_built (*ComputeUncomputeFactorizedOverlap property*), 961
 is_built (*DestructiveSwapTest property*), 929
 is_built (*FactorizedOverlap property*), 948
 is_built (*HadamardTest property*), 917
 is_built (*HadamardTestDerivative property*), 978
 is_built (*HadamardTestDerivativeOverlap property*), 971
 is_built (*HadamardTestOverlap property*), 942
 is_built (*IterativePhaseEstimation property*), 990
 is_built (*IterativePhaseEstimationQuantinuum property*), 994
 is_built (*IterativePhaseEstimationStatevector property*), 998
 is_built (*PauliAveraging property*), 910
 is_built (*PhaseShift property*), 985
 is_built (*ProjectiveMeasurements property*), 1002
 is_built (*SwapFactorizedOverlap property*), 955
 is_built (*SwapTest property*), 935
 is_commuting_operator() (*FermionOperator method*), 692
 is_commuting_operator() (*QubitOperator method*), 744
 is_commuting_operator() (*SymmetryOperatorFermionic method*), 811
 is_commuting_operator() (*SymmetryOperatorPauli method*), 857
 is_empty() (*FermionOperatorString method*), 725
 is_empty() (*SymmetryOperatorFermionicFactorised method*), 835
 is_empty() (*SymmetryOperatorPauliFactorised method*), 893
 is_hermitian() (*FermionOperator method*), 692
 is_hermitian() (*QubitOperator method*), 745
 is_hermitian() (*SymmetryOperatorFermionic method*), 811
 is_hermitian() (*SymmetryOperatorPauli method*), 857
 is_hermitian_coeff() (*QubitOperator static method*), 745
 is_hermitian_coeff() (*SymmetryOperatorPauli static method*), 857
 is_leaf() (*CommutatorComputable method*), 533
 is_leaf() (*ComputableFunction method*), 486
 is_leaf() (*ComputableInt method*), 488
 is_leaf() (*ComputableList method*), 490
 is_leaf() (*ComputableNDArray method*), 506
 is_leaf() (*ComputableNode method*), 528
 is_leaf() (*ComputableSingleChild method*), 530
 is_leaf() (*ExpectationValue method*), 463
 is_leaf() (*ExpectationValueBraDerivativeImag method*), 465
 is_leaf() (*ExpectationValueBraDerivativeReal method*), 467
 is_leaf() (*ExpectationValueDerivative method*), 469
 is_leaf() (*ExpectationValueKetDerivativeImag method*), 470
 is_leaf() (*ExpectationValueKetDerivativeReal method*), 472
 is_leaf() (*ExpectationValueNonHermitian method*), 474
 is_leaf() (*ExpectationValueSumComputable method*), 535
 is_leaf() (*HoleGFComputable method*), 536
 is_leaf() (*KrylovSubspaceComputable method*), 540
 is_leaf() (*LanczosCoefficientsComputable method*), 542
 is_leaf() (*LanczosMatrixComputable method*), 543
 is_leaf() (*ManyBodyGFCcomputable method*), 546
 is_leaf() (*MetricTensorImag method*), 476
 is_leaf() (*MetricTensorReal method*), 477
 is_leaf() (*NonOrthogonalMatricesComputable method*), 547
 is_leaf() (*Overlap method*), 479
 is_leaf() (*OverlapImag method*), 481
 is_leaf() (*OverlapMatrixComputable method*), 549
 is_leaf() (*OverlapReal method*), 483
 is_leaf() (*OverlapSquared method*), 484
 is_leaf() (*ParticleGFCComputable method*), 553
 is_leaf() (*PDM1234RealComputable method*), 551

is_leaf() (*QCM4Computable method*), 555
 is_leaf() (*QSEMatricesComputable method*), 556
 is_leaf() (*RDM1234RealComputable method*), 558
 is_leaf() (*RestrictedOneBodyRDMComputable method*),
 560
 is_leaf() (*RestrictedOneBodyRDMRealComputable
 method*), 562
 is_leaf() (*SCEOMMatrixComputable method*), 564
 is_leaf() (*SpinlessNBodyPDMArrayRealComputable
 method*), 566
 is_leaf() (*SpinlessNBodyRDMArrayRealComputable
 method*), 568
 is_leaf() (*UnrestrictedOneBodyRDMComputable
 method*), 570
 is_leaf() (*UnrestrictedOneBodyRDMRealComputable
 method*), 571
 is_normal_ordered() (*FermionOperator method*), 692
 is_normal_ordered() (*SymmetryOperatorFermionic
 method*), 811
 is_normalized() (*FermionOperator method*), 693
 is_normalized() (*FermionState method*), 1060
 is_normalized() (*QubitOperator method*), 745
 is_normalized() (*QubitState method*), 1076
 is_normalized() (*State method*), 1090
 is_normalized() (*SymmetryOperatorFermionic
 method*), 811
 is_normalized() (*SymmetryOperatorPauli
 method*),
 857
 is_numeric (*ComputeUncompute property*), 923
 is_numeric (*ComputeUncomputeFactorizedOverlap
 property*), 961
 is_numeric (*DestructiveSwapTest property*), 929
 is_numeric (*FactorizedOverlap property*), 949
 is_numeric (*HadamardTest property*), 917
 is_numeric (*HadamardTestDerivative property*), 978
 is_numeric (*HadamardTestDerivativeOverlap property*),
 971
 is_numeric (*HadamardTestOverlap property*), 942
 is_numeric (*IterativePhaseEstimation property*), 990
 is_numeric (*IterativePhaseEstimationQuantinuum prop-
 erty*), 994
 is_numeric (*PauliAveraging property*), 910
 is_numeric (*PhaseShift property*), 985
 is_numeric (*ProjectiveMeasurements property*), 1002
 is_numeric (*ProtocolList property*), 1007
 is_numeric (*SwapFactorizedOverlap property*), 955
 is_numeric (*SwapTest property*), 936
 is_openshell() (*PySCFChemistryRestrictedIntegralOp-
 erator method*), 1251
 is_operator_permutation_invariant() (*Fermion-
 SpaceSupercell static method*), 1046
 is_parallel_with() (*FermionOperator method*), 693
 is_parallel_with() (*FermionState method*), 1061
 is_parallel_with() (*QubitOperator method*), 745
 is_parallel_with() (*QubitState method*), 1076
 is_parallel_with() (*State method*), 1090
 is_parallel_with() (*SymmetryOperatorFermionic
 method*), 812
 is_parallel_with() (*SymmetryOperatorPauli
 method*), 858
 is_particle_conserving() (*FermionOperatorString
 method*), 725
 is_run (*ComputeUncompute property*), 923
 is_run (*ComputeUncomputeFactorizedOverlap property*),
 961
 is_run (*DestructiveSwapTest property*), 929
 is_run (*FactorizedOverlap property*), 949
 is_run (*HadamardTest property*), 917
 is_run (*HadamardTestDerivative property*), 979
 is_run (*HadamardTestDerivativeOverlap property*), 971
 is_run (*HadamardTestOverlap property*), 942
 is_run (*IterativePhaseEstimation property*), 990
 is_run (*IterativePhaseEstimationQuantinuum property*),
 994
 is_run (*IterativePhaseEstimationStatevector property*),
 998
 is_run (*PauliAveraging property*), 910
 is_run (*PhaseShift property*), 985
 is_run (*ProjectiveMeasurements property*), 1002
 is_run (*ProtocolList property*), 1007
 is_run (*SwapFactorizedOverlap property*), 955
 is_run (*SwapTest property*), 936
 is_self_inverse() (*FermionOperator method*), 693
 is_self_inverse() (*QubitOperator method*), 745
 is_self_inverse() (*SymmetryOperatorFermionic
 method*), 812
 is_self_inverse() (*SymmetryOperatorPauli method*),
 858
 is_symbolic (*ComputeUncompute property*), 923
 is_symbolic (*ComputeUncomputeFactorizedOverlap
 property*), 961
 is_symbolic (*DestructiveSwapTest property*), 929
 is_symbolic (*FactorizedOverlap property*), 949
 is_symbolic (*HadamardTest property*), 917
 is_symbolic (*HadamardTestDerivative property*), 979
 is_symbolic (*HadamardTestDerivativeOverlap prop-
 erty*), 971
 is_symbolic (*HadamardTestOverlap property*), 942
 is_symbolic (*IterativePhaseEstimation property*), 990
 is_symbolic (*IterativePhaseEstimationQuantinuum
 property*), 995
 is_symbolic (*PauliAveraging property*), 910
 is_symbolic (*PhaseShift property*), 985
 is_symbolic (*ProjectiveMeasurements property*), 1002
 is_symbolic (*ProtocolList property*), 1007
 is_symbolic (*SwapFactorizedOverlap property*), 955
 is_symbolic (*SwapTest property*), 936

`is_symmetry_of()` (*SymmetryOperatorFermionic method*), 812
`is_symmetry_of()` (*SymmetryOperatorFermionicFactorised method*), 835
`is_symmetry_of()` (*SymmetryOperatorPauli method*), 858
`is_symmetry_of()` (*SymmetryOperatorPauliFactorised method*), 893
`is_transf` (*AVAS property*), 1106
`is_transf` (*CASSCF property*), 1107
`is_two_body_number_conserving()` (*FermionOperator method*), 693
`is_two_body_number_conserving()` (*SymmetryOperatorFermionic method*), 812
`is_unit_1norm()` (*FermionOperator method*), 694
`is_unit_1norm()` (*FermionState method*), 1061
`is_unit_1norm()` (*QubitOperator method*), 745
`is_unit_1norm()` (*QubitState method*), 1076
`is_unit_1norm()` (*State method*), 1091
`is_unit_1norm()` (*SymmetryOperatorFermionic method*), 813
`is_unit_1norm()` (*SymmetryOperatorPauli method*), 858
`is_unit_2norm()` (*FermionOperator method*), 694
`is_unit_2norm()` (*FermionState method*), 1061
`is_unit_2norm()` (*QubitOperator method*), 746
`is_unit_2norm()` (*QubitState method*), 1076
`is_unit_2norm()` (*State method*), 1091
`is_unit_2norm()` (*SymmetryOperatorFermionic method*), 813
`is_unit_2norm()` (*SymmetryOperatorPauli method*), 858
`is_unit_norm()` (*FermionOperator method*), 694
`is_unit_norm()` (*FermionState method*), 1061
`is_unit_norm()` (*QubitOperator method*), 746
`is_unit_norm()` (*QubitState method*), 1077
`is_unit_norm()` (*State method*), 1091
`is_unit_norm()` (*SymmetryOperatorFermionic method*), 813
`is_unit_norm()` (*SymmetryOperatorPauli method*), 859
`is_unitary()` (*FermionOperator method*), 695
`is_unitary()` (*QubitOperator method*), 746
`is_unitary()` (*SymmetryOperatorFermionic method*), 813
`is_unitary()` (*SymmetryOperatorPauli method*), 859
`isalnum()` (*FermionOperatorList.CompressScalarsBehavior method*), 704
`isalnum()` (*FermionOperatorList.FactoryCoefficientsLocation method*), 708
`isalnum()` (*FermionOperator.TrotterizeCoefficientsLocation method*), 681
`isalnum()` (*QubitOperatorList.CompressScalarsBehavior method*), 758
`isalpha()` (*QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method*), 763
`isalpha()` (*QubitOperatorList.FactoryCoefficientsLocation method*), 768
`isalpha()` (*QubitOperator.TrotterizeCoefficientsLocation method*), 731
`isalpha()` (*SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method*), 822
`isalpha()` (*SymmetryOperatorFermionicFactoryCoefficientsLocation method*), 827
`isalpha()` (*SymmetryOperatorFermionic.TrotterizeCoefficientsLocation method*), 799
`isalpha()` (*SymmetryOperatorPauliFactorised.CompressScalarsBehavior method*), 871
`isalpha()` (*SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method*), 876
`isalpha()` (*SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method*), 881
`isalpha()` (*SymmetryOperatorPauli.TrotterizeCoefficientsLocation method*), 844
`isalpha()` (*FermionOperatorList.CompressScalarsBehavior method*), 704
`isalpha()` (*FermionOperatorList.FactoryCoefficientsLocation method*), 708
`isalpha()` (*FermionOperator.TrotterizeCoefficientsLocation method*), 681
`isalpha()` (*QubitOperatorList.CompressScalarsBehavior method*), 758
`isalpha()` (*QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method*), 763
`isalpha()` (*QubitOperatorList.FactoryCoefficientsLocation method*), 768
`isalpha()` (*QubitOperator.TrotterizeCoefficientsLocation method*), 731
`isalpha()` (*SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method*), 822
`isalpha()` (*SymmetryOperatorFermionicFactoryCoefficientsLocation method*), 827

<code>torised.FactoryCoefficientsLocation</code>	<code>method),</code>	<code>isascii()</code>	<code>(SymmetryOperator-</code>
<code>827</code>			<code>Pauli.TrotterizeCoefficientsLocation</code> <code>method),</code>
<code>isalpha()</code>	<code>(SymmetryOperator-</code>	<code>844</code>	<code>844</code>
<code>Fermionic.TrotterizeCoefficientsLocation</code>	<code>method),</code>	<code>isdecimal()</code>	<code>(FermionOper-</code>
<code>799</code>		<code>atorList.CompressScalarsBehavior</code>	<code>ator</code>
<code>isalpha()</code>	<code>(SymmetryOperatorPauliFac-</code>	<code>704</code>	<code>704</code>
<code>torised.CompressScalarsBehavior</code>	<code>method),</code>	<code>isdecimal()</code>	<code>(FermionOper-</code>
<code>871</code>		<code>atorList.FactoryCoefficientsLocation</code>	<code>ator</code>
<code>isalpha()</code>	<code>(SymmetryOperatorPauliFac-</code>	<code>709</code>	<code>709</code>
<code>torised.ExpandExponentialProductCoefficientsBehavior</code>	<code>method),</code>	<code>decimal()</code>	<code>(FermionOper-</code>
<code>876</code>		<code>ator.TrotterizeCoefficientsLocation</code>	<code>ator</code>
<code>isalpha()</code>	<code>(SymmetryOperatorPauliFac-</code>	<code>681</code>	<code>681</code>
<code>torised.FactoryCoefficientsLocation</code>	<code>method),</code>	<code>isdecimal()</code>	<code>(QubitOpera-</code>
<code>881</code>		<code>torList.CompressScalarsBehavior</code>	<code>method),</code>
<code>isalpha()</code>	<code>(SymmetryOperator-</code>	<code>758</code>	<code>758</code>
<code>Pauli.TrotterizeCoefficientsLocation</code>	<code>method),</code>	<code>isdecimal()</code>	<code>(QubitOpera-</code>
<code>844</code>		<code>torList.ExpandExponentialProductCoefficientsBehavior</code>	<code>method),</code>
<code>isascii()</code>	<code>(FermionOper-</code>	<code>763</code>	<code>763</code>
<code>atorList.CompressScalarsBehavior</code>	<code>method),</code>	<code>isdecimal()</code>	<code>(QubitOpera-</code>
<code>704</code>		<code>torList.FactoryCoefficientsLocation</code>	<code>method),</code>
<code>isascii()</code>	<code>(FermionOper-</code>	<code>768</code>	<code>768</code>
<code>atorList.FactoryCoefficientsLocation</code>	<code>method),</code>	<code>isdecimal()</code>	<code>(QubitOpera-</code>
<code>709</code>		<code>tor.TrotterizeCoefficientsLocation</code>	<code>tor</code>
<code>isascii()</code>	<code>(FermionOper-</code>	<code>732</code>	<code>732</code>
<code>ator.TrotterizeCoefficientsLocation</code>	<code>method),</code>	<code>isdecimal()</code>	<code>(SymmetryOperatorFermionicFac-</code>
<code>681</code>		<code>torised.CompressScalarsBehavior</code>	<code>torised</code>
<code>isascii()</code>	<code>(QubitOperatorList.CompressScalarsBehavior</code>	<code>823</code>	<code>823</code>
<code>method),</code>		<code>isdecimal()</code>	<code>(SymmetryOperatorFermionicFac-</code>
<code>758</code>		<code>torised.FactoryCoefficientsLocation</code>	<code>torised</code>
<code>isascii()</code>	<code>(QubitOperator-</code>	<code>827</code>	<code>827</code>
<code>torList.ExpandExponentialProductCoefficientsBehavior</code>		<code>isdecimal()</code>	<code>(SymmetryOperator-</code>
<code>method),</code>		<code>Fermionic.TrotterizeCoefficientsLocation</code>	<code>Fermionic</code>
<code>763</code>		<code>method),</code>	<code>TrotterizeCoefficientsLocation</code>
<code>isascii()</code>	<code>(QubitOpera-</code>	<code>800</code>	<code>method),</code>
<code>torList.FactoryCoefficientsLocation</code>			<code>method),</code>
<code>768</code>			<code>800</code>
<code>isascii()</code>	<code>(QubitOperator.TrotterizeCoefficientsLocation</code>		
<code>method),</code>			
<code>731</code>			
<code>isascii()</code>	<code>(SymmetryOperatorFermionicFac-</code>		
<code>torised.CompressScalarsBehavior</code>			
<code>823</code>			
<code>isascii()</code>	<code>(SymmetryOperatorFermionicFac-</code>		
<code>torised.FactoryCoefficientsLocation</code>			
<code>827</code>			
<code>isascii()</code>	<code>(SymmetryOperator-</code>		
<code>Fermionic.TrotterizeCoefficientsLocation</code>			
<code>method),</code>			
<code>799</code>			
<code>isascii()</code>	<code>(SymmetryOperatorPauliFac-</code>		
<code>torised.CompressScalarsBehavior</code>			
<code>871</code>			
<code>isascii()</code>	<code>(SymmetryOperatorPauliFac-</code>		
<code>torised.ExpandExponentialProductCoefficientsBehavior</code>			
<code>method),</code>			
<code>876</code>			
<code>isascii()</code>	<code>(SymmetryOperatorPauliFac-</code>		
<code>torised.FactoryCoefficientsLocation</code>			
<code>881</code>			
		<code>isdigit()</code>	<code>(FermionOper-</code>
		<code>atorList.CompressScalarsBehavior</code>	<code>ator</code>
		<code>704</code>	<code>704</code>
		<code>isdigit()</code>	<code>(FermionOper-</code>
		<code>torList.FactoryCoefficientsLocation</code>	<code>ator</code>
		<code>709</code>	<code>709</code>
		<code>isdigit()</code>	<code>(FermionOper-</code>
		<code>tor.TrotterizeCoefficientsLocation</code>	<code>ator</code>
		<code>681</code>	<code>681</code>

isdigit() (<i>QubitOperatorList.CompressScalarsBehavior method</i>), 758	823
isidentifier() (<i>QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method</i>), 763	isidentifier() (<i>SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method</i>), 828
isidentifier() (<i>QubitOperatorList.FactoryCoefficientsLocation method</i>), 768	isidentifier() (<i>SymmetryOperatorFermionic.TrotterizeCoefficientsLocation method</i>), 800
isidentifier() (<i>QubitOperator.TrotterizeCoefficientsLocation method</i>), 732	isidentifier() (<i>SymmetryOperatorPauliFactorised.CompressScalarsBehavior method</i>), 871
isidentifier() (<i>SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method</i>), 823	isidentifier() (<i>SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method</i>), 876
isidentifier() (<i>SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method</i>), 827	isidentifier() (<i>SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method</i>), 881
isidentifier() (<i>SymmetryOperatorFermionic.TrotterizeCoefficientsLocation method</i>), 800	isidentifier() (<i>SymmetryOperatorPauli.TrotterizeCoefficientsLocation method</i>), 844
isidentifier() (<i>SymmetryOperatorPauliFactorised.CompressScalarsBehavior method</i>), 871	islower() (<i>FermionOperatorList.CompressScalarsBehavior method</i>), 704
isidentifier() (<i>SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method</i>), 876	islower() (<i>FermionOperatorList.FactoryCoefficientsLocation method</i>), 709
isidentifier() (<i>SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method</i>), 881	islower() (<i>FermionOperator.TrotterizeCoefficientsLocation method</i>), 681
isidentifier() (<i>SymmetryOperatorPauli.TrotterizeCoefficientsLocation method</i>), 844	islower() (<i>QubitOperatorList.CompressScalarsBehavior method</i>), 758
isidentifier() (<i>FermionOperatorList.CompressScalarsBehavior method</i>), 704	islower() (<i>QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method</i>), 763
isidentifier() (<i>FermionOperatorList.FactoryCoefficientsLocation method</i>), 709	islower() (<i>QubitOperatorList.FactoryCoefficientsLocation method</i>), 768
isidentifier() (<i>FermionOperator.TrotterizeCoefficientsLocation method</i>), 681	islower() (<i>QubitOperator.TrotterizeCoefficientsLocation method</i>), 732
isidentifier() (<i>QubitOperatorList.CompressScalarsBehavior method</i>), 758	islower() (<i>SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method</i>), 823
isidentifier() (<i>QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method</i>), 763	islower() (<i>SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method</i>), 828
isidentifier() (<i>QubitOperatorList.FactoryCoefficientsLocation method</i>), 768	islower() (<i>SymmetryOperatorFermionic.TrotterizeCoefficientsLocation method</i>), 800
isidentifier() (<i>QubitOperator.TrotterizeCoefficientsLocation method</i>), 732	islower() (<i>SymmetryOperatorPauliFactorised.CompressScalarsBehavior method</i>), 871
isidentifier() (<i>SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method</i>), 876	islower() (<i>SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method</i>), 876

<i>torised.FactoryCoefficientsLocation</i>	<i>method),</i>	<i>tor.TrotterizeCoefficientsLocation</i>	<i>method),</i>
881		681	
<i>islower()</i>	<i>(SymmetryOperator-</i>	<i>isprintable()</i>	<i>(QubitOpera-</i>
	<i>Pauli.TrotterizeCoefficientsLocation</i>	<i>torList.CompressScalarsBehavior</i>	<i>method),</i>
844		759	
<i>isnumeric()</i>	<i>(FermionOpera-</i>	<i>isprintable()</i>	<i>(QubitOpera-</i>
	<i>torList.CompressScalarsBehavior</i>	<i>torList.ExpandExponentialProductCoefficientsBehavior</i>	<i>method),</i>
704		764	
<i>isnumeric()</i>	<i>(FermionOpera-</i>	<i>isprintable()</i>	<i>(QubitOpera-</i>
	<i>torList.FactoryCoefficientsLocation</i>	<i>torList.FactoryCoefficientsLocation</i>	<i>method),</i>
709		768	
<i>isnumeric()</i>	<i>(FermionOpera-</i>	<i>isprintable()</i>	<i>(QubitOpera-</i>
	<i>tor.TrotterizeCoefficientsLocation</i>	<i>tor.TrotterizeCoefficientsLocation</i>	<i>method),</i>
681		732	
<i>isnumeric()</i>	<i>(QubitOpera-</i>	<i>isprintable()</i>	<i>(SymmetryOperatorFermionicFac-</i>
	<i>torList.CompressScalarsBehavior</i>	<i>torised.CompressScalarsBehavior</i>	<i>method),</i>
759		823	
<i>isnumeric()</i>	<i>(QubitOpera-</i>	<i>isprintable()</i>	<i>(SymmetryOperatorFermionicFac-</i>
	<i>torList.ExpandExponentialProductCoefficientsBehavior</i>	<i>torised.FactoryCoefficientsLocation</i>	<i>method),</i>
763		828	
<i>isnumeric()</i>	<i>(QubitOpera-</i>	<i>isprintable()</i>	<i>(SymmetryOperator-</i>
	<i>torList.FactoryCoefficientsLocation</i>	<i>Fermionic.TrotterizeCoefficientsLocation</i>	<i>method),</i>
768		800	
<i>isnumeric()</i>	<i>(QubitOpera-</i>	<i>isprintable()</i>	<i>(SymmetryOperatorPauliFac-</i>
	<i>tor.TrotterizeCoefficientsLocation</i>	<i>torised.CompressScalarsBehavior</i>	<i>method),</i>
732		871	
<i>isnumeric()</i>	<i>(SymmetryOperatorFermionicFac-</i>	<i>isprintable()</i>	<i>(SymmetryOperatorPauliFac-</i>
	<i>torised.CompressScalarsBehavior</i>	<i>torised.ExpandExponentialProductCoefficientsBehavior</i>	<i>method),</i>
823		876	
<i>isnumeric()</i>	<i>(SymmetryOperatorFermionicFac-</i>	<i>isprintable()</i>	<i>(SymmetryOperatorPauliFac-</i>
	<i>torised.FactoryCoefficientsLocation</i>	<i>torised.FactoryCoefficientsLocation</i>	<i>method),</i>
828		881	
<i>isnumeric()</i>	<i>(SymmetryOperator-</i>	<i>isprintable()</i>	<i>(SymmetryOperator-</i>
	<i>Fermionic.TrotterizeCoefficientsLocation</i>	<i>Pauli.TrotterizeCoefficientsLocation</i>	<i>method),</i>
800		845	
<i>isnumeric()</i>	<i>(SymmetryOperatorPauliFac-</i>	<i>isspace()</i>	<i>(FermionOpera-</i>
	<i>torised.CompressScalarsBehavior</i>	<i>torList.CompressScalarsBehavior</i>	<i>method),</i>
871		704	
<i>isnumeric()</i>	<i>(SymmetryOperatorPauliFac-</i>	<i>isspace()</i>	<i>(FermionOpera-</i>
	<i>torised.ExpandExponentialProductCoefficientsBehavior</i>	<i>torList.FactoryCoefficientsLocation</i>	<i>method),</i>
876		709	
<i>isnumeric()</i>	<i>(SymmetryOperatorPauliFac-</i>	<i>isspace()</i>	<i>(FermionOpera-</i>
	<i>torised.FactoryCoefficientsLocation</i>	<i>tor.TrotterizeCoefficientsLocation</i>	<i>method),</i>
881		681	
<i>isnumeric()</i>	<i>(SymmetryOperator-</i>	<i>isspace()</i>	<i>(QubitOperatorList.CompressScalarsBehavior</i>
	<i>Pauli.TrotterizeCoefficientsLocation</i>		<i>method),</i>
845		759	
<i>isprintable()</i>	<i>(FermionOpera-</i>	<i>isspace()</i>	<i>(QubitOpera-</i>
	<i>torList.CompressScalarsBehavior</i>	<i>torList.ExpandExponentialProductCoefficientsBehavior</i>	<i>method),</i>
704		764	
<i>isprintable()</i>	<i>(FermionOpera-</i>	<i>isspace()</i>	<i>(QubitOpera-</i>
	<i>torList.FactoryCoefficientsLocation</i>	<i>torList.FactoryCoefficientsLocation</i>	<i>method),</i>
709		768	
<i>isprintable()</i>	<i>(FermionOpera-</i>	<i>isspace()</i>	<i>(QubitOperator.TrotterizeCoefficientsLocation</i>
			<i>method),</i>
			732

isspace()	(<i>SymmetryOperatorFermionicFac-torised.CompressScalarsBehavior</i> method),	method), 877
823		
isspace()	(<i>SymmetryOperatorFermionicFac-torised.FactoryCoefficientsLocation</i> method),	(<i>SymmetryOperatorPauliFac-torised.FactoryCoefficientsLocation</i> method), 881
828		
isspace()	(<i>SymmetryOperator-Fermionic.TrotterizeCoefficientsLocation</i> method),	(<i>SymmetryOperator-Pauli.TrotterizeCoefficientsLocation</i> method), 845
800		
isspace()	(<i>SymmetryOperatorPauliFac-torised.CompressScalarsBehavior</i> method),	(<i>FermionOper-atorList.CompressScalarsBehavior</i> method), 705
872		
isspace()	(<i>SymmetryOperatorPauliFac-torised.ExpandExponentialProductCoefficientsBehavior</i> method),	(<i>FermionOper-atorList.FactoryCoefficientsLocation</i> method), 709
877		
isspace()	(<i>SymmetryOperatorPauliFac-torised.FactoryCoefficientsLocation</i> method),	(<i>FermionOper-ator.TrotterizeCoefficientsLocation</i> method), 682
881		
isspace()	(<i>SymmetryOperator-Pauli.TrotterizeCoefficientsLocation</i> method),	(<i>QubitOper-atorList.CompressScalarsBehavior</i> method), 759
845		
istitle()	(<i>FermionOper-atorList.CompressScalarsBehavior</i> method),	(<i>QubitOper-atorList.ExpandExponentialProductCoefficientsBehavior</i> method), 764
704		
istitle()	(<i>FermionOper-atorList.FactoryCoefficientsLocation</i> method),	(<i>QubitOper-ator.TrotterizeCoefficientsLocation</i> method), 732
709		
istitle()	(<i>FermionOper-ator.TrotterizeCoefficientsLocation</i> method),	(<i>SymmetryOperatorFermionicFac-torised.CompressScalarsBehavior</i> method), 823
681		
istitle()	(<i>QubitOperatorList.CompressScalarsBehavior</i> method),	(<i>SymmetryOperatorFermionicFac-torised.FactoryCoefficientsLocation</i> method), 828
759		
istitle()	(<i>QubitOper-atorList.ExpandExponentialProductCoefficientsBehavior</i> method),	(<i>SymmetryOperator-Fermionic.TrotterizeCoefficientsLocation</i> method), 800
764		
istitle()	(<i>QubitOper-atorList.FactoryCoefficientsLocation</i> method),	(<i>SymmetryOperatorPauliFac-torised.CompressScalarsBehavior</i> method), 872
769		
istitle()	(<i>QubitOperator.TrotterizeCoefficientsLocation</i> method),	(<i>SymmetryOperatorPauliFac-torised.ExpandExponentialProductCoefficientsBehavior</i> method), 877
732		
istitle()	(<i>SymmetryOperatorFermionicFac-torised.CompressScalarsBehavior</i> method),	(<i>SymmetryOperatorPauliFac-torised.FactoryCoefficientsLocation</i> method), 882
823		
istitle()	(<i>SymmetryOperatorFermionicFac-torised.FactoryCoefficientsLocation</i> method),	(<i>SymmetryOperator-Pauli.TrotterizeCoefficientsLocation</i> method), 845
828		
istitle()	(<i>SymmetryOperator-Fermionic.TrotterizeCoefficientsLocation</i> method),	item() (<i>ComputableNDArray</i> method), 506
800		items() (<i>ChemistryRestrictedIntegralOperator</i> method), 656
istitle()	(<i>SymmetryOperatorPauliFac-torised.CompressScalarsBehavior</i> method),	items() (<i>ChemistryRestrictedIntegralOperatorCompact</i> method), 661
872		items() (<i>ChemistryUnrestrictedIntegralOperator</i> method), 666
istitle()	(<i>SymmetryOperatorPauliFac-torised.ExpandExponentialProductCoefficientsBehavior</i> method),	

items () (<i>ChemistryUnrestrictedIntegralOperatorCompact method</i>), 671	join ()	(<i>SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method</i>), 828
items () (<i>DoubleFactorizedHamiltonian method</i>), 676	join ()	(<i>SymmetryOperatorFermionic.TrotterizeCoefficientsLocation method</i>), 800
items () (<i>FermionOperator method</i>), 695	join ()	(<i>SymmetryOperatorPauliFactorised.CompressScalarsBehavior method</i>), 872
items () (<i>FermionOperatorList method</i>), 716	join ()	(<i>SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method</i>), 877
items () (<i>FermionOperatorString method</i>), 725	join ()	(<i>SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method</i>), 882
items () (<i>FermionState method</i>), 1061	join ()	(<i>SymmetryOperatorPauliFactorised.TrotterizeCoefficientsLocation method</i>), 845
items () (<i>PySCFChemistryRestrictedIntegralOperator method</i>), 1251		
items () (<i>PySCFChemistryUnrestrictedIntegralOperator method</i>), 1255		
items () (<i>QubitOperator method</i>), 746		
items () (<i>QubitOperatorList method</i>), 780		
items () (<i>QubitState method</i>), 1077		
items () (<i>State method</i>), 1091		
items () (<i>SymbolDict method</i>), 573		
items () (<i>SymmetryOperatorFermionic method</i>), 814		
items () (<i>SymmetryOperatorFermionicFactorised method</i>), 835		
items () (<i>SymmetryOperatorPauli method</i>), 859		
items () (<i>SymmetryOperatorPauliFactorised method</i>), 893		
itemset () (<i>ComputableNDArray method</i>), 507		
itemsize (<i>ComputableNDArray attribute</i>), 508		
IterativePhaseEstimation (class in <i>in quanto.protocols</i>), 987		
IterativePhaseEstimationQuantinuum (class in <i>in quanto.protocols</i>), 991		
IterativePhaseEstimationSingleCircuit (class in <i>in quanto.protocols</i>), 986		
IterativePhaseEstimationStatevector (class in <i>in quanto.protocols</i>), 996		
J	K	
join () (<i>AlgorithmBayesianQPE method</i>), 1266	k_ipqe (<i>IterativePhaseEstimation property</i>), 990	
join () (<i>AlgorithmInfoTheoryQPE method</i>), 324	k_ipqe (<i>IterativePhaseEstimationQuantinuum property</i>), 995	
join () (<i>AlgorithmKitaevQPE method</i>), 325	KB (<i>CacheSizeUnit attribute</i>), 581	
join () (<i>FermionOperatorList.CompressScalarsBehavior method</i>), 705	kernel (<i>ExpectationValue attribute</i>), 464	
join () (<i>FermionOperatorList.FactoryCoefficientsLocation method</i>), 709	kernel (<i>ExpectationValueBraDerivativeImag attribute</i>), 465	
join () (<i>FermionOperator.TrotterizeCoefficientsLocation method</i>), 682	kernel (<i>ExpectationValueBraDerivativeReal attribute</i>), 467	
join () (<i>QubitOperatorList.CompressScalarsBehavior method</i>), 759	kernel (<i>ExpectationValueDerivative attribute</i>), 469	
join () (<i>QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method</i>), 764	kernel (<i>ExpectationValueKetDerivativeImag attribute</i>), 470	
join () (<i>QubitOperatorList.FactoryCoefficientsLocation method</i>), 769	kernel (<i>ExpectationValueKetDerivativeReal attribute</i>), 472	
join () (<i>QubitOperator.TrotterizeCoefficientsLocation method</i>), 732	kernel (<i>ExpectationValueNonHermitian attribute</i>), 474	
join () (<i>SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method</i>), 823	kernel (<i>Overlap attribute</i>), 479	
	kernel (<i>OverlapImag attribute</i>), 481	
	kernel (<i>OverlapReal attribute</i>), 483	
	kernel (<i>OverlapSquared attribute</i>), 484	
	ket_state (<i>Overlap attribute</i>), 479	
	ket_state (<i>OverlapImag attribute</i>), 481	
	ket_state (<i>OverlapReal attribute</i>), 483	
	ket_state (<i>OverlapSquared attribute</i>), 485	
	key_from_str () (<i>FermionOperator static method</i>), 695	
	key_from_str () (<i>FermionState class method</i>), 1061	
	key_from_str () (<i>QubitOperator static method</i>), 746	
	key_from_str () (<i>QubitState class method</i>), 1077	
	key_from_str () (<i>State class method</i>), 1091	
	key_from_str () (<i>SymmetryOperatorFermionic static method</i>), 814	
	key_from_str () (<i>SymmetryOperatorPauli static method</i>), 859	
	keys () (<i>SymbolDict method</i>), 574	

KrylovSubspace	(class in <code>in quanto.computables.composite</code>),	537	in-	LanczosCoefficientsComputable	(class in <code>in quanto.computables.composite</code>),	540
KrylovSubspaceComputable	(class in <code>in quanto.computables.composite</code>),	539	in-	LanczosMatrixComputable	(class in <code>in quanto.computables.composite</code>),	542
L			launch()	(<code>ComputeUncompute</code> method),	923	
<code>L_BFGS_B_coarse</code>	(<code>OptimizationMethod</code> attribute),	649	launch()	(<code>ComputeUncomputeFactorizedOverlap</code> method),	961	
<code>L_BFGS_B_smooth</code>	(<code>OptimizationMethod</code> attribute),	649	launch()	(<code>DestructiveSwapTest</code> method),	930	
<code>label</code>	(<code>CommutatorComputable</code> attribute),	533	launch()	(<code>FactorizedOverlap</code> method),	949	
<code>label</code>	(<code>ComputableFunction</code> attribute),	486	launch()	(<code>HadamardTest</code> method),	917	
<code>label</code>	(<code>ComputableInt</code> attribute),	488	launch()	(<code>HadamardTestDerivative</code> method),	979	
<code>label</code>	(<code>ComputableList</code> attribute),	490	launch()	(<code>HadamardTestDerivativeOverlap</code> method),	972	
<code>label</code>	(<code>ComputableNDArray</code> attribute),	508	launch()	(<code>HadamardTestOverlap</code> method),	942	
<code>label</code>	(<code>ComputableNode</code> attribute),	529	launch()	(<code>IterativePhaseEstimation</code> method),	990	
<code>label</code>	(<code>ComputableSingleChild</code> attribute),	530	launch()	(<code>IterativePhaseEstimationQuantinuum</code> method),	995	
<code>label</code>	(<code>ExpectationValue</code> attribute),	464	launch()	(<code>IterativePhaseEstimationStatevector</code> method),	998	
<code>label</code>	(<code>ExpectationValueBraDerivativeImag</code> attribute),	465	launch()	(<code>PauliAveraging</code> method),	910	
<code>label</code>	(<code>ExpectationValueBraDerivativeReal</code> attribute),	467	launch()	(<code>PhaseShift</code> method),	985	
<code>label</code>	(<code>ExpectationValueDerivative</code> attribute),	469	launch()	(<code>ProjectiveMeasurements</code> method),	1002	
<code>label</code>	(<code>ExpectationValueKetDerivativeImag</code> attribute),	471	launch()	(<code>ProtocolList</code> method),	1007	
<code>label</code>	(<code>ExpectationValueKetDerivativeReal</code> attribute),	472	launch()	(<code>SwapFactorizedOverlap</code> method),	955	
<code>label</code>	(<code>ExpectationValueNonHermitian</code> attribute),	474	launch()	(<code>SwapTest</code> method),	936	
<code>label</code>	(<code>ExpectationValueSumComputable</code> attribute),	535	launch_experiment()	(<code>AlgorithmDeterministicQPE</code> method),	322	
<code>label</code>	(<code>HoleGFComputable</code> attribute),	536	LayeredAnsatz	(class in <code>inquanto.ansatzes</code>),	452	
<code>label</code>	(<code>KrylovSubspaceComputable</code> attribute),	540	level	(<code>Cache</code> property),	579	
<code>label</code>	(<code>LanczosCoefficientsComputable</code> attribute),	542	level	(<code>ProtocolCache</code> property),	1012	
<code>label</code>	(<code>LanczosMatrixComputable</code> attribute),	544	LIFETIME	(<code>CacheLevels</code> attribute),	579	
<code>label</code>	(<code>ManyBodyGFComputable</code> attribute),	546	linear_solver_scipy_linalg()	(<code>NaiveEulerIntegrator</code> static method),	648	
<code>label</code>	(<code>MetricTensorImag</code> attribute),	476	linear_solver_scipy_linalg()	(<code>ScipyIVPIntegrator</code> static method),	650	
<code>label</code>	(<code>MetricTensorReal</code> attribute),	477	linear_solver_scipy_linalg()	(<code>ScipyODEIntegrator</code> static method),	651	
<code>label</code>	(<code>NonOrthogonalMatricesComputable</code> attribute),	548	linear_solver_scipy_pinvh()	(<code>NaiveEulerIntegrator</code> static method),	648	
<code>label</code>	(<code>Overlap</code> attribute),	479	linear_solver_scipy_pinvh()	(<code>ScipyIVPIntegrator</code> static method),	650	
<code>label</code>	(<code>OverlapImag</code> attribute),	481	linear_solver_scipy_pinvh()	(<code>ScipyODEIntegrator</code> static method),	651	
<code>label</code>	(<code>OverlapMatrixComputable</code> attribute),	549	linear_solver_scipy_pinvh()	(<code>NaiveEulerIntegrator</code> static method),	648	
<code>label</code>	(<code>OverlapReal</code> attribute),	483	LinearInterpolatorPhaseEstimator	(class in <code>inquanto.protocols</code>),	987	
<code>label</code>	(<code>OverlapSquared</code> attribute),	485	list_class	(<code>FermionOperator</code> attribute),	695	
<code>label</code>	(<code>ParticleGFComputable</code> attribute),	553	list_class	(<code>QubitOperator</code> attribute),	746	
<code>label</code>	(<code>PDM1234RealComputable</code> attribute),	551	list_class	(<code>SymmetryOperatorFermionic</code> attribute),	814	
<code>label</code>	(<code>QCM4Computable</code> attribute),	555	list_class	(<code>SymmetryOperatorPauli</code> attribute),	859	
<code>label</code>	(<code>QSEMatricesComputable</code> attribute),	556	list_h5()	(in module <code>inquanto.express</code>),	600	
<code>label</code>	(<code>RDM1234RealComputable</code> attribute),	558	ljust()	(<code>FermionOperatorList.CompressScalarsBehavior</code> method),	705	
<code>label</code>	(<code>RestrictedOneBodyRDMComputable</code> attribute),	560	ljust()	(<code>FermionOperatorList.FactoryCoefficientsLocation</code> method),	710	
<code>label</code>	(<code>RestrictedOneBodyRDMRealComputable</code> attribute),	562		(<code>FermionOperatorList.FactoryCoefficientsLocation</code> method),	710	
<code>label</code>	(<code>SCEOMMatrixComputable</code> attribute),	564				
<code>label</code>	(<code>SpinlessNBodyPDMArrayRealComputable</code> attribute),	566				
<code>label</code>	(<code>SpinlessNBodyRDMArrayRealComputable</code> attribute),	568				
<code>label</code>	(<code>UnrestrictedOneBodyRDMComputable</code> attribute),	570				
<code>label</code>	(<code>UnrestrictedOneBodyRDMRealComputable</code> attribute),	571				

```

ljust() (FermionOperator.TrotterizeCoefficientsLocation
         method), 682
ljust() (QubitOperatorList.CompressScalarsBehavior
         method), 759
ljust() (QubitOperatorList.ExpandExponentialProductCoefficientsBehavior
         method), 764
ljust() (QubitOperatorList.FactoryCoefficientsLocation
         method), 769
ljust() (QubitOperator.TrotterizeCoefficientsLocation
         method), 732
ljust() (SymmetryOperatorFermionicFactorised.CompressScalarsBehavior
         method), 824
ljust() (SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation
         method), 828
ljust() (SymmetryOperatorFermionic.TrotterizeCoefficientsLocation
         method), 800
ljust() (SymmetryOperatorPauliFactorised.CompressScalarsBehavior
         method), 872
ljust() (SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior
         method), 877
ljust() (SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation
         method), 882
ljust() (SymmetryOperatorPauli.TrotterizeCoefficientsLocation
         method), 845
load() (ComputeUncompute class method), 924
load() (DestructiveSwapTest class method), 930
load() (FCIDumpRestricted method), 677
load() (HadamardTest class method), 917
load() (HadamardTestDerivativeOverlap class method),
       972
load() (PauliAveraging class method), 911
load() (ProjectiveMeasurements class method), 1003
load() (SwapTest class method), 936
load_csv() (GeometryMolecular method), 608
load_csv() (GeometryPeriodic method), 619
load_h5() (ChemistryRestrictedIntegralOperator class
          method), 656
load_h5() (ChemistryRestrictedIntegralOperatorCompact
          class method), 661
load_h5() (ChemistryUnrestrictedIntegralOperator class
          method), 666
load_h5() (ChemistryUnrestrictedIntegralOperatorCompact
          class method), 671
load_h5() (FermionSpace class method), 1030
load_h5() (FermionSpaceBrillouin class method), 1038
load_h5() (FermionSpaceSupercell class method), 1046
load_h5() (FermionState class method), 1062
load_h5() (FermionStateString class method), 1068
load_h5() (in module inquanto.express), 600
load_h5() (ParaFermionSpace class method), 1052
load_h5() (PySCFChemistryRestrictedIntegralOperator
          Behaviors method), 1251
load_h5() (PySCFChemistryUnrestrictedIntegralOperator
          class method), 1256
load_h5() (QubitSpace class method), 1054
load_h5() (QubitState class method), 1077
load_h5() (QubitStateString class method), 1084
load_h5() (RestrictedOneBodyRDM class method), 796
load_h5() (RestrictedTwoBodyRDM class method), 797
load_h5() (State class method), 1091
load_h5() (StateString class method), 1098
load_h5() (UnrestrictedOneBodyRDM class method), 902
load_h5() (UnrestrictedTwoBodyRDM class method),
       904
load_json() (GeometryMolecular method), 608
load_json() (GeometryPeriodic method), 619
load_xyz() (GeometryMolecular class method), 608
load_xyz() (GeometryPeriodic class method), 620
load_zmatrix() (GeometryMolecular method), 608
loads() (ComputeUncompute class method), 924
loads() (DestructiveSwapTest class method), 930
loads() (HadamardTest class method), 918
loads() (HadamardTestDerivativeOverlap class method),
       972
loads() (PauliAveraging class method), 911
loads() (ProjectiveMeasurements class method), 1003
loads() (SwapTest class method), 936
LOGICAL (CompilationLevel attribute), 1014
lower() (FermionOperatorList.CompressScalarsBehavior
          method), 705
lower() (FermionOperatorList.FactoryCoefficientsLocation
          method), 710
lower() (FermionOperator.TrotterizeCoefficientsLocation
          method), 682
lower() (QubitOperatorList.CompressScalarsBehavior
          method), 759
lower() (QubitOperatorList.ExpandExponentialProductCoefficientsBehavior
          method), 764
lower() (QubitOperatorList.FactoryCoefficientsLocation
          method), 769
lower() (QubitOperator.TrotterizeCoefficientsLocation
          method), 733
lower() (SymmetryOperatorFermionicFactorised.CompressScalarsBehavior
          method), 824
lower() (SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation
          method), 828
lower() (SymmetryOperatorFermionic.TrotterizeCoefficientsLocation
          method)

```

method), 801

lower() (SymmetryOperatorPauliFactorised.CompressScalarsBehavior method), 872

lower() (SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method), 877

lower() (SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method), 882

lower() (SymmetryOperatorPauli.TrotterizeCoefficientsLocation method), 845

lowest_eigenvalue() (KrylovSubspace method), 538

lstrip() (FermionOperatorList.CompressScalarsBehavior method), 705

lstrip() (FermionOperatorList.FactoryCoefficientsLocation method), 710

lstrip() (FermionOperator.TrotterizeCoefficientsLocation method), 682

lstrip() (QubitOperatorList.CompressScalarsBehavior method), 759

lstrip() (QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method), 764

lstrip() (QubitOperatorList.FactoryCoefficientsLocation method), 769

lstrip() (QubitOperator.TrotterizeCoefficientsLocation method), 733

lstrip() (SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method), 824

lstrip() (SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method), 828

lstrip() (SymmetryOperatorFermionic.TrotterizeCoefficientsLocation method), 801

lstrip() (SymmetryOperatorPauliFactorised.CompressScalarsBehavior method), 872

lstrip() (SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method), 877

lstrip() (SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method), 882

lstrip() (SymmetryOperatorPauli.TrotterizeCoefficientsLocation method), 845

M

make_actives_contiguous() (ChemistryDriver-PySCFEmbeddingGammaRHF method), 1115

make_actives_contiguous() (ChemistryDriver-PySCFEmbeddingGammaROHF_UHF method), 1124

make_actives_contiguous() (ChemistryDriver-PySCFEmbeddingRHF method), 1133

make_actives_contiguous() (ChemistryDriver-PySCFEmbeddingROHF method), 1142

make_actives_contiguous() (ChemistryDriver-PySCFEmbeddingROHF_UHF method), 1151

make_actives_contiguous() (ChemistryDriver-PySCFGammaRHF method), 1160

make_actives_contiguous() (ChemistryDriver-PySCFGammaROHF method), 1169

make_actives_contiguous() (ChemistryDriver-PySCFIntegrals method), 1176

make_actives_contiguous() (ChemistryDriver-PySCFMolecularRHF method), 1185

make_actives_contiguous() (ChemistryDriver-PySCFMolecularRHFQMMMCOSMO method), 1195

make_actives_contiguous() (ChemistryDriver-PySCFMolecularROHF method), 1203

make_actives_contiguous() (ChemistryDriver-PySCFMolecularROHFQMMMCOSMO method), 1213

make_actives_contiguous() (ChemistryDriver-PySCFMolecularUHF method), 1222

make_actives_contiguous() (ChemistryDriver-PySCFMolecularUHFQMMMCOSMO method), 1232

make_hashable() (CircuitAnsatz method), 337

make_hashable() (ComposedAnsatz method), 343

make_hashable() (FermionOperator method), 695

make_hashable() (FermionOperatorList method), 716

make_hashable() (FermionSpaceAnsatzChemicallyAwareUCCSD method), 376

make_hashable() (FermionSpaceAnsatzkUpCCGD method), 382

make_hashable() (FermionSpaceAnsatzkUpCCGSD method), 387

make_hashable() (FermionSpaceAnsatzkUpCCGDSinglet method), 393

make_hashable() (FermionSpaceAnsatzUCCD method), 366

make_hashable() (FermionSpaceAnsatzUCCGD method), 399

make_hashable() (FermionSpaceAnsatzUCCGSD method), 404

make_hashable() (FermionSpaceAnsatzUCCSD method), 360

make_hashable() (FermionSpaceAnsatzUCCSDSinglet

method), 410
make_hashable() (FermionSpaceStateExp method), 355
make_hashable() (FermionSpaceStateExpChemicallyAware method), 371
make_hashable() (FermionState method), 1062
make_hashable() (GeneralAnsatz method), 332
make_hashable() (HamiltonianVariationalAnsatz method), 449
make_hashable() (HardwareEfficientAnsatz method), 460
make_hashable() (LayeredAnsatz method), 455
make_hashable() (MultiConfigurationAnsatz method), 416
make_hashable() (MultiConfigurationState method), 421
make_hashable() (MultiConfigurationStateBox method), 426
make_hashable() (QubitOperator method), 746
make_hashable() (QubitOperatorList method), 780
make_hashable() (QubitState method), 1077
make_hashable() (RealGeneralizedBasisRotationAnsatz method), 431
make_hashable() (RealRestrictedBasisRotationAnsatz method), 437
make_hashable() (RealUnrestrictedBasisRotationAnsatz method), 442
make_hashable() (State method), 1091
make_hashable() (SymbolDict method), 574
make_hashable() (SymbolSet method), 577
make_hashable() (SymmetryOperatorFermionic method), 814
make_hashable() (SymmetryOperatorFermionicFactorised method), 835
make_hashable() (SymmetryOperatorPauli method), 859
make_hashable() (SymmetryOperatorPauliFactorised method), 893
make_hashable() (TrotterAnsatz method), 349
maketrans() (FermionOperatorList.CompressScalarsBehavior static method), 705
maketrans() (FermionOperatorList.FactoryCoefficientsLocation static method), 710
maketrans() (FermionOperator.TrotterizeCoefficientsLocation static method), 682
maketrans() (QubitOperatorList.CompressScalarsBehavior static method), 759
maketrans() (QubitOperatorList.ExpandExponentialProductCoefficientsBehavior static method), 764
maketrans() (QubitOperator
torList.FactoryCoefficientsLocation static method), 769
maketrans() (QubitOperator.TrotterizeCoefficientsLocation static method), 733
maketrans() (SymmetryOperatorFermionicFactorised.CompressScalarsBehavior static method), 824
maketrans() (SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation static method), 829
maketrans() (SymmetryOperatorFermionic.TrotterizeCoefficientsLocation static method), 801
maketrans() (SymmetryOperatorPauliFactorised.CompressScalarsBehavior static method), 872
maketrans() (SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior static method), 877
maketrans() (SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation static method), 882
maketrans() (SymmetryOperatorPauli.TrotterizeCoefficientsLocation static method), 845
ManyBodyGFComputable (class in inquanto.computables.composite), 544
map (QubitOperatorString property), 791
map() (FermionOperator method), 695
map() (FermionOperatorList method), 716
map() (FermionState method), 1062
map() (QubitOperator method), 747
map() (QubitOperatorList method), 780
map() (QubitState method), 1077
map() (State method), 1092
map() (SymmetryOperatorFermionic method), 814
map() (SymmetryOperatorFermionicFactorised method), 835
map() (SymmetryOperatorPauli method), 859
map() (SymmetryOperatorPauliFactorised method), 893
map_variables_to_rotation_matrix() (OrbitalOptimizer method), 727
map_variables_to_skew_matrix() (OrbitalOptimizer method), 727
matrix_to_dict() (in module inquanto.core), 584
max() (ComputableNDArray method), 508
max_mem_size (Cache property), 579
max_mem_size (ProtocolCache property), 1012
MB (CacheSizeUnit attribute), 581
mean() (ComputableNDArray method), 508
mean_field_rdm2() (RestrictedOneBodyRDM method), 796
mean_field_rdm2() (UnrestrictedOneBodyRDM

method), 903

MeasurementPluralityPhaseEstimator (class in `in quanto.protocols`), 987

mem_size (`Cache` property), 579

mem_size (`ProtocolCache` property), 1012

method (`MinimizerScipy` property), 647

MetricTensorImag (class in `in quanto.computables.atomic`), 474

MetricTensorReal (class in `in quanto.computables.atomic`), 476

mf_energy (`ChemistryDriverPySCFEmbeddingGammaRHF` property), 1115

mf_energy (`ChemistryDriverPySCFEmbeddingGammaRHF_UHF` property), 1124

mf_energy (`ChemistryDriverPySCFEmbeddingRHF` property), 1133

mf_energy (`ChemistryDriverPySCFEmbeddingROHF` property), 1142

mf_energy (`ChemistryDriverPySCFEmbeddinggROHF_UHF` property), 1151

mf_energy (`ChemistryDriverPySCFGammaRHF` property), 1160

mf_energy (`ChemistryDriverPySCFGammaROHF` property), 1169

mf_energy (`ChemistryDriverPySCFIintegrals` property), 1176

mf_energy (`ChemistryDriverPySCFMolecularRHF` property), 1185

mf_energy (`ChemistryDriverPySCFMolecularRHFQMMMCOSMO` property), 1195

mf_energy (`ChemistryDriverPySCFMolecularROHF` property), 1204

mf_energy (`ChemistryDriverPySCFMolecularROHFQMMCOSMO` property), 1204

mf_energy (`ChemistryDriverPySCFMolecularUHF` property), 1222

mf_energy (`ChemistryDriverPySCFMolecularUHFQMMCOSMO` property), 1232

mf_energy (`ChemistryDriverPySCFMomentumRHF` property), 1235

mf_energy (`ChemistryDriverPySCFMomentumROHF` property), 1238

mf_type (`ChemistryDriverPySCFEmbeddingGammaRHF` property), 1115

mf_type (`ChemistryDriverPySCFEmbeddingGammaRHF_UHF` property), 1124

mf_type (`ChemistryDriverPySCFEmbeddingRHF` property), 1133

mf_type (`ChemistryDriverPySCFEmbeddingROHF` property), 1142

mf_type (`ChemistryDriverPySCFEmbeddingROHF_UHF` property), 1151

mf_type (`ChemistryDriverPySCFGammaRHF` property), 1160

mf_type (`ChemistryDriverPySCFGammaROHF` property), 1169

mf_type (`ChemistryDriverPySCFIintegrals` property), 1176

mf_type (`ChemistryDriverPySCFMolecularRHF` property), 1185

mf_type (`ChemistryDriverPySCFMolecularRHFQMMCOSMO` property), 1195

mf_type (`ChemistryDriverPySCFMolecularROHF` property), 1204

mf_type (`ChemistryDriverPySCFMolecularROHFQMMCOSMO` property), 1213

mf_type (`ChemistryDriverPySCFMolecularUHF` property), 1222

mf_type (`ChemistryDriverPySCFMolecularUHFQMMCOSMO` property), 1232

mf_type (`ChemistryDriverPySCFMomentumRHF` property), 1235

mf_type (`ChemistryDriverPySCFMomentumROHF` property), 1238

min() (`ComputableNDArray` method), 508

mini_character_table() (`PointGroup` class method), 1100

minimize() (`MinimizerRotosolve` method), 644

minimize() (`MinimizerScipy` method), 647

minimize() (`MinimizerSGD` method), 645

minimize() (`MinimizerSPSA` method), 645

MinimizerRotosolve (class in `inquanto.minimizers`), 644

MinimizerScipy (class in `inquanto.minimizers`), 646

MinimizerSGD (class in `inquanto.minimizers`), 644

MinimizerSPSA (class in `inquanto.minimizers`), 645

MIXED (`FermionOperator.TrotterizeCoefficientsLocation` attribute), 679

MIXED (`QubitOperator.TrotterizeCoefficientsLocation` attribute), 730

MIXED (`SymmetryOperatorFermionic.TrotterizeCoefficientsLocation` attribute), 798

MIXED (`SymmetryOperatorPauli.TrotterizeCoefficientsLocation` attribute), 843

mo_coeff (`ChemistryDriverPySCFIintegrals` property), 1176

mode_class (`FermionStateString` attribute), 1069

mode_class (`QubitStateString` attribute), 1084

mode_class (`StateString` attribute), 1098

modify_bond_angle() (`GeometryMolecular` method), 608

modify_bond_angle() (`GeometryPeriodic` method), 620

modify_bond_angle_by_group() (`GeometryMolecular` method), 609

modify_bond_angle_by_group() (`GeometryPeriodic` method), 620

modify_bond_length() (`GeometryMolecular` method),

609
modify_bond_length() (*GeometryPeriodic method*), 620
modify_bond_length_by_group() (*GeometryMolecular method*), 609
modify_bond_length_by_group() (*GeometryPeriodic method*), 621
modify_dihedral_angle() (*GeometryMolecular method*), 610
modify_dihedral_angle() (*GeometryPeriodic method*), 621
modify_dihedral_angle_by_group() (*GeometryMolecular method*), 610
modify_dihedral_angle_by_group() (*GeometryPeriodic method*), 621
module
 inquanto.computables.atomic, 462
 inquanto.computables.composite, 532
 inquanto.computables.primitive, 485
 inquanto.core, 584
 inquanto.embeddings.dmet, 585
 inquanto.express, 600
 inquanto.extensions.nglview, 1264
 inquanto.extensions.phayes, 1266
 inquanto.extensions.pyscf, 1104
 inquanto.extensions.pyscf.fmo, 1258
 inquanto.minimizers, 644
 inquanto.operators, 652
 inquanto.spaces, 1017
 inquanto.states, 1055
 inquanto.symmetry, 1099
moments() (*KrylovSubspace method*), 538
MultiConfigurationAnsatz (*class*) *in* *in-*
 quanto.ansatzes, 412
MultiConfigurationState (*class*) *in* *in-*
 quanto.ansatzes, 417
MultiConfigurationStateBox (*class*) *in* *in-*
 quanto.ansatzes, 423

N

n_circuit (*ComputeUncompute property*), 924
n_circuit (*ComputeUncomputeFactorizedOverlap property*), 961
n_circuit (*DestructiveSwapTest property*), 930
n_circuit (*FactorizedOverlap property*), 949
n_circuit (*HadamardTest property*), 918
n_circuit (*HadamardTestDerivative property*), 979
n_circuit (*HadamardTestDerivativeOverlap property*), 972
n_circuit (*HadamardTestOverlap property*), 943
n_circuit (*IterativePhaseEstimation property*), 990
n_circuit (*IterativePhaseEstimationQuantinuum property*), 995
n_circuit (*PauliAveraging property*), 911
n_circuit (*PhaseShift property*), 985
n_circuit (*ProjectiveMeasurements property*), 1003
n_circuit (*ProtocolList property*), 1007
n_circuit (*SwapFactorizedOverlap property*), 955
n_circuit (*SwapTest property*), 937
n_electron (*ChemistryDriverPySCFEmbeddingGammaRHF property*), 1115
n_electron (*ChemistryDriverPySCFEmbeddingGammaROHF_UHF property*), 1124
n_electron (*ChemistryDriverPySCFEmbeddingRHF property*), 1133
n_electron (*ChemistryDriverPySCFEmbeddingROHF property*), 1142
n_electron (*ChemistryDriverPySCFEmbeddingROHF_UHF property*), 1151
n_electron (*ChemistryDriverPySCFGammaRHF property*), 1160
n_electron (*ChemistryDriverPySCFGammaROHF property*), 1169
n_electron (*ChemistryDriverPySCFIntegrals property*), 1176
n_electron (*ChemistryDriverPySCFMolecularRHF property*), 1185
n_electron (*ChemistryDriverPySCFMolecularRHFQMMMCOSMO property*), 1195
n_electron (*ChemistryDriverPySCFMolecularROHF property*), 1204
n_electron (*ChemistryDriverPySCFMolecularROHFQMMCOSMO property*), 1213
n_electron (*ChemistryDriverPySCFMolecularUHF property*), 1222
n_electron (*ChemistryDriverPySCFMolecularUHFQMMCOSMO property*), 1232
n_electron (*ChemistryDriverPySCFMomentumRHF property*), 1235
n_electron (*ChemistryDriverPySCFMomentumROHF property*), 1238
n_electron (*DriverGeneralizedHubbard property*), 597
n_electron (*DriverHubbardDimer property*), 598
n_kp (*ChemistryDriverPySCFMomentumRHF property*), 1235
n_kp (*ChemistryDriverPySCFMomentumROHF property*), 1238
n_kp (*FermionSpaceBrillouin property*), 1038
n_ones() (*ParaFermionSpace method*), 1052
n_orb (*ChemistryDriverPySCFEmbeddingGammaRHF property*), 1115
n_orb (*ChemistryDriverPySCFEmbeddingGammaROHF_UHF property*), 1124
n_orb (*ChemistryDriverPySCFEmbeddingRHF property*), 1133
n_orb (*ChemistryDriverPySCFEmbeddingROHF property*), 1142
n_orb (*ChemistryDriverPySCFEmbeddingROHF_UHF*)

n_orb (*property*), 1151
 n_orb (*ChemistryDriverPySCFGammaRHF property*), 1160
 n_orb (*ChemistryDriverPySCFGammaROHF property*), 1169
 n_orb (*ChemistryDriverPySCFIintegrals property*), 1177
 n_orb (*ChemistryDriverPySCFMolecularRHF property*), 1185
 n_orb (*ChemistryDriverPySCFMolecularRHFQMMM-COSMO property*), 1195
 n_orb (*ChemistryDriverPySCFMolecularROHF property*), 1204
 n_orb (*ChemistryDriverPySCFMolecularROHFQMMM-COSMO property*), 1213
 n_orb (*ChemistryDriverPySCFMolecularUHF property*), 1222
 n_orb (*ChemistryDriverPySCFMolecularUHFQMMM-COSMO property*), 1232
 n_orb (*ChemistryDriverPySCFMomentumRHF property*), 1235
 n_orb (*ChemistryDriverPySCFMomentumROHF property*), 1238
 n_orb (*DriverGeneralizedHubbard property*), 597
 n_orb (*DriverHubbardDimer property*), 598
 n_orb (*FermionSpace property*), 1030
 n_orb (*ParaFermionSpace property*), 1052
 n_orb () (*RestrictedOneBodyRDM method*), 796
 n_orb () (*RestrictedTwoBodyRDM method*), 797
 n_orb () (*UnrestrictedOneBodyRDM method*), 903
 n_orb () (*UnrestrictedTwoBodyRDM method*), 904
 n_plus_states (*IcebergOptions attribute*), 1013
 n_plus_states (*PlainOptions attribute*), 1013
 n_qubits (*CircuitAnsatz property*), 338
 n_qubits (*ComposedAnsatz property*), 343
 n_qubits (*FermionSpaceAnsatzChemicallyAwareUCCSD property*), 377
 n_qubits (*FermionSpaceAnsatzUpCCGD property*), 382
 n_qubits (*FermionSpaceAnsatzkUpCCGSD property*), 388
 n_qubits (*FermionSpaceAnsatzkUpCCGSDSinglet property*), 393
 n_qubits (*FermionSpaceAnsatzUCCD property*), 366
 n_qubits (*FermionSpaceAnsatzUCCGD property*), 399
 n_qubits (*FermionSpaceAnsatzUCCGSD property*), 404
 n_qubits (*FermionSpaceAnsatzUCCSD property*), 360
 n_qubits (*FermionSpaceAnsatzUCCSDSinglet property*), 410
 n_qubits (*FermionSpaceStateExp property*), 355
 n_qubits (*FermionSpaceStateExpChemicallyAware property*), 371
 n_qubits (*GeneralAnsatz property*), 333
 n_qubits (*HamiltonianVariationalAnsatz property*), 449
 n_qubits (*HardwareEfficientAnsatz property*), 461
 n_qubits (*LayeredAnsatz property*), 456
 n_symbols (*MultiConfigurationAnsatz property*), 416
 n_symbols (*MultiConfigurationState property*), 421
 n_symbols (*MultiConfigurationStateBox property*), 426
 n_symbols (*QubitState property*), 1077
 n_symbols (*RealGeneralizedBasisRotationAnsatz property*), 432
 n_symbols (*RealRestrictedBasisRotationAnsatz property*), 437
 n_symbols (*RealUnrestrictedBasisRotationAnsatz property*), 443
 n_symbols (*TrotterAnsatz property*), 349
 n_rp (*FermionSpaceSupercell property*), 1046
 n_shots (*IterativePhaseEstimation property*), 990
 n_shots (*IterativePhaseEstimationQuantinuum property*), 995
 n_spin_orb (*FermionSpace property*), 1030
 n_spin_orb (*FermionSpaceBrillouin property*), 1038
 n_spin_orb (*FermionSpaceSupercell property*), 1046
 n_spin_orb (*ParaFermionSpace property*), 1052
 n_spin_orb () (*RestrictedOneBodyRDM method*), 796
 n_spin_orb () (*RestrictedTwoBodyRDM method*), 797
 n_spin_orb () (*UnrestrictedOneBodyRDM method*), 903
 n_spin_orb () (*UnrestrictedTwoBodyRDM method*), 904
 n_symbols (*CircuitAnsatz property*), 338
 n_symbols (*ComposedAnsatz property*), 343
 n_symbols (*FermionOperator property*), 695
 n_symbols (*FermionOperatorList property*), 716
 n_symbols (*FermionSpaceAnsatzChemicallyAwareUCCSD property*), 377
 n_symbols (*FermionSpaceAnsatzkUpCCGD property*), 382
 n_symbols (*FermionSpaceAnsatzkUpCCGSD property*), 388
 n_symbols (*FermionSpaceAnsatzkUpCCGSDSinglet property*), 393
 n_symbols (*FermionSpaceAnsatzUCCD property*), 366
 n_symbols (*FermionSpaceAnsatzUCCGD property*), 399
 n_symbols (*FermionSpaceAnsatzUCCGSD property*), 404
 n_symbols (*FermionSpaceAnsatzUCCSD property*), 360
 n_symbols (*FermionSpaceAnsatzUCCSDSinglet property*), 410
 n_symbols (*FermionSpaceStateExp property*), 355
 n_symbols (*FermionSpaceStateExpChemicallyAware property*), 371
 n_symbols (*FermionState property*), 1062
 n_symbols (*GeneralAnsatz property*), 333
 n_symbols (*HamiltonianVariationalAnsatz property*), 449
 n_symbols (*HardwareEfficientAnsatz property*), 461
 n_symbols (*LayeredAnsatz property*), 456
 n_symbols (*MultiConfigurationAnsatz property*), 416
 n_symbols (*MultiConfigurationState property*), 421
 n_symbols (*MultiConfigurationStateBox property*), 426
 n_symbols (*QubitOperator property*), 747
 n_symbols (*QubitOperatorList property*), 780

n_symbols (*QubitState* property), 1078
 n_symbols (*RealGeneralizedBasisRotationAnsatz* property), 432
 n_symbols (*RealRestrictedBasisRotationAnsatz* property), 437
 n_symbols (*RealUnrestrictedBasisRotationAnsatz* property), 443
 n_symbols (*State* property), 1092
 n_symbols (*SymmetryOperatorFermionic* property), 814
 n_symbols (*SymmetryOperatorFermionicFactorised* property), 836
 n_symbols (*SymmetryOperatorPauli* property), 859
 n_symbols (*SymmetryOperatorPauliFactorised* property), 894
 n_symbols (*TrotterAnsatz* property), 349
NaiveEulerIntegrator (class in *in quanto.minimizers*), 647
 nbytes (*ComputableNDArray* attribute), 509
 ndim (*ComputableNDArray* attribute), 509
 newbyteorder () (*ComputableNDArray* method), 509
 NONE (*CacheLevels* attribute), 580
NonOrthogonalMatricesComputable (class in *in quanto.computables.composite*), 546
 nonzero () (*ComputableNDArray* method), 510
 norm () (*ChemistryRestrictedIntegralOperator* method), 656
 norm () (*ChemistryRestrictedIntegralOperatorCompact* method), 661
 norm_coefficients () (*FermionOperator* method), 696
 norm_coefficients () (*FermionState* method), 1062
 norm_coefficients () (*QubitOperator* method), 747
 norm_coefficients () (*QubitState* method), 1078
 norm_coefficients () (*State* method), 1092
 norm_coefficients () (*SymmetryOperatorFermionic* method), 814
 norm_coefficients () (*SymmetryOperatorPauli* method), 860
 normal_ordered () (*FermionOperator* method), 696
 normal_ordered () (*SymmetryOperatorFermionic* method), 814
 normalized () (*FermionOperator* method), 696
 normalized () (*FermionState* method), 1062
 normalized () (*QubitOperator* method), 747
 normalized () (*QubitState* method), 1078
 normalized () (*State* method), 1092
 normalized () (*SymmetryOperatorFermionic* method), 815
 normalized () (*SymmetryOperatorPauli* method), 860
 num_entries (*Cache* property), 579
 num_entries (*ProtocolCache* property), 1012
 num_modes (*FermionState* property), 1062
 num_modes (*FermionStateString* property), 1069
 num_modes (*QubitState* property), 1078
 num_modes (*QubitStateString* property), 1084
 num_modes (*State* property), 1092
 num_modes (*StateString* property), 1098
 num_qubits (*QubitState* property), 1078
 num_spin_orbs (*FermionOperator* property), 696
 num_spin_orbs (*FermionOperatorList* property), 716
 num_spin_orbs (*SymmetryOperatorFermionic* property), 815
 num_spin_orbs (*SymmetryOperatorFermionicFactorised* property), 836
 numerator (*CacheLevels* attribute), 581
 numerator (*CacheSizeUnit* attribute), 582
 numerator (*CompilationLevel* attribute), 1015
 numerator (*CtrluStrat* attribute), 1017

O

occupations_ordered () (*FermionStateString* method), 1069
 occupations_ordered () (*QubitStateString* method), 1084
 occupations_ordered () (*StateString* method), 1098
 one_body_to_array () (*FCIDumpRestricted* method), 678
 ONLY_IDENTITIES_AND_ZERO (*FermionOperatorList.CompressScalarsBehavior* attribute), 703
 ONLY_IDENTITIES_AND_ZERO (*QubitOperatorList.CompressScalarsBehavior* attribute), 757
 ONLY_IDENTITIES_AND_ZERO (*SymmetryOperatorFermionicFactorised.CompressScalarsBehavior* attribute), 821
 ONLY_IDENTITIES_AND_ZERO (*SymmetryOperatorPauliFactorised.CompressScalarsBehavior* attribute), 870
 operator_class (*FermionOperatorList* attribute), 716
 operator_class (*QubitOperatorList* attribute), 780
 operator_class (*SymmetryOperatorFermionicFactorised* attribute), 836
 operator_class (*SymmetryOperatorPauliFactorised* attribute), 894
 operator_map () (*QubitMapping* class method), 627
 operator_map () (*QubitMappingBravyiKitaev* class method), 633
 operator_map () (*QubitMappingJordanWigner* class method), 630
 operator_map () (*QubitMappingParaparticular* class method), 641
 operator_map () (*QubitMappingParity* class method), 637
 OPERATOR_MAP_TYPES (*QubitMapping* attribute), 626
 OPERATOR_MAP_TYPES (*QubitMappingBravyiKitaev* attribute), 633
 OPERATOR_MAP_TYPES (*QubitMappingJordanWigner* attribute), 629

OPERATOR_MAP_TYPES (*QubitMappingParaparticular attribute*), 640

OPERATOR_MAP_TYPES (*QubitMappingParity attribute*), 637

operator_to_latex() (*FermionSpace method*), 1030

operator_to_latex() (*FermionSpaceBrillouin method*), 1038

operator_to_latex() (*FermionSpaceSupercell method*), 1046

optimisation_level (*IterativePhaseEstimation property*), 990

optimisation_level (*IterativePhaseEstimation-Quantinuum property*), 995

OptimizationMethod (*class in inquanto.minimizers*), 649

optimize() (*OrbitalOptimizer method*), 727

options (*MinimizerScipy property*), 647

orb_irreps() (*FermionSpace method*), 1030

OrbitalOptimizer (*class in inquanto.operators*), 726

OrbitalTransformer (*class in inquanto.operators*), 728

original (*AVAS property*), 1106

original (*CASSCF property*), 1107

orthonormalize() (*OrbitalOptimizer static method*), 728

orthonormalize() (*OrbitalTransformer static method*), 729

OUTER (*FermionOperatorList.CompressScalarsBehavior attribute*), 703

OUTER (*FermionOperatorList.FactoryCoefficientsLocation attribute*), 707

OUTER (*FermionOperator.TrotterizeCoefficientsLocation attribute*), 680

OUTER (*QubitOperatorList.CompressScalarsBehavior attribute*), 757

OUTER (*QubitOperatorList.FactoryCoefficientsLocation attribute*), 767

OUTER (*QubitOperator.TrotterizeCoefficientsLocation attribute*), 730

OUTER (*SymmetryOperatorFermionicFactorised.CompressScalarsBehavior attribute*), 821

OUTER (*SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation attribute*), 826

OUTER (*SymmetryOperatorFermionic.TrotterizeCoefficientsLocation attribute*), 798

OUTER (*SymmetryOperatorPauliFactorised.CompressScalarsBehavior attribute*), 870

OUTER (*SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation attribute*), 880

OUTER (*SymmetryOperator-*

Pauli.TrotterizeCoefficientsLocation attribute), 843

OUTSIDE_EXPONENT (*QubitOperatorList.ExpandExponentialProductCoefficientsBehavior attribute*), 762

OUTSIDE_EXPONENT (*SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior attribute*), 875

Overlap (*class in inquanto.computables.atomic*), 478

OverlapImag (*class in inquanto.computables.atomic*), 479

OverlapMatrixComputable (*class in inquanto.computables.composite*), 548

OverlapReal (*class in inquanto.computables.atomic*), 481

OverlapSquared (*class in inquanto.computables.atomic*), 483

OWNDATA (*ComputableNDArray attribute*), 503

P

pad() (*QubitOperator method*), 747

pad() (*SymmetryOperatorPauli method*), 860

padded() (*QubitOperator method*), 748

padded() (*QubitOperatorString method*), 791

padded() (*SymmetryOperatorPauli method*), 861

pairs() (*CompactTwoBodyIntegralsS4 static method*), 673

pairs() (*CompactTwoBodyIntegralsS8 static method*), 675

ParaFermionSpace (*class in inquanto.spaces*), 1050

parallelity_matrix() (*QubitOperatorList method*), 780

parallelity_matrix() (*SymmetryOperatorPauliFactorised method*), 894

parameters (*HadamardTestDerivativeOverlap attribute*), 972

parity_set() (*QubitMapping class method*), 627

parity_set() (*QubitMappingBravyiKitaev class method*), 634

parity_set() (*QubitMappingJordanWigner class method*), 631

parity_set() (*QubitMappingParaparticular class method*), 641

parity_set() (*QubitMappingParity class method*), 638

ParticleGFComputable (*class in inquanto.computables.composite*), 552

partition() (*ComputableNDArray method*), 510

partition() (*FermionOperatorList.CompressScalarsBehavior method*), 705

partition() (*FermionOperatorList.FactoryCoefficientsLocation method*), 710

partition() (*FermionOperator.TrotterizeCoefficientsLocation method*), 682

partition() *(QubitOperator method)*, 760
 partition() *(QubitOperator method)*, 764
 partition() *(QubitOperator method)*, 769
 partition() *(QubitOperator method)*, 733
 partition() *(SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method)*, 824
 partition() *(SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method)*, 829
 partition() *(SymmetryOperatorFermionic.TrotterizeCoefficientsLocation method)*, 801
 partition() *(SymmetryOperatorPauliFactorised.CompressScalarsBehavior method)*, 872
 partition() *(SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method)*, 877
 partition() *(SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method)*, 882
 partition() *(SymmetryOperatorPauli.TrotterizeCoefficientsLocation method)*, 846
 pattern_from_locations() *(DMETRHF static method)*, 587
 PAULI_EXP_BOX (*CtrluStrat attribute*), 1015
 PAULI_GADGET_RZZ (*CtrluStrat attribute*), 1016
 Pauli_GADGET_RZZ (*CtrluStrat attribute*), 1015
 pauli_list (*QubitOperatorString property*), 792
 pauli_strings (*QubitOperator property*), 749
 pauli_strings (*SymmetryOperatorPauli property*), 862
 PauliAveraging (*class in inquanto.protocols*), 905
 paulis (*ParaFermionSpace property*), 1052
 paulis (*QubitSpace property*), 1054
 pd_safe_eigh() (*in module inquanto.core*), 584
 PDM1234RealComputable (*class in inquanto.computables.composite*), 550
 permutation() (*FermionSpaceSupercell method*), 1047
 permutation_matrix() (*FermionSpaceSupercell method*), 1047
 permuted_operator() (*FermionOperator method*), 696
 permuted_operator() (*SymmetryOperatorFermionic method*), 815
 PhaseShift (*class in inquanto.protocols*), 980
 phayes_state (*AlgorithmBayesianQPE property*), 1267
 PLAIN (*CircuitEncoderQuantinuum attribute*), 1013, 1014
 PlainOptions (*class in inquanto.protocols*), 1012
 PMSV (*class in inquanto.protocols*), 1008
 print_group() (*FermionSpace method*), 1030
 PointGroup (*class in inquanto.symmetry*), 1099
 pop() (*ComputableList method*), 490
 pop() (*SymbolSet method*), 577
 populations (*IterativePhaseEstimationStatevector property*), 998
 post() (*CombinedMitigation method*), 1011
 post() (*PMSV method*), 1009
 post() (*SPAM method*), 1010
 post_propagation_evaluation() (*AlgorithmMLachlanImagTime method*), 329
 post_propagation_evaluation() (*AlgorithmMLachlanRealTime method*), 327
 post_propagation_evaluation() (*AlgorithmVQS method*), 326
 pre() (*CombinedMitigation method*), 1011
 pre() (*PMSV method*), 1009
 pre() (*SPAM method*), 1010
 prefix (*InQuantoContext property*), 584
 print_character_table() (*PointGroup method*), 1100
 print_json_report() (*ChemistryDriverPySCFEmbeddingGammaRHF method*), 1115
 print_json_report() (*ChemistryDriverPySCFEmbeddingGammaROHF_UHF method*), 1124
 print_json_report() (*ChemistryDriverPySCFEmbeddingRHF method*), 1133
 print_json_report() (*ChemistryDriverPySCFEmbeddingROHF method*), 1142
 print_json_report() (*ChemistryDriverPySCFEmbeddingROHF_UHF method*), 1151
 print_json_report() (*ChemistryDriverPySCFGammaRHF method*), 1160
 print_json_report() (*ChemistryDriverPySCFGammaROHF method*), 1169
 print_json_report() (*ChemistryDriverPySCFIntegrals method*), 1177
 print_json_report() (*ChemistryDriverPySCFMolecularRHF method*), 1185
 print_json_report() (*ChemistryDriverPySCFMolecularRHFQMMMCOSMO method*), 1195
 print_json_report() (*ChemistryDriverPySCFMolecularROHF method*), 1204
 print_json_report() (*ChemistryDriverPySCFMolecularROHFQMMMCOSMO method*), 1214
 print_json_report() (*ChemistryDriverPySCFMolecularUHF method*), 1222
 print_json_report() (*ChemistryDriverPySCFMolecularUHFQMMMCOSMO method*), 1232
 print_json_report() (*ChemistryDriverPySCFMolecularUHFQMMMCOSMO method*), 1233

mentumRHF method), 1235
print_json_report() (*ChemistryDriverPySCFMethod*, 1238)
print_json_report() (*DriverGeneralizedHubbard method*, 597)
print_json_report() (*DriverHubbardDimer method*, 598)
print_json_report() (*DriverIsing1D method*, 599)
print_json_report() (*DriverIsing1DRing method*, 599)
print_json_report() (*DriverIsingCustomConnectivity method*, 598)
print_report() (*SymbolDict method*, 574)
print_sceom_states() (*AlgorithmSCEOM method*, 321)
print_state() (*FermionSpace method*, 1030)
print_state() (*FermionSpaceBrillouin method*, 1038)
print_state() (*FermionSpaceSupercell method*, 1047)
print_table() (*ChemistryRestrictedIntegralOperator method*, 656)
print_table() (*ChemistryRestrictedIntegralOperator-Compact method*, 661)
print_table() (*ChemistryUnrestrictedIntegralOperator method*, 666)
print_table() (*ChemistryUnrestrictedIntegralOperator-Compact method*, 671)
print_table() (*FermionOperator method*, 697)
print_table() (*FermionOperatorList method*, 716)
print_table() (*FermionState method*, 1062)
print_table() (*PySCFChemistryRestrictedIntegralOperator method*, 1251)
print_table() (*PySCFChemistryUnrestrictedIntegral-Operator method*, 1256)
print_table() (*QubitOperator method*, 749)
print_table() (*QubitOperatorList method*, 781)
print_table() (*QubitState method*, 1078)
print_table() (*State method*, 1092)
print_table() (*SymmetryOperatorFermionic method*, 816)
print_table() (*SymmetryOperatorFermionicFactorised method*, 836)
print_table() (*SymmetryOperatorPauli method*, 862)
print_table() (*SymmetryOperatorPauliFactorised method*, 894)
print_tree() (*CommutatorComputable method*, 533)
print_tree() (*ComputableFunction method*, 487)
print_tree() (*ComputableInt method*, 488)
print_tree() (*ComputableList method*, 490)
print_tree() (*ComputableNDArray method*, 511)
print_tree() (*ComputableNode method*, 529)
print_tree() (*ComputableSingleChild method*, 530)
print_tree() (*ExpectationValue method*, 464)
print_tree() (*ExpectationValueBraDerivativeImag method*, 465)
print_tree() (*ExpectationValueBraDerivativeReal method*, 467)
print_tree() (*ExpectationValueDerivative method*, 469)
print_tree() (*ExpectationValueKetDerivativeImag method*, 471)
print_tree() (*ExpectationValueKetDerivativeReal method*, 472)
print_tree() (*ExpectationValueNonHermitian method*, 474)
print_tree() (*ExpectationValueSumComputable method*, 535)
print_tree() (*HoleGFComputable method*, 536)
print_tree() (*KrylovSubspaceComputable method*, 540)
print_tree() (*LanczosCoefficientsComputable method*, 542)
print_tree() (*LanczosMatrixComputable method*, 544)
print_tree() (*ManyBodyGFComputable method*, 546)
print_tree() (*MetricTensorImag method*, 476)
print_tree() (*MetricTensorReal method*, 477)
print_tree() (*NonOrthogonalMatricesComputable method*, 548)
print_tree() (*Overlap method*, 479)
print_tree() (*OverlapImag method*, 481)
print_tree() (*OverlapMatrixComputable method*, 549)
print_tree() (*OverlapReal method*, 483)
print_tree() (*OverlapSquared method*, 485)
print_tree() (*ParticleGFComputable method*, 553)
print_tree() (*PDM1234RealComputable method*, 551)
print_tree() (*QCM4Computable method*, 555)
print_tree() (*QSEMatricesComputable method*, 557)
print_tree() (*RDM1234RealComputable method*, 559)
print_tree() (*RestrictedOneBodyRDMComputable method*, 560)
print_tree() (*RestrictedOneBodyRDMRealComputable method*, 562)
print_tree() (*SCEOMMatrixComputable method*, 564)
print_tree() (*SpinlessNBodyPDMArrayRealComputable method*, 566)
print_tree() (*SpinlessNBodyRDMArrayRealComputable method*, 568)
print_tree() (*UnrestrictedOneBodyRDMComputable method*, 570)
print_tree() (*UnrestrictedOneBodyRDMRealComputable method*, 571)
prod() (*ComputableNDArray method*, 511)
ProjectiveMeasurements (*class in inquanto.protocols*, 999)
propagate() (*in module inquanto.express*, 600)
ProtocolCache (*class in inquanto.protocols*, 1011)
ProtocolList (*class in inquanto.protocols*, 1004)
ptp() (*ComputableNDArray method*, 511)
put() (*ComputableNDArray method*, 512)
PySCFChemistryRestrictedIntegralOperator

(*class in inquanto.extensions.pyscf*), 1248
PySCFChemistryUnrestrictedIntegralOperator
 (*class in inquanto.extensions.pyscf*), 1253

Q

QCM4Computable (*class in inquanto.computables.composite*), 553
QSEMatricesComputable (*class in inquanto.computables.composite*), 555
quantum_label() (*FermionSpace method*), 1031
quantum_label() (*FermionSpaceBrillouin method*), 1039
quantum_label() (*FermionSpaceSupercell method*), 1048
quantum_label() (*ParaFermionSpace method*), 1052
quantum_number() (*FermionSpace method*), 1031
quantum_number() (*FermionSpaceBrillouin method*), 1039
quantum_number() (*FermionSpaceSupercell method*), 1048
quantum_number() (*ParaFermionSpace method*), 1052
quantum_number_kp() (*FermionSpaceBrillouin method*), 1039
quantum_number_orb() (*FermionSpace method*), 1031
quantum_number_orb() (*FermionSpaceBrillouin method*), 1039
quantum_number_orb() (*FermionSpaceSupercell method*), 1048
quantum_number_orb() (*ParaFermionSpace method*), 1053
quantum_number_rp() (*FermionSpaceSupercell method*), 1048
quantum_number_spin() (*FermionSpace method*), 1032
quantum_number_spin() (*FermionSpaceBrillouin method*), 1040
quantum_number_spin() (*FermionSpaceSupercell method*), 1048
quantum_number_spin() (*ParaFermionSpace method*), 1053
qubit_encode() (*ChemistryRestrictedIntegralOperator method*), 656
qubit_encode() (*ChemistryRestrictedIntegralOperator Compact method*), 662
qubit_encode() (*ChemistryUnrestrictedIntegralOperator method*), 666
qubit_encode() (*ChemistryUnrestrictedIntegralOperatorCompact method*), 671
qubit_encode() (*FermionOperator method*), 697
qubit_encode() (*FermionOperatorList method*), 717
qubit_encode() (*FermionState method*), 1063
qubit_encode() (*FermionStateString method*), 1069
qubit_encode() (*PySCFChemistryRestrictedIntegralOperator method*), 1252

qubit_encode() (*PySCFChemistryUnrestrictedIntegralOperator method*), 1256
qubit_encode() (*SymmetryOperatorFermionic method*), 816
qubit_encode() (*SymmetryOperatorFermionicFactorised method*), 836
qubit_id_list (*QubitOperatorString property*), 792
qubit_list (*QubitOperatorString property*), 792
QubitMapping (*class in inquanto.mappings*), 626
QubitMappingBravyiKitaev (*class in inquanto.mappings*), 633
QubitMappingJordanWigner (*class in inquanto.mappings*), 629
QubitMappingParaparticular (*class in inquanto.mappings*), 640
QubitMappingParity (*class in inquanto.mappings*), 636
QubitOperator (*class in inquanto.operators*), 729
QubitOperatorList (*class in inquanto.operators*), 756
QubitOperatorList.CompressScalarsBehavior
 (*class in inquanto.operators*), 757
QubitOperatorList.ExpandExponentialProductCoefficientsBehavior
 (*class in inquanto.operators*), 762
QubitOperatorList.FactoryCoefficientsLocation
 (*class in inquanto.operators*), 766
QubitOperatorString (*class in inquanto.operators*), 788
QubitOperator.TrotterizeCoefficientsLocation
 (*class in inquanto.operators*), 730
QubitSpace (*class in inquanto.spaces*), 1053
QubitState (*class in inquanto.states*), 1070
QubitStateString (*class in inquanto.states*), 1082
qubitwise_anticomutes_with() (*QubitOperator method*), 749
qubitwise_anticomutes_with() (*QubitOperatorString method*), 792
qubitwise_anticomutes_with() (*SymmetryOperatorPauli method*), 862
qubitwise_commutates_with() (*QubitOperator method*), 749
qubitwise_commutates_with() (*QubitOperatorString method*), 793
qubitwise_commutates_with() (*SymmetryOperatorPauli method*), 862
qubitwise_compatibility_matrix() (*QubitOperatorList method*), 781
qubitwise_compatibility_matrix() (*SymmetryOperatorPauliFactorised method*), 894
qubitwise_incompatibility_matrix() (*QubitOperatorList method*), 781
qubitwise_incompatibility_matrix() (*SymmetryOperatorPauliFactorised method*), 894

R

random_circuit_ansatz() (*in module in*

quanto.express), 601
randomize_xyz() (GeometryMolecular method), 610
randomize_xyz() (GeometryPeriodic method), 621
ravel() (ComputableNDArray method), 512
RDM1234RealComputable (class in in-quanto.computables.composite), 557
real (CacheLevels attribute), 581
real (CacheSizeUnit attribute), 582
real (ChemistryRestrictedIntegralOperator property), 657
real (ChemistryRestrictedIntegralOperatorCompact property), 662
real (CompactTwoBodyIntegralsS4 property), 674
real (CompactTwoBodyIntegralsS8 property), 675
real (CompilationLevel attribute), 1015
real (ComputableNDArray attribute), 512
real (CtrluStrat attribute), 1017
RealGeneralizedBasisRotationAnsatz (class in in-quanto.ansatzes), 428
RealRestrictedBasisRotationAnsatz (class in in-quanto.ansatzes), 433
RealUnrestrictedBasisRotationAnsatz (class in in-quanto.ansatzes), 439
rebuild() (ComputeUncompute method), 924
rebuild() (ComputeUncomputeFactorizedOverlap method), 961
rebuild() (DestructiveSwapTest method), 930
rebuild() (FactorizedOverlap method), 949
rebuild() (HadamardTest method), 918
rebuild() (HadamardTestDerivative method), 979
rebuild() (HadamardTestDerivativeOverlap method), 972
rebuild() (HadamardTestOverlap method), 943
rebuild() (IterativePhaseEstimation method), 990
rebuild() (IterativePhaseEstimationQuantinuum method), 995
rebuild() (IterativePhaseEstimationStatevector method), 998
rebuild() (PauliAveraging method), 911
rebuild() (PhaseShift method), 985
rebuild() (ProjectiveMeasurements method), 1003
rebuild() (SwapFactorizedOverlap method), 955
rebuild() (SwapTest method), 937
reduce_exponents_by_commutation() (QubitOperatorList method), 781
reduce_exponents_by_commutation() (SymmetryOperatorPauliFactorised method), 895
reference_circuit_builder() (in module in-quanto.ansatzes), 350
reference_qubit_state() (CircuitAnsatz method), 338
reference_qubit_state() (ComposedAnsatz method), 343
reference_qubit_state() (FermionSpaceAnsatzChemicallyAwareUCCSD method), 377
reference_qubit_state() (FermionSpaceAnsatzUpCCGD method), 382
reference_qubit_state() (FermionSpaceAnsatzUpCCGSD method), 388
reference_qubit_state() (FermionSpaceAnsatzUpCCGSDSinglet method), 393
reference_qubit_state() (FermionSpaceAnsatzUCCD method), 366
reference_qubit_state() (FermionSpaceAnsatzUC-CGD method), 399
reference_qubit_state() (FermionSpaceAnsatzUC-GSD method), 404
reference_qubit_state() (FermionSpaceAnsatzUCCSD method), 361
reference_qubit_state() (FermionSpaceAnsatzUCCSDSinglet method), 410
reference_qubit_state() (FermionSpaceStateExp method), 355
reference_qubit_state() (FermionSpaceStateExpChemicallyAware method), 371
reference_qubit_state() (GeneralAnsatz method), 333
reference_qubit_state() (HamiltonianVariationalAnsatz method), 449
reference_qubit_state() (HardwareEfficientAnsatz method), 461
reference_qubit_state() (LayeredAnsatz method), 456
reference_qubit_state() (MultiConfigurationAnsatz method), 416
reference_qubit_state() (MultiConfigurationState method), 421
reference_qubit_state() (MultiConfigurationState-Box method), 426
reference_qubit_state() (RealGeneralizedBasisRotationAnsatz method), 432
reference_qubit_state() (RealRestrictedBasisRotationAnsatz method), 437
reference_qubit_state() (RealUnrestrictedBasisRotationAnsatz method), 443
reference_qubit_state() (TrotterAnsatz method), 349
register_size() (QubitOperatorString method), 793
remainder_set() (QubitMappingBravyiKitaev class method), 634
remove() (ComputableList method), 490
remove_global_phase() (FermionOperator method), 697
remove_global_phase() (FermionState method), 1063
remove_global_phase() (QubitOperator method), 750
remove_global_phase() (QubitState method), 1078
remove_global_phase() (State method), 1092
remove_global_phase() (SymmetryOperatorFermionic method), 816

<code>remove_global_phase()</code>	<i>(SymmetryOperatorPauli method)</i> , 862	<i>torList.CompressScalarsBehavior</i>	<i>method), 760</i>
<code>removeprefix()</code>	<i>(FermionOperatorList.CompressScalarsBehavior method)</i> , 705	<code>removesuffix()</code>	<i>(QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method), 765</i>
<code>removeprefix()</code>	<i>(FermionOperatorList.FactoryCoefficientsLocation method)</i> , 710	<code>removesuffix()</code>	<i>(QubitOperatorList.FactoryCoefficientsLocation method), 769</i>
<code>removeprefix()</code>	<i>(FermionOperatorList.TrotterizeCoefficientsLocation method)</i> , 682	<code>removesuffix()</code>	<i>(QubitOperatorList.TrotterizeCoefficientsLocation method), 733</i>
<code>removeprefix()</code>	<i>(QubitOperatorList.CompressScalarsBehavior method)</i> , 760	<code>removesuffix()</code>	<i>(SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method), 824</i>
<code>removeprefix()</code>	<i>(QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method)</i> , 765	<code>removesuffix()</code>	<i>(SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method), 829</i>
<code>removeprefix()</code>	<i>(QubitOperatorList.FactoryCoefficientsLocation method)</i> , 769	<code>removesuffix()</code>	<i>(SymmetryOperatorFermionicFactorised.TrotterizeCoefficientsLocation method), 801</i>
<code>removeprefix()</code>	<i>(QubitOperatorList.TrotterizeCoefficientsLocation method)</i> , 733	<code>removesuffix()</code>	<i>(SymmetryOperatorPauliFactorised.CompressScalarsBehavior method), 873</i>
<code>removeprefix()</code>	<i>(SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method)</i> , 824	<code>removesuffix()</code>	<i>(SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method), 878</i>
<code>removeprefix()</code>	<i>(SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method)</i> , 829	<code>removesuffix()</code>	<i>(SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method), 882</i>
<code>removeprefix()</code>	<i>(SymmetryOperatorFermionicTrotterizeCoefficientsLocation method)</i> , 801	<code>removesuffix()</code>	<i>(SymmetryOperatorPauliFactorised.TrotterizeCoefficientsLocation method), 846</i>
<code>removeprefix()</code>	<i>(SymmetryOperatorPauliFactorised.CompressScalarsBehavior method)</i> , 872	<code>renamed()</code>	<i>(SymbolSet method), 577</i>
<code>removeprefix()</code>	<i>(SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method)</i> , 877	<code>repeat()</code>	<i>(ComputableNDArray method), 512</i>
<code>removeprefix()</code>	<i>(SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method)</i> , 882	<code>replace()</code>	<i>(FermionOperatorList.CompressScalarsBehavior method), 705</i>
<code>removeprefix()</code>	<i>(SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method)</i> , 846	<code>replace()</code>	<i>(FermionOperatorList.FactoryCoefficientsLocation method), 710</i>
<code>removeprefix()</code>	<i>(SymmetryOperatorPauliFactorised.TrotterizeCoefficientsLocation method)</i> , 682	<code>replace()</code>	<i>(FermionOperatorList.TrotterizeCoefficientsLocation method), 682</i>
<code>removeprefix()</code>	<i>(SymmetryOperatorPauli.TrotterizeCoefficientsLocation method)</i> , 846	<code>replace()</code>	<i>(QubitOperatorList.CompressScalarsBehavior method), 760</i>
<code>removesuffix()</code>	<i>(FermionOperatorList.CompressScalarsBehavior method)</i> , 705	<code>replace()</code>	<i>(QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method), 765</i>
<code>removesuffix()</code>	<i>(FermionOperatorList.FactoryCoefficientsLocation method)</i> , 710	<code>replace()</code>	<i>(QubitOperatorList.FactoryCoefficientsLocation method), 770</i>
<code>removesuffix()</code>	<i>(FermionOperatorList.TrotterizeCoefficientsLocation method)</i> , 682	<code>replace()</code>	<i>(QubitOperatorList.TrotterizeCoefficientsLocation method), 733</i>
<code>removesuffix()</code>	<i>(QubitOperatorList.TrotterizeCoefficientsLocation method)</i>	<code>replace()</code>	<i>(SymmetryOperatorFermionicFactorised.TrotterizeCoefficientsLocation method), 801</i>

torised.CompressScalarsBehavior *method),*
824

replace() *(SymmetryOperatorFermionicFac-*
torised.FactoryCoefficientsLocation *method),*
829

replace() *(SymmetryOperator-*
Fermionic.TrotterizeCoefficientsLocation
method), **801**

replace() *(SymmetryOperatorPauliFac-*
torised.CompressScalarsBehavior *method),*
873

replace() *(SymmetryOperatorPauliFac-*
torised.ExpandExponentialProductCoefficientsBehavior
method), **878**

replace() *(SymmetryOperatorPauliFac-*
torised.FactoryCoefficientsLocation *method),*
882

replace() *(SymmetryOperatorPauli-*
Pauli.TrotterizeCoefficientsLocation *method),*
846

report() (*Cache method*), **579**

report() (*ProtocolCache method*), **1012**

rescale_position_vectors() (*GeometryMolecular*
method), **610**

rescale_position_vectors() (*GeometryPeriodic*
method), **622**

reset_reference() (*CircuitAnsatz method*), **338**

reset_reference() (*ComposedAnsatz method*), **343**

reset_reference() (*FermionSpaceAnsatzChemi-*
callyAwareUCCSD method), **377**

reset_reference() (*FermionSpaceAnsatzkUpCCGD*
method), **382**

reset_reference() (*FermionSpaceAnsatzkUpCCGSD*
method), **388**

reset_reference() (*FermionSpaceAnsatzkUpCCGS-*
DSinglet method), **393**

reset_reference() (*FermionSpaceAnsatzUCCD*
method), **366**

reset_reference() (*FermionSpaceAnsatzUCCGD*
method), **399**

reset_reference() (*FermionSpaceAnsatzUCCGSD*
method), **404**

reset_reference() (*FermionSpaceAnsatzUCCSD*
method), **361**

reset_reference() (*FermionSpaceAnsatzUCCSDSinglet*
method), **410**

reset_reference() (*FermionSpaceStateExp* *method*),
355

reset_reference() (*FermionSpaceStateExpChemi-*
callyAware method), **371**

reset_reference() (*GeneralAnsatz method*), **333**

reset_reference() (*HamiltonianVariationalAnsatz*
method), **449**

reset_reference() (*HardwareEfficientAnsatz* *method*),

461

reset_reference() (*LayeredAnsatz method*), **456**

reset_reference() (*MultiConfigurationAnsatz*
method), **416**

reset_reference() (*MultiConfigurationState* *method*),
421

reset_reference() (*MultiConfigurationStateBox*
method), **426**

reset_reference() (*RealGeneralizedBasisRotatio-*
nAnsatz method), **432**

reset_reference() (*RealRestrictedBasisRotatio-*
nAnsatz method), **437**

reset_reference() (*RealUnrestrictedBasisRotatio-*
nAnsatz method), **443**

reset_reference() (*TrotterAnsatz method*), **349**

reshape() (*ComputableNDArray method*), **513**

resize() (*ComputableNDArray method*), **513**

RESTRICTED (*IntegralType attribute*), **726**

restricted_basis_rotation_to_circuit() (*in*
module inquanto.ansatzes), **445**

RestrictedOneBodyRDM (*class in inquanto.operators*),
795

RestrictedOneBodyRDMComputable (*class in in-*
quanto.computables.composite), **559**

RestrictedOneBodyRDMRealComputable (*class in in-*
quanto.computables.composite), **560**

RestrictedTwoBodyRDM (*class in inquanto.operators*),
797

results (*ProjectiveMeasurements attribute*), **1003**

retrieve() (*ComputeUncompute method*), **924**

retrieve() (*ComputeUncomputeFactorizedOverlap*
method), **962**

retrieve() (*DestructiveSwapTest method*), **931**

retrieve() (*FactorizedOverlap method*), **949**

retrieve() (*HadamardTest method*), **918**

retrieve() (*HadamardTestDerivative method*), **979**

retrieve() (*HadamardTestDerivativeOverlap* *method*),
973

retrieve() (*HadamardTestOverlap method*), **943**

retrieve() (*IterativePhaseEstimation method*), **991**

retrieve() (*IterativePhaseEstimationQuantinuum*
method), **995**

retrieve() (*IterativePhaseEstimationStatevector*
method), **998**

retrieve() (*PauliAveraging method*), **911**

retrieve() (*PhaseShift method*), **986**

retrieve() (*ProjectiveMeasurements method*), **1004**

retrieve() (*ProtocolList method*), **1008**

retrieve() (*SwapFactorizedOverlap method*), **956**

retrieve() (*SwapTest method*), **937**

retrieve_experiment() (*AlgorithmDeterministicQPE*
method), **323**

retrotterize() (*FermionOperatorList method*), **717**

retrotterize() (*QubitOperatorList method*), **782**

<code>retrotterize()</code>	<i>(SymmetryOperatorFermionicFactorised method)</i>	836	<code>rfind()</code>	<i>(SymmetryOperatorPauli.TrotterizeCoefficientsLocation method)</i>	883
<code>retrotterize()</code>	<i>(SymmetryOperatorPauliFactorised method)</i>	895	<code>rho_set()</code>	<i>(QubitMapping class method)</i>	846
<code>reverse()</code>	<i>(ComputableList method)</i>	490	<code>rho_set()</code>	<i>(QubitMappingBravyiKitaev class method)</i>	628
<code>reverse_rp_permutation()</code>	<i>(FermionSpaceSupercell method)</i>	1049	<code>rho_set()</code>	<i>(QubitMappingJordanWigner class method)</i>	635
<code>reversed_order()</code>	<i>(FermionOperator method)</i>	697	<code>rho_set()</code>	<i>(QubitMappingParaparticular class method)</i>	631
<code>reversed_order()</code>	<i>(FermionOperatorList method)</i>	718	<code>rho_set()</code>	<i>(QubitMappingParity class method)</i>	642
<code>reversed_order()</code>	<i>(FermionState method)</i>	1063	<code>rindex()</code>	<i>(FermionOperatorList.CompressScalarsBehavior method)</i>	638
<code>reversed_order()</code>	<i>(QubitOperator method)</i>	750	<code>rindex()</code>	<i>(FermionOperatorList.FactoryCoefficientsLocation method)</i>	706
<code>reversed_order()</code>	<i>(QubitOperatorList method)</i>	783	<code>rindex()</code>	<i>(FermionOperator.TrotterizeCoefficientsLocation method)</i>	710
<code>reversed_order()</code>	<i>(QubitState method)</i>	1078	<code>rindex()</code>	<i>(FermionOperator.TrotterizeCoefficientsLocation method)</i>	683
<code>reversed_order()</code>	<i>(State method)</i>	1092	<code>rindex()</code>	<i>(QubitOperatorList.CompressScalarsBehavior method)</i>	760
<code>reversed_order()</code>	<i>(SymmetryOperatorFermionicFactorised method)</i>	816	<code>rindex()</code>	<i>(QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method)</i>	765
<code>reversed_order()</code>	<i>(SymmetryOperatorFermionicFactorised method)</i>	838	<code>rindex()</code>	<i>(QubitOperatorList.FactoryCoefficientsLocation method)</i>	770
<code>reversed_order()</code>	<i>(SymmetryOperatorPauli method)</i>	863	<code>rindex()</code>	<i>(QubitOperator.TrotterizeCoefficientsLocation method)</i>	733
<code>rfind()</code>	<i>(SymmetryOperatorPauliFactorised method)</i>	896	<code>rindex()</code>	<i>(SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method)</i>	825
<code>rfind()</code>	<i>(FermionOperatorList.CompressScalarsBehavior method)</i>	706	<code>rindex()</code>	<i>(SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method)</i>	829
<code>rfind()</code>	<i>(FermionOperatorList.FactoryCoefficientsLocation method)</i>	710	<code>rindex()</code>	<i>(SymmetryOperatorFermionic.TrotterizeCoefficientsLocation method)</i>	801
<code>rfind()</code>	<i>(FermionOperator.TrotterizeCoefficientsLocation method)</i>	683	<code>rindex()</code>	<i>(SymmetryOperatorPauliFactorised.CompressScalarsBehavior method)</i>	873
<code>rfind()</code>	<i>(QubitOperatorList.CompressScalarsBehavior method)</i>	760	<code>rindex()</code>	<i>(SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method)</i>	878
<code>rfind()</code>	<i>(QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method)</i>	765	<code>rindex()</code>	<i>(SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method)</i>	883
<code>rfind()</code>	<i>(QubitOperatorList.FactoryCoefficientsLocation method)</i>	770	<code>rindex()</code>	<i>(SymmetryOperatorPauli.TrotterizeCoefficientsLocation method)</i>	846
<code>rfind()</code>	<i>(QubitOperator.TrotterizeCoefficientsLocation method)</i>	733	<code>rjust()</code>	<i>(FermionOperatorList.CompressScalarsBehavior method)</i>	706
<code>rfind()</code>	<i>(SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method)</i>	824	<code>rjust()</code>	<i>(FermionOperator.TrotterizeCoefficientsLocation method)</i>	846
<code>rfind()</code>	<i>(SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method)</i>	829			
<code>rfind()</code>	<i>(SymmetryOperatorFermionic.TrotterizeCoefficientsLocation method)</i>	801			
<code>rfind()</code>	<i>(SymmetryOperatorPauliFactorised.CompressScalarsBehavior method)</i>	873			
<code>rfind()</code>	<i>(SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method)</i>	878			
<code>rfind()</code>	<i>(SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method)</i>	883			

```

        torList.FactoryCoefficientsLocation      method),       610
    rjust() (FermionOperator.TrotterizeCoefficientsLocation
        method), 683
    rjust() (QubitOperatorList.CompressScalarsBehavior
        method), 760
    rjust() (QubitOperatorList.ExpandExponentialProductCoefficientsBehavior
        method), 765
    rjust() (QubitOperatorList.FactoryCoefficientsLocation
        method), 770
    rjust() (QubitOperator.TrotterizeCoefficientsLocation
        method), 733
    rjust() (SymmetryOperatorFermionicFac-
        torised.CompressScalarsBehavior      method), 825
    rjust() (SymmetryOperatorFermionicFac-
        torised.FactoryCoefficientsLocation      method), 829
    rjust() (SymmetryOperatorFermionicTrotterizeCoefficientsLocation
        method), 801
    rjust() (SymmetryOperatorPauliFac-
        torised.CompressScalarsBehavior      method), 873
    rjust() (SymmetryOperatorPauliFac-
        torised.ExpandExponentialProductCoefficientsBehavior
        method), 878
    rjust() (SymmetryOperatorPauliFac-
        torised.FactoryCoefficientsLocation      method), 883
    rjust() (SymmetryOperatorPauliTrotterizeCoefficientsLocation
        method), 846
    rotate() (ChemistryRestrictedIntegralOperator method), 657
    rotate() (ChemistryRestrictedIntegralOperatorCompact
        method), 662
    rotate() (ChemistryUnrestrictedIntegralOperator
        method), 666
    rotate() (ChemistryUnrestrictedIntegralOperatorCom-
        pact method), 671
    rotate() (CompactTwoBodyIntegralsS4 method), 674
    rotate() (CompactTwoBodyIntegralsS8 method), 676
    rotate() (PySCFChemistryRestrictedIntegralOperator
        method), 1252
    rotate() (PySCFChemistryUnrestrictedIntegralOperator
        method), 1256
    rotate() (RestrictedOneBodyRDM method), 796
    rotate() (RestrictedTwoBodyRDM method), 797
    rotate() (UnrestrictedOneBodyRDM method), 903
    rotate() (UnrestrictedTwoBodyRDM method), 904
    rotate_ansatz_restricted() (in module in-
        quanto.ansatzes), 445
    rotate_around_axis() (GeometryMolecular method),
        622
    rotate_around_vector() (GeometryMolecular
        method), 611
    rotate_around_vector() (GeometryPeriodic method),
        622
    round() (ComputableNDArray method), 514
    rpartition() (FermionOper-
        torList.CompressScalarsBehavior      method), 706
    rpartition() (FermionOper-
        torList.FactoryCoefficientsLocation      method), 711
    rpartition() (FermionOper-
        tor.TrotterizeCoefficientsLocation      method), 683
    rpartition() (QubitOper-
        torList.CompressScalarsBehavior      method), 760
    rpartition() (QubitOper-
        torList.ExpandExponentialProductCoefficientsBehavior
        method), 765
    rpartition() (QubitOper-
        torList.FactoryCoefficientsLocation      method), 770
    rpartition() (QubitOper-
        tor.TrotterizeCoefficientsLocation      method), 734
    rpartition() (SymmetryOperatorFermionicFac-
        torised.CompressScalarsBehavior      method), 825
    rpartition() (SymmetryOperatorFermionicFac-
        torised.FactoryCoefficientsLocation      method), 829
    rpartition() (SymmetryOperatorFermionicTrotterizeCoefficientsLocation
        method), 802
    rpartition() (SymmetryOperatorPauliFac-
        torised.CompressScalarsBehavior      method), 873
    rpartition() (SymmetryOperatorPauliFac-
        torised.ExpandExponentialProductCoefficientsBehavior
        method), 878
    rpartition() (SymmetryOperatorPauliFac-
        torised.FactoryCoefficientsLocation      method), 883
    rpartition() (SymmetryOperatorPauliTrotterizeCoefficientsLocation
        method), 846
    rsplit() (FermionOper-
        torList.CompressScalarsBehavior      method), 706
    rsplit() (FermionOper-

```

<code>torList.FactoryCoefficientsLocation</code>	<code>method),</code>	<code>torised.CompressScalarsBehavior</code>	<code>method),</code>
<code>711</code>		<code>825</code>	
<code>rsplit()</code>	<code>(FermionOper-</code>	<code>rstrip()</code>	<code>(SymmetryOperatorFermionicFac-</code>
<code>tor.TrotterizeCoefficientsLocation</code>	<code>method),</code>	<code>torised.FactoryCoefficientsLocation</code>	<code>method),</code>
<code>683</code>		<code>830</code>	
<code>rsplit()</code>	<code>(QubitOperatorList.CompressScalarsBehavior</code>	<code>rstrip()</code>	<code>(SymmetryOperator-</code>
<code>method),</code>	<code>760</code>	<code>Fermionic.TrotterizeCoefficientsLoca-</code>	<code>Fermionic.TrotterizeCoefficientsLoca-</code>
<code>rsplit()</code>	<code>(QubitOpera-</code>	<code>tion</code>	<code>tion),</code>
<code>torList.ExpandExponentialProductCoefficientsBehavior</code>	<code>method),</code>	<code>trip()</code>	<code>(SymmetryOperatorPauliFac-</code>
<code>method),</code>	<code>765</code>	<code>torised.CompressScalarsBehavior</code>	<code>method),</code>
<code>rsplit()</code>	<code>(QubitOperatorList.FactoryCoefficientsLocation</code>	<code>873</code>	<code>873</code>
<code>method),</code>	<code>770</code>	<code>rstrip()</code>	<code>(SymmetryOperatorPauliFac-</code>
<code>rsplit()</code>	<code>(QubitOperator.TrotterizeCoefficientsLocation</code>	<code>torised.ExpandExponentialProductCoefficientsBehavior</code>	<code>method),</code>
<code>method),</code>	<code>734</code>	<code>method),</code>	<code>878</code>
<code>rsplit()</code>	<code>(SymmetryOperatorFermionicFac-</code>	<code>rstrip()</code>	<code>(SymmetryOperatorPauliFac-</code>
<code>torised.CompressScalarsBehavior</code>	<code>method),</code>	<code>torised.FactoryCoefficientsLocation</code>	<code>method),</code>
<code>825</code>		<code>883</code>	
<code>rsplit()</code>	<code>(SymmetryOperatorFermionicFac-</code>	<code>rstrip()</code>	<code>(SymmetryOperator-</code>
<code>torised.FactoryCoefficientsLocation</code>	<code>method),</code>	<code>Pauli.TrotterizeCoefficientsLocation</code>	<code>Pauli.TrotterizeCoefficientsLocation</code>
<code>830</code>		<code>method),</code>	<code>847</code>
<code>rsplit()</code>	<code>(SymmetryOperator-</code>	<code>run()</code>	<code>(AlgorithmAdaptVQE method),</code>
<code>Fermionic.TrotterizeCoefficientsLocation</code>	<code>method),</code>	<code>315</code>	<code>315</code>
<code>802</code>		<code>run()</code>	<code>(AlgorithmBayesianQPE method),</code>
<code>rsplit()</code>	<code>(SymmetryOperatorPauliFac-</code>	<code>1267</code>	<code>1267</code>
<code>torised.CompressScalarsBehavior</code>	<code>method),</code>	<code>run()</code>	<code>(AlgorithmDeterministicQPE method),</code>
<code>873</code>		<code>323</code>	<code>323</code>
<code>rsplit()</code>	<code>(SymmetryOperatorPauliFac-</code>	<code>run()</code>	<code>(AlgorithmFermionicAdaptVQE method),</code>
<code>torised.ExpandExponentialProductCoefficientsBehavior</code>	<code>method),</code>	<code>314</code>	<code>314</code>
<code>method),</code>	<code>878</code>	<code>run()</code>	<code>(AlgorithmInfoTheoryQPE method),</code>
<code>rsplit()</code>	<code>(SymmetryOperatorPauliFac-</code>	<code>324</code>	<code>324</code>
<code>torised.FactoryCoefficientsLocation</code>	<code>method),</code>	<code>run()</code>	<code>(AlgorithmIQEB method),</code>
<code>883</code>		<code>316</code>	
<code>rsplit()</code>	<code>(SymmetryOperatorPauliFac-</code>	<code>run()</code>	<code>(AlgorithmKitaevQPE method),</code>
<code>torised.ExpandExponentialProductCoefficientsBehavior</code>	<code>method),</code>	<code>325</code>	<code>325</code>
<code>method),</code>	<code>878</code>	<code>run()</code>	<code>(AlgorithmMcLachlanImagTime method),</code>
<code>rsplit()</code>	<code>(SymmetryOperatorPauliFac-</code>	<code>329</code>	<code>329</code>
<code>torised.FactoryCoefficientsLocation</code>	<code>method),</code>	<code>run()</code>	<code>(AlgorithmMcLachlanRealTime method),</code>
<code>846</code>		<code>328</code>	
<code>rsplit()</code>	<code>(SymmetryOperator-</code>	<code>run()</code>	<code>(AlgorithmQSE method),</code>
<code>Pauli.TrotterizeCoefficientsLocation</code>	<code>method),</code>	<code>320</code>	<code>320</code>
<code>846</code>		<code>run()</code>	<code>(AlgorithmSCEOM method),</code>
<code>rstrip()</code>	<code>(FermionOper-</code>	<code>321</code>	<code>321</code>
<code>torList.CompressScalarsBehavior</code>	<code>method),</code>	<code>run()</code>	<code>(AlgorithmVQD method),</code>
<code>706</code>		<code>319</code>	
<code>rstrip()</code>	<code>(FermionOper-</code>	<code>run()</code>	<code>(AlgorithmVQE method),</code>
<code>torList.FactoryCoefficientsLocation</code>	<code>method),</code>	<code>318</code>	<code>318</code>
<code>711</code>		<code>run()</code>	<code>(AlgorithmVQS method),</code>
<code>rstrip()</code>	<code>(FermionOper-</code>	<code>326</code>	
<code>tor.TrotterizeCoefficientsLocation</code>	<code>method),</code>	<code>run()</code>	<code>(AVAS method),</code>
<code>683</code>		<code>1106</code>	
<code>rstrip()</code>	<code>(QubitOperatorList.CompressScalarsBehavior</code>	<code>run()</code>	<code>(ComputeUncompute method),</code>
<code>method),</code>	<code>761</code>	<code>925</code>	<code>925</code>
<code>rstrip()</code>	<code>(QubitOpera-</code>	<code>run()</code>	<code>(ComputeUncomputeFactorizedOverlap method),</code>
<code>torList.ExpandExponentialProductCoefficientsBehavior</code>	<code>method),</code>	<code>962</code>	
<code>method),</code>	<code>766</code>		
<code>rstrip()</code>	<code>(QubitOperatorList.FactoryCoefficientsLocation</code>	<code>run()</code>	<code>(DestructiveSwapTest method),</code>
<code>method),</code>	<code>770</code>	<code>931</code>	<code>931</code>
<code>rstrip()</code>	<code>(QubitOperator.TrotterizeCoefficientsLocation</code>	<code>run()</code>	<code>(DMETRHF method),</code>
<code>method),</code>	<code>734</code>	<code>588</code>	<code>588</code>
<code>rstrip()</code>	<code>(SymmetryOperatorFermionicFac-</code>	<code>run()</code>	<code>(FactorizedOverlap method),</code>
<code>torised.CompressScalarsBehavior</code>	<code>method),</code>	<code>950</code>	<code>950</code>
<code>method),</code>			
<code>rstrip()</code>	<code>(SymmetryOperator-</code>	<code>run()</code>	<code>(FMO method),</code>
<code>Fermionic.TrotterizeCoefficientsLocation</code>	<code>method),</code>	<code>1259</code>	<code>1259</code>
<code>method),</code>			
<code>rstrip()</code>	<code>(QubitOperatorList.CompressScalarsBehavior</code>	<code>run()</code>	<code>(HadamardTest method),</code>
<code>method),</code>	<code>761</code>	<code>918</code>	<code>918</code>
<code>rstrip()</code>	<code>(QubitOpera-</code>	<code>run()</code>	<code>(HadamardTestDerivative method),</code>
<code>torList.ExpandExponentialProductCoefficientsBehavior</code>	<code>method),</code>	<code>980</code>	<code>980</code>
<code>method),</code>			
<code>rstrip()</code>	<code>(QubitOperatorList.FactoryCoefficientsLocation</code>	<code>run()</code>	<code>(HadamardTestDerivativeOverlap method),</code>
<code>method),</code>	<code>770</code>	<code>973</code>	<code>973</code>
<code>rstrip()</code>	<code>(QubitOperator.TrotterizeCoefficientsLocation</code>	<code>run()</code>	<code>(HadamardTestOverlap method),</code>
<code>method),</code>	<code>734</code>	<code>943</code>	<code>943</code>
<code>rstrip()</code>	<code>(SymmetryOperatorFermionicFac-</code>	<code>run()</code>	<code>(ImpurityDMETROHF method),</code>
<code>torised.CompressScalarsBehavior</code>	<code>method),</code>	<code>593</code>	<code>593</code>
<code>method),</code>			
<code>rstrip()</code>	<code>(SymmetryOperator-</code>	<code>run()</code>	<code>(IterativePhaseEstimation method),</code>
<code>Fermionic.TrotterizeCoefficientsLocation</code>	<code>method),</code>	<code>991</code>	<code>991</code>
<code>method),</code>			
<code>rstrip()</code>	<code>(QubitOperatorList.CompressScalarsBehavior</code>	<code>run()</code>	<code>(IterativePhaseEstimationQuantinuum method),</code>
<code>method),</code>	<code>761</code>	<code>996</code>	<code>996</code>
<code>rstrip()</code>	<code>(QubitOpera-</code>	<code>run()</code>	<code>(IterativePhaseEstimationStatevector method),</code>
<code>torList.ExpandExponentialProductCoefficientsBehavior</code>	<code>method),</code>	<code>998</code>	<code>998</code>
<code>method),</code>			
<code>rstrip()</code>	<code>(QubitOperatorList.FactoryCoefficientsLocation</code>	<code>run()</code>	<code>(PauliAveraging method),</code>
<code>method),</code>	<code>770</code>	<code>912</code>	<code>912</code>
<code>rstrip()</code>	<code>(QubitOperator.TrotterizeCoefficientsLocation</code>	<code>run()</code>	<code>(PhaseShift method),</code>
<code>method),</code>	<code>734</code>	<code>986</code>	
<code>rstrip()</code>	<code>(SymmetryOperatorFermionicFac-</code>	<code>run()</code>	<code>(ProjectiveMeasurements method),</code>
<code>torised.CompressScalarsBehavior</code>	<code>method),</code>	<code>1004</code>	
<code>method),</code>			

run() (*SwapFactorizedOverlap method*), 956
 run() (*SwapTest method*), 937
 run_async() (*AlgorithmBayesianQPE method*), 1267
 run_async() (*AlgorithmInfoTheoryQPE method*), 324
 run_async() (*AlgorithmKitaevQPE method*), 325
 run_cascl() (*ChemistryDriverPySCFEmbeddingGammaRHF method*), 1115
 run_cascl() (*ChemistryDriverPySCFEmbeddingGammaRHF_UHF method*), 1125
 run_cascl() (*ChemistryDriverPySCFEmbeddingRHF method*), 1133
 run_cascl() (*ChemistryDriverPySCFEmbeddingROHF method*), 1142
 run_cascl() (*ChemistryDriverPySCFEmbeddinggROHF_UHF method*), 1151
 run_cascl() (*ChemistryDriverPySCFGammaRHF method*), 1160
 run_cascl() (*ChemistryDriverPySCFGammaROHF method*), 1169
 run_cascl() (*ChemistryDriverPySCFIintegrals method*), 1177
 run_cascl() (*ChemistryDriverPySCFMolecularRHF method*), 1186
 run_cascl() (*ChemistryDriverPySCFMolecularRHFQMMMCOSMO method*), 1195
 run_cascl() (*ChemistryDriverPySCFMolecularROHF method*), 1204
 run_cascl() (*ChemistryDriverPySCFMolecularROHFQMMMCOSMO method*), 1214
 run_cascl() (*ChemistryDriverPySCFMolecularUHF method*), 1222
 run_cascl() (*ChemistryDriverPySCFMolecularUHFQMMMCOSMO method*), 1232
 run_cascl() (*ChemistryDriverPySCFMomentumRHF method*), 1235
 run_cascl() (*ChemistryDriverPySCFMomentumROHF method*), 1238
 run_ccsd() (*ChemistryDriverPySCFEmbeddingGammaRHF method*), 1115
 run_ccsd() (*ChemistryDriverPySCFEmbeddingGammaRHF_UHF method*), 1125
 run_ccsd() (*ChemistryDriverPySCFEmbeddingRHF method*), 1133
 run_ccsd() (*ChemistryDriverPySCFEmbeddingROHF method*), 1142
 run_ccsd() (*ChemistryDriverPySCFEmbeddinggROHF_UHF method*), 1151
 run_ccsd() (*ChemistryDriverPySCFGammaRHF method*), 1160
 run_ccsd() (*ChemistryDriverPySCFGammaROHF method*), 1169
 run_ccsd() (*ChemistryDriverPySCFIintegrals method*), 1177
 run_ccsd() (*ChemistryDriverPySCFMolecularRHF method*), 1186
 run_ccsd() (*ChemistryDriverPySCFMolecularRHFQMMMCOSMO method*), 1195
 run_ccsd() (*ChemistryDriverPySCFMolecularROHF method*), 1204
 run_ccsd() (*ChemistryDriverPySCFMolecularROHFQMMMCOSMO method*), 1214
 run_ccsd() (*ChemistryDriverPySCFMolecularUHF method*), 1223
 run_ccsd() (*ChemistryDriverPySCFMolecularUHFQMMMCOSMO method*), 1232
 run_ccsd() (*ChemistryDriverPySCFMolecularUHFQMMMCOSMO method*), 1236
 run_ccsd() (*ChemistryDriverPySCFMomentumRHF method*), 1238
 run_experiment() (*AlgorithmDeterministicQPE method*), 323
 run_hf() (*ChemistryDriverPySCFEmbeddingGammaRHF method*), 1115
 run_hf() (*ChemistryDriverPySCFEmbeddingGammaROHF_UHF method*), 1125
 run_hf() (*ChemistryDriverPySCFEmbeddingRHF method*), 1134
 run_hf() (*ChemistryDriverPySCFEmbeddingROHF method*), 1143
 run_hf() (*ChemistryDriverPySCFEmbeddinggROHF_UHF method*), 1152
 run_hf() (*ChemistryDriverPySCFGammaRHF method*), 1161
 run_hf() (*ChemistryDriverPySCFGammaROHF method*), 1170
 run_hf() (*ChemistryDriverPySCFIintegrals method*), 1177
 run_hf() (*ChemistryDriverPySCFMolecularRHF method*), 1186
 run_hf() (*ChemistryDriverPySCFMolecularRHFQMMMCOSMO method*), 1195
 run_hf() (*ChemistryDriverPySCFMolecularROHF method*), 1204
 run_hf() (*ChemistryDriverPySCFMolecularROHFQMMMCOSMO method*), 1214
 run_hf() (*ChemistryDriverPySCFMolecularUHF method*), 1223
 run_hf() (*ChemistryDriverPySCFMolecularUHFQMMMCOSMO method*), 1232
 run_hf() (*ChemistryDriverPySCFMomentumRHF method*), 1236
 run_hf() (*ChemistryDriverPySCFMomentumROHF method*), 1239
 run_mitex() (*PauliAveraging method*), 912
 run_mitres() (*PauliAveraging method*), 912
 run_mp2() (*ChemistryDriverPySCFEmbeddingGammaRHF method*), 1116
 run_mp2() (*ChemistryDriverPySCFEmbeddingGammaROHF_UHF method*), 1125

run_mp2 () (*ChemistryDriverPySCFEmbeddingRHF method*), 1134
 run_mp2 () (*ChemistryDriverPySCFEmbeddingROHF method*), 1143
 run_mp2 () (*ChemistryDriverPySCFEmbeddingROHF_UHF method*), 1152
 run_mp2 () (*ChemistryDriverPySCFGammaRHF method*), 1161
 run_mp2 () (*ChemistryDriverPySCFGammaROHF method*), 1170
 run_mp2 () (*ChemistryDriverPySCFIintegrals method*), 1177
 run_mp2 () (*ChemistryDriverPySCFMolecularRHF method*), 1186
 run_mp2 () (*ChemistryDriverPySCFMolecularRHFQM-MMCOSMO method*), 1196
 run_mp2 () (*ChemistryDriverPySCFMolecularROHF method*), 1204
 run_mp2 () (*ChemistryDriverPySCFMolecularROHFQM-MMCOSMO method*), 1214
 run_mp2 () (*ChemistryDriverPySCFMolecularUHF method*), 1223
 run_mp2 () (*ChemistryDriverPySCFMolecularUHFQM-MMCOSMO method*), 1232
 run_mp2 () (*ChemistryDriverPySCFMomentumRHF method*), 1236
 run_mp2 () (*ChemistryDriverPySCFMomentumROHF method*), 1239
 run_one () (*DMETRHF method*), 588
 run_rhf () (*in module inquanto.express*), 601
 run_rhf () (*PySCFChemistryRestrictedIntegralOperator method*), 1252
 run_rohf () (*in module inquanto.express*), 601
 run_time_evolution () (*in module inquanto.express*), 602
 run_uhf () (*PySCFChemistryUnrestrictedIntegralOperator method*), 1257
 run_vqe () (*in module inquanto.express*), 602
 RUNTIME (*CacheLevels attribute*), 580

S

save_csv () (*GeometryMolecular method*), 611
 save_csv () (*GeometryPeriodic method*), 622
 save_h5 () (*ChemistryRestrictedIntegralOperator method*), 657
 save_h5 () (*ChemistryRestrictedIntegralOperatorCompact method*), 662
 save_h5 () (*ChemistryUnrestrictedIntegralOperator method*), 667
 save_h5 () (*ChemistryUnrestrictedIntegralOperatorCompact method*), 672
 save_h5 () (*FermionSpace method*), 1032
 save_h5 () (*FermionSpaceBrillouin method*), 1040
 save_h5 () (*FermionSpaceSupercell method*), 1049

save_h5 () (*FermionState method*), 1063
 save_h5 () (*FermionString method*), 1069
 save_h5 () (*ParaFermionSpace method*), 1053
 save_h5 () (*PySCFChemistryRestrictedIntegralOperator method*), 1252
 save_h5 () (*PySCFChemistryUnrestrictedIntegralOperator method*), 1257
 save_h5 () (*QubitSpace method*), 1054
 save_h5 () (*QubitState method*), 1078
 save_h5 () (*QubitString method*), 1084
 save_h5 () (*RestrictedOneBodyRDM method*), 796
 save_h5 () (*RestrictedTwoBodyRDM method*), 798
 save_h5 () (*State method*), 1093
 save_h5 () (*StateString method*), 1098
 save_h5 () (*UnrestrictedOneBodyRDM method*), 903
 save_h5 () (*UnrestrictedTwoBodyRDM method*), 904
 save_h5_system () (*in module inquanto.express*), 603
 save_json () (*GeometryMolecular method*), 611
 save_json () (*GeometryPeriodic method*), 622
 save_xyz () (*GeometryMolecular method*), 611
 save_xyz () (*GeometryPeriodic method*), 623
 save_zmatrix () (*GeometryMolecular method*), 611
 scan_bond_angle () (*GeometryMolecular method*), 612
 scan_bond_angle () (*GeometryPeriodic method*), 623
 scan_bond_angle_by_group () (*GeometryMolecular method*), 612
 scan_bond_angle_by_group () (*GeometryPeriodic method*), 623
 scan_bond_length () (*GeometryMolecular method*), 612
 scan_bond_length () (*GeometryPeriodic method*), 623
 scan_bond_length_by_group () (*GeometryMolecular method*), 612
 scan_bond_length_by_group () (*GeometryPeriodic method*), 624
 scan_dihedral_angle () (*GeometryMolecular method*), 613
 scan_dihedral_angle () (*GeometryPeriodic method*), 624
 scan_dihedral_angle_by_group () (*GeometryMolecular method*), 613
 scan_dihedral_angle_by_group () (*GeometryPeriodic method*), 624
 SCEOMMatrixComputable (*class in inquanto.computables.composite*), 562
 ScipyIVPIntegrator (*class in inquanto.minimizers*), 649
 ScipyODEIntegrator (*class in inquanto.minimizers*), 651
 searchsorted () (*ComputableNDArray method*), 515
 select () (*FermionSpace method*), 1032
 select () (*FermionSpaceBrillouin method*), 1040
 select () (*FermionSpaceSupercell method*), 1049
 set () (*SymbolDict method*), 574

set_block() (*RestrictedOneBodyRDM method*), 797
 set_block() (*UnrestrictedOneBodyRDM method*), 903
 set_checkfile() (*ChemistryDriverPySCFEmbeddingGammaRHF method*), 1116
 set_checkfile() (*ChemistryDriverPySCFEmbeddingGammaROHF_UHF method*), 1125
 set_checkfile() (*ChemistryDriverPySCFEmbeddinggRHF method*), 1134
 set_checkfile() (*ChemistryDriverPySCFEmbeddinggROHF method*), 1143
 set_checkfile() (*ChemistryDriverPySCFEmbeddinggROHF_UHF method*), 1152
 set_checkfile() (*ChemistryDriverPySCFGammaRHF method*), 1161
 set_checkfile() (*ChemistryDriverPySCFGammaROHF method*), 1170
 set_checkfile() (*ChemistryDriverPySCFIintegrals method*), 1177
 set_checkfile() (*ChemistryDriverPySCFMolecularRHF method*), 1186
 set_checkfile() (*ChemistryDriverPySCFMolecularRHFQMMMCOSMO method*), 1196
 set_checkfile() (*ChemistryDriverPySCFMolecularROHF method*), 1204
 set_checkfile() (*ChemistryDriverPySCFMolecularROHFQMMMCOSMO method*), 1214
 set_checkfile() (*ChemistryDriverPySCFMolecularUHF method*), 1223
 set_checkfile() (*ChemistryDriverPySCFMolecularUHFQMMMCOSMO method*), 1233
 set_diis_space_dimension() (*ChemistryDriverPySCFMolecularROHFQMMMCOSMO method*), 1214
 set_diis_space_dimension() (*ChemistryDriverPySCFMolecularUHF method*), 1223
 set_diis_space_dimension() (*ChemistryDriverPySCFMolecularUHFQMMMCOSMO method*), 1233
 set_global_phase() (*FermionOperator method*), 697
 set_global_phase() (*FermionState method*), 1063
 set_global_phase() (*QubitOperator method*), 750
 set_global_phase() (*QubitState method*), 1079
 set_global_phase() (*State method*), 1093
 set_global_phase() (*SymmetryOperatorFermionic method*), 816
 set_global_phase() (*SymmetryOperatorPauli method*), 863
 set_groups() (*GeometryMolecular method*), 613
 set_groups() (*GeometryPeriodic method*), 624
 set_init_orbitals() (*ChemistryDriverPySCFEmbeddingGammaRHF method*), 1116
 set_init_orbitals() (*ChemistryDriverPySCFEmbeddingGammaROHF_UHF method*), 1125
 set_init_orbitals() (*ChemistryDriverPySCFEmbeddinggRHF method*), 1134
 set_init_orbitals() (*ChemistryDriverPySCFEmbeddinggROHF method*), 1143
 set_init_orbitals() (*ChemistryDriverPySCFEmbeddinggROHF_UHF method*), 1152
 set_init_orbitals() (*ChemistryDriverPySCFGammaRHF method*), 1161
 set_init_orbitals() (*ChemistryDriverPySCFGammaROHF method*), 1170
 set_init_orbitals() (*ChemistryDriverPySCFIintegrals method*), 1177
 set_init_orbitals() (*ChemistryDriverPySCFMolecularRHF method*), 1186
 set_init_orbitals() (*ChemistryDriverPySCFMolecularRHFQMMMCOSMO method*), 1196
 set_init_orbitals() (*ChemistryDriverPySCFMolecularROHF method*), 1205
 set_init_orbitals() (*ChemistryDriverPySCFMolecularROHFQMMMCOSMO method*), 1214
 set_init_orbitals() (*ChemistryDriverPySCFMolecularUHF method*), 1223
 set_init_orbitals() (*ChemistryDriverPySCFMolecularUHFQMMMCOSMO method*), 1233
 set_init_orbitals() (*ChemistryDriverPySCFMomentumRHF method*), 1236
 set_init_orbitals() (*ChemistryDriverPySCFMomentumROHF method*), 1239
 set_level_shift() (*ChemistryDriverPySCFEmbeddingGammaRHF method*), 1116
 set_level_shift() (*ChemistryDriverPySCFEmbeddingGammaROHF method*), 1205

dingGammaROHF_UHF method), 1125

set_level_shift() (ChemistryDriverPySCFEmbeddingRHF method), 1134

set_level_shift() (ChemistryDriverPySCFEmbeddingROHF method), 1143

set_level_shift() (ChemistryDriverPySCFEmbeddingROHF_UHF method), 1152

set_level_shift() (ChemistryDriverPySCFGammaRHF method), 1161

set_level_shift() (ChemistryDriverPySCFGammaROHF method), 1170

set_level_shift() (ChemistryDriverPySCFIIntegrals method), 1178

set_level_shift() (ChemistryDriverPySCFMolecularRHF method), 1186

set_level_shift() (ChemistryDriverPySCFMolecularRHFQMMMCOSMO method), 1196

set_level_shift() (ChemistryDriverPySCFMolecularROHF method), 1205

set_level_shift() (ChemistryDriverPySCFMolecularROHFQMMMCOSMO method), 1215

set_level_shift() (ChemistryDriverPySCFMolecularUHF method), 1223

set_level_shift() (ChemistryDriverPySCFMolecularUHFQMMMCOSMO method), 1233

set_max_scf_cycles() (ChemistryDriverPySCFEmbeddingGammaRHF method), 1116

set_max_scf_cycles() (ChemistryDriverPySCFEmbeddingGammaROHF_UHF method), 1126

set_max_scf_cycles() (ChemistryDriverPySCFEmbeddingRHF method), 1135

set_max_scf_cycles() (ChemistryDriverPySCFEmbeddingROHF method), 1144

set_max_scf_cycles() (ChemistryDriverPySCFEmbeddingROHF_UHF method), 1152

set_max_scf_cycles() (ChemistryDriverPySCFGammaRHF method), 1161

set_max_scf_cycles() (ChemistryDriverPySCFGammaROHF method), 1171

set_max_scf_cycles() (ChemistryDriverPySCFIIntegrals method), 1178

set_max_scf_cycles() (ChemistryDriverPySCFMolecularRHF method), 1187

set_max_scf_cycles() (ChemistryDriverPySCFMolecularRHFQMMMCOSMO method), 1196

set_max_scf_cycles() (ChemistryDriverPySCFMolecularROHF method), 1205

set_max_scf_cycles() (ChemistryDriverPySCFMolecularROHFQMMMCOSMO method), 1215

set_max_scf_cycles() (ChemistryDriverPySCFMolecularUHF method), 1224

set_max_scf_cycles() (ChemistryDriverPySCFMolecularUHFQMMMCOSMO method), 1233

set_subgroups() (GeometryMolecular method), 613

set_subgroups() (GeometryPeriodic method), 625

setfield() (ComputableNDArray method), 515

setflags() (ComputableNDArray method), 516

shape (CompactTwoBodyIntegralsS4 property), 674

shape (CompactTwoBodyIntegralsS8 property), 676

shape (ComputableNDArray attribute), 517

shift_rp_permutation() (FermionSpaceSupercell method), 1049

simplify() (FermionOperator method), 698

simplify() (FermionOperatorList method), 718

simplify() (FermionState method), 1063

simplify() (QubitOperator method), 750

simplify() (QubitOperatorList method), 783

simplify() (QubitState method), 1079

simplify() (State method), 1093

simplify() (SymmetryOperatorFermionic method), 816

simplify() (SymmetryOperatorFermionicFactorised method), 838

simplify() (SymmetryOperatorPauli method), 863

simplify() (SymmetryOperatorPauliFactorised method), 896

single_term (FermionState property), 1064

single_term (QubitState property), 1079

single_term (State property), 1093

size (ComputableNDArray attribute), 518

size_unit (Cache property), 579

size_unit (ProtocolCache property), 1012

solve() (DMETRHFFragment method), 589

solve() (DMETRHFFragmentActive method), 590

solve() (DMETRHFFragmentDirect method), 591

solve() (DMETRHFFragmentPySCFActive method), 1240

solve() (DMETRHFFragmentPySCFCCSD method), 1241

solve() (DMETRHFFragmentPySCFFCI method), 1242

solve() (DMETRHFFragmentPySCFMP2 method), 1243

solve() (DMETRHFFragmentPySCFRHF method), 1244

solve() (DMETRHFFragmentUCCSDVQE method), 592

solve() (FMOFragment method), 1260

solve() (FMOFragmentPySCFActive method), 1261

solve() (FMOFragmentPySCFCCSD method), 1262

solve() (FMOFragmentPySCFMP2 method), 1263

solve() (FMOFragmentPySCFRHF method), 1264

solve() (ImpurityDMETROHFFragment method), 594

solve() (ImpurityDMETROHFFragmentActive method), 595

solve() (ImpurityDMETROHFFragmentED method), 595

solve() (ImpurityDMETROHFFragmentPySCFActive method), 1246

solve() (ImpurityDMETROHFFragmentPySCFCCSD method), 1246

solve() (ImpurityDMETROHFFragmentPySCFFCI method), 1247

solve() (*ImpurityDMETROHFFragmentPySCFMP2 method*), 1247
 solve() (*ImpurityDMETROHFFragmentPySCFROHF method*), 1248
 solve() (*ImpurityDMETROHFFragmentWithoutRDM method*), 596
 solve() (*NaiveEulerIntegrator method*), 648
 solve() (*ScipyIVPIntegrator method*), 650
 solve() (*ScipyODEIntegrator method*), 652
 solve_active() (*DMETRHFFragmentActive method*), 590
 solve_active() (*DMETRHFFragmentPySCFActive method*), 1240
 solve_active() (*DMETRHFFragmentUCCSDVQE method*), 592
 solve_active() (*ImpurityDMETROHFFragmentActive method*), 595
 solve_active() (*ImpurityDMETROHFFragmentPySCFActive method*), 1246
 solve_final() (*FMOFragment method*), 1260
 solve_final() (*FMOFragmentPySCFActive method*), 1261
 solve_final() (*FMOFragmentPySCFCCSD method*), 1262
 solve_final() (*FMOFragmentPySCFMP2 method*), 1263
 solve_final() (*FMOFragmentPySCFRHF method*), 1264
 solve_final_active() (*FMOFragmentPySCFActive method*), 1261
 sort() (*ComputableList method*), 491
 sort() (*ComputableNDArray method*), 518
 SPAM (class in *in quanto.protocols*), 1009
 SparseStatevectorProtocol (class in *in quanto.protocols*), 962
 SPIN_ALPHA (*FermionSpace attribute*), 1017
 SPIN_ALPHA (*FermionSpaceBrillouin attribute*), 1034
 SPIN_ALPHA (*FermionSpaceSupercell attribute*), 1041
 SPIN_BETA (*FermionSpace attribute*), 1017
 SPIN_BETA (*FermionSpaceBrillouin attribute*), 1034
 SPIN_BETA (*FermionSpaceSupercell attribute*), 1041
 SPIN_DOWN (*FermionSpace attribute*), 1018
 SPIN_DOWN (*FermionSpaceBrillouin attribute*), 1034
 SPIN_DOWN (*FermionSpaceSupercell attribute*), 1041
 SPIN_UP (*FermionSpace attribute*), 1018
 SPIN_UP (*FermionSpaceBrillouin attribute*), 1034
 SPIN_UP (*FermionSpaceSupercell attribute*), 1041
 SpinlessNBodyPDMArrayRealComputable (class in *in quanto.computables.composite*), 565
 SpinlessNBodyRDMArrayRealComputable (class in *in quanto.computables.composite*), 567
 split() (*FermionOperator method*), 698
 split() (*FermionOperatorList method*), 719
 split() (*FermionOperatorList.CompressScalarsBehavior method*), 706
 split() (*FermionOperatorList.FactoryCoefficientsLocation method*), 711
 split() (*FermionOperator.TrotterizeCoefficientsLocation method*), 683
 split() (*FermionState method*), 1064
 split() (*QubitOperator method*), 750
 split() (*QubitOperatorList method*), 783
 split() (*QubitOperatorList.CompressScalarsBehavior method*), 761
 split() (*QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method*), 766
 split() (*QubitOperatorList.FactoryCoefficientsLocation method*), 770
 split() (*QubitOperator.TrotterizeCoefficientsLocation method*), 734
 split() (*QubitState method*), 1079
 split() (*State method*), 1093
 split() (*SymmetryOperatorFermionic method*), 816
 split() (*SymmetryOperatorFermionicFactorised method*), 838
 split() (*SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method*), 825
 split() (*SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method*), 830
 split() (*SymmetryOperatorFermionic.TrotterizeCoefficientsLocation method*), 802
 split() (*SymmetryOperatorPauli method*), 863
 split() (*SymmetryOperatorPauliFactorised method*), 897
 split() (*SymmetryOperatorPauliFactorised.CompressScalarsBehavior method*), 873
 split() (*SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method*), 878
 split() (*SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method*), 883
 split() (*SymmetryOperatorPauli.TrotterizeCoefficientsLocation method*), 847
 split_hamiltonian (*HamiltonianVariationalAnsatz property*), 450
 split_totally_commuting_set() (*QubitOperatorList method*), 783
 split_totally_commuting_set() (*SymmetryOperatorPauliFactorised method*), 897
 splitlines() (*FermionOperatorList.CompressScalarsBehavior method*), 706

```

splitlines() (FermionOpera- torList.ExpandExponentialProductCoefficientsBehavior
torList.FactoryCoefficientsLocation method), 766
711
splitlines() (FermionOpera- torList.TrotterizeCoefficientsLocation method), 771
683
splitlines() (QubitOpera- torList.CompressScalarsBehavior method), 734
761
splitlines() (QubitOpera- torList.ExpandExponentialProductCoefficientsBehavior method), 825
766
splitlines() (QubitOpera- torList.FactoryCoefficientsLocation method), 830
771
splitlines() (QubitOpera- torList.TrotterizeCoefficientsLocation method), 802
734
splitlines() (SymmetryOperatorFermionicFac- torised.CompressScalarsBehavior method), 874
825
splitlines() (SymmetryOperatorFermionicFac- torised.FactoryCoefficientsLocation method), 884
830
splitlines() (SymmetryOperatorFermionicFac- torised.ExpandExponentialProductCoefficientsBehavior
method), 879
830
splitlines() (SymmetryOperatorFermionicFactory
torised.TrotterizeCoefficientsLocation method), 847
802
splitlines() (SymmetryOperatorPauliFac- torised.CompressScalarsBehavior method), 1085
874
splitlines() (SymmetryOperatorPauliFac- torised.ExpandExponentialProductCoefficientsBehavior
method), 464
879
splitlines() (SymmetryOperatorPauliFac- torised.FactoryCoefficientsLocation method), 465
883
splitlines() (SymmetryOperatorPauliFac- torised.ExpandExponentialProductCoefficientsBehavior
method), 467
883
splitlines() (SymmetryOperatorPauliFac- torised.FactoryCoefficientsLocation method), 469
847
splitlines() (SymmetryOperatorPauliFac- torised.TrotterizeCoefficientsLocation method), 471
847
squeeze() (ComputableNDArray method), 338
519
start() (Timer method), 343
583
startswith() (FermionOpera- torList.CompressScalarsBehavior method), 377
707
startswith() (FermionOpera- torList.FactoryCoefficientsLocation method), 382
711
startswith() (FermionOpera- torList.TrotterizeCoefficientsLocation method), 394
684
startswith() (QubitOpera- torList.CompressScalarsBehavior method), 366
761
startswith() (QubitOpera-

```

erty), 405
 state_circuit (*FermionSpaceAnsatzUCCSD* property), 361
 state_circuit (*FermionSpaceAnsatzUCCSDSinglet* property), 410
 state_circuit (*FermionSpaceStateExp* property), 355
 state_circuit (*FermionSpaceStateExpChemicallyAware* property), 372
 state_circuit (*GeneralAnsatz* property), 333
 state_circuit (*HamiltonianVariationalAnsatz* property), 450
 state_circuit (*HardwareEfficientAnsatz* property), 461
 state_circuit (*LayeredAnsatz* property), 456
 state_circuit (*MultiConfigurationAnsatz* property), 416
 state_circuit (*MultiConfigurationState* property), 421
 state_circuit (*MultiConfigurationStateBox* property), 427
 state_circuit (*RealGeneralizedBasisRotationAnsatz* property), 432
 state_circuit (*RealRestrictedBasisRotationAnsatz* property), 437
 state_circuit (*RealUnrestrictedBasisRotationAnsatz* property), 443
 state_circuit (*TrotterAnsatz* property), 349
 state_expectation() (*QubitOperator* method), 750
 state_expectation() (*QubitOperatorString* method), 793
 state_expectation() (*SymmetryOperatorPauli* method), 863
 state_map() (*QubitMapping* class method), 628
 state_map() (*QubitMappingBravyiKitaev* class method), 635
 state_map() (*QubitMappingJordanWigner* method), 631
 state_map() (*QubitMappingParaparticular* method), 642
 state_map() (*QubitMappingParity* class method), 638
 state_map_conventional() (*QubitMapping* class method), 629
 state_map_conventional() (*QubitMappingBravyiKitaev* class method), 636
 state_map_conventional() (*QubitMappingJordanWigner* class method), 632
 state_map_conventional() (*QubitMappingParaparticular* class method), 643
 state_map_conventional() (*QubitMappingParity* class method), 639
 state_map_matrix() (*QubitMapping* class method), 629
 state_map_matrix() (*QubitMappingBravyiKitaev* class method), 636
 state_map_matrix() (*QubitMappingJordanWigner* static method), 632
 state_map_matrix() (*QubitMappingParaparticular* static method), 643
 state_map_matrix() (*QubitMappingParity* static method), 643
 state_map_matrix() (*QubitMappingParity* static method), 639
 state_symbols (*CircuitAnsatz* property), 338
 state_symbols (*ComposedAnsatz* property), 344
 state_symbols (*FermionSpaceAnsatzChemicallyAwareUCCSD* property), 377
 state_symbols (*FermionSpaceAnsatzkUpCCGD* property), 382
 state_symbols (*FermionSpaceAnsatzkUpCCGSD* property), 388
 state_symbols (*FermionSpaceAnsatzkUpCCGSDSinglet* property), 394
 state_symbols (*FermionSpaceAnsatzUCCD* property), 366
 state_symbols (*FermionSpaceAnsatzUCCGD* property), 399
 state_symbols (*FermionSpaceAnsatzUCCGSD* property), 405
 state_symbols (*FermionSpaceAnsatzUCCSD* property), 361
 state_symbols (*FermionSpaceAnsatzUCCSDSinglet* property), 410
 state_symbols (*FermionSpaceStateExp* property), 355
 state_symbols (*FermionSpaceStateExpChemicallyAware* property), 372
 state_symbols (*GeneralAnsatz* property), 333
 state_symbols (*HamiltonianVariationalAnsatz* property), 450
 state_symbols (*HardwareEfficientAnsatz* property), 461
 state_symbols (*LayeredAnsatz* property), 456
 state_symbols (*MultiConfigurationAnsatz* property), 416
 state_symbols (*MultiConfigurationState* property), 422
 state_symbols (*MultiConfigurationStateBox* property), 427
 state_symbols (*QubitState* property), 1079
 state_symbols (*RealGeneralizedBasisRotationAnsatz* property), 432
 state_symbols (*RealRestrictedBasisRotationAnsatz* property), 438
 state_symbols (*RealUnrestrictedBasisRotationAnsatz* property), 443
 state_symbols (*TrotterAnsatz* property), 349
 StateString (class in *inquanto.states*), 1096
 std() (*ComputableNDArray* method), 519
 stop() (*Timer* method), 583
 strides (*ComputableNDArray* attribute), 520
 string_class (*FermionState* attribute), 1064
 string_class (*QubitState* attribute), 1079
 string_class (*State* attribute), 1093
 strip() (*FermionOperatorList.CompressScalarsBehavior* method), 707

```

strip() (FermionOperatorList.FactoryCoefficientsLocation method), 711
strip() (FermionOperator.TrotterizeCoefficientsLocation method), 684
strip() (QubitOperatorList.CompressScalarsBehavior method), 761
strip() (QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method), 766
strip() (QubitOperatorList.FactoryCoefficientsLocation method), 771
strip() (QubitOperator.TrotterizeCoefficientsLocation method), 734
strip() (SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method), 825
strip() (SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method), 830
strip() (SymmetryOperatorFermionic.TrotterizeCoefficientsLocation method), 802
strip() (SymmetryOperatorPauliFactorised.CompressScalarsBehavior method), 874
strip() (SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method), 879
strip() (SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method), 884
strip() (SymmetryOperatorPauli.TrotterizeCoefficientsLocation method), 847
sublist() (FermionOperatorList method), 719
sublist() (QubitOperatorList method), 784
sublist() (SymmetryOperatorFermionicFactorised method), 838
sublist() (SymmetryOperatorPauliFactorised method), 897
subs() (CircuitAnsatz method), 338
subs() (ComposedAnsatz method), 344
subs() (FermionOperator method), 698
subs() (FermionOperatorList method), 719
subs() (FermionSpaceAnsatzChemicallyAwareUCCSD method), 377
subs() (FermionSpaceAnsatzkUpCCGD method), 382
subs() (FermionSpaceAnsatzkUpCCGSD method), 388
subs() (FermionSpaceAnsatzkUpCCGSDSinglet method), 394
subs() (FermionSpaceAnsatzUCCD method), 366
subs() (FermionSpaceAnsatzUCCGD method), 399
subs() (FermionSpaceAnsatzUCCGSD method), 405
subs() (FermionSpaceAnsatzUCCSD method), 361
subs() (FermionSpaceAnsatzUCCSDSinglet method), 410
subs() (FermionSpaceStateExp method), 355
subs() (FermionSpaceStateExpChemicallyAware method), 372
subs() (FermionState method), 1064
subs() (GeneralAnsatz method), 333
subs() (HamiltonianVariationalAnsatz method), 450
subs() (HardwareEfficientAnsatz method), 461
subs() (LayeredAnsatz method), 456
subs() (MultiConfigurationAnsatz method), 416
subs() (MultiConfigurationState method), 422
subs() (MultiConfigurationStateBox method), 427
subs() (QubitOperator method), 751
subs() (QubitOperatorList method), 784
subs() (QubitState method), 1079
subs() (RealGeneralizedBasisRotationAnsatz method), 432
subs() (RealRestrictedBasisRotationAnsatz method), 438
subs() (RealUnrestrictedBasisRotationAnsatz method), 443
subs() (State method), 1093
subs() (SymmetryOperatorFermionic method), 816
subs() (SymmetryOperatorFermionicFactorised method), 838
subs() (SymmetryOperatorPauli method), 863
subs() (SymmetryOperatorPauliFactorised method), 897
subs() (TrotterAnsatz method), 349
sum() (ComputableNDArray method), 521
supported_groups() (PointGroup static method), 1100
swap_rp_permutation() (FermionSpaceSupercell method), 1049
swapaxes() (ComputableNDArray method), 521
swapcase() (FermionOperatorList.CompressScalarsBehavior method), 707
swapcase() (FermionOperatorList.FactoryCoefficientsLocation method), 711
swapcase() (FermionOperator.TrotterizeCoefficientsLocation method), 684
swapcase() (QubitOperatorList.CompressScalarsBehavior method), 761
swapcase() (QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method), 766
swapcase() (QubitOperatorList.FactoryCoefficientsLocation method), 771
swapcase() (QubitOperator.TrotterizeCoefficientsLocation method), 734
swapcase() (SymmetryOperatorFermionicFac-
```

torised.CompressScalarsBehavior *method),*
826
swapcase() (*SymmetryOperatorFermionicFac-*
torised.FactoryCoefficientsLocation *method),*
830
swapcase() (*SymmetryOperator-*
Fermionic.TrotterizeCoefficientsLocation
method), **802**
swapcase() (*SymmetryOperatorPauliFac-*
torised.CompressScalarsBehavior *method),*
874
swapcase() (*SymmetryOperatorPauliFac-*
torised.ExpandExponentialProductCoefficientsBehavior
method), **879**
swapcase() (*SymmetryOperatorPauliFac-*
torised.FactoryCoefficientsLocation *method),*
884
swapcase() (*SymmetryOperatorPauliFac-*
torised.TrotterizeCoefficientsLocation *method),*
847
SwapFactorizedOverlap (*class in inquanto.protocols*),
950
SwapTest (*class in inquanto.protocols*), **931**
sx_insertion (*IcebergOptions attribute*), **1013**
symbol_substitution() (*CircuitAnsatz method*), **338**
symbol_substitution() (*ComposedAnsatz method*),
344
symbol_substitution() (*FermionOperator method*),
698
symbol_substitution() (*FermionOperatorList*
method), **719**
symbol_substitution() (*FermionSpaceAnsatzChemi-*
callyAwareUCCSD method), **377**
symbol_substitution() (*FermionSpaceAnsatzUpC-*
CGD method), **383**
symbol_substitution() (*FermionSpaceAnsatzUpC-*
CGSD method), **388**
symbol_substitution() (*FermionSpaceAnsatzUpC-*
CGSDSinglet method), **394**
symbol_substitution() (*FermionSpaceAnsatzUCCD*
method), **367**
symbol_substitution() (*FermionSpaceAnsatzUC-*
CGD method), **400**
symbol_substitution() (*FermionSpaceAnsatzUC-*
CGSD method), **405**
symbol_substitution() (*FermionSpaceAnsatzUCCSD*
method), **361**
symbol_substitution() (*FermionSpaceAnsatzUCCS-*
DSinglet method), **411**
symbol_substitution() (*FermionSpaceStateExp*
method), **356**
symbol_substitution() (*FermionSpaceStateExp*
ChemicallyAware method), **372**
symbol_substitution() (*FermionState method*), **1064**
symbol_substitution() (*GeneralAnsatz method*), **333**
symbol_substitution() (*HamiltonianVariation-*
alAnsatz method), **451**
symbol_substitution() (*HardwareEfficientAnsatz*
method), **461**
symbol_substitution() (*LayeredAnsatz method*), **456**
symbol_substitution() (*MultiConfigurationAnsatz*
method), **416**
symbol_substitution() (*MultiConfigurationState*
method), **422**
symbol_substitution() (*MultiConfigurationStateBox*
method), **427**
symbol_substitution() (*QubitOperator method*), **751**
symbol_substitution() (*QubitOperatorList method*),
784
symbol_substitution() (*QubitState method*), **1079**
symbol_substitution() (*RealGeneralizedBasisRotati-*
onAnsatz method), **432**
symbol_substitution() (*RealRestrictedBasisRotatio-*
nAnsatz method), **438**
symbol_substitution() (*RealUnrestrictedBasisRotati-*
onAnsatz method), **443**
symbol_substitution() (*State method*), **1094**
symbol_substitution() (*SymmetryOperatorFermionic*
method), **817**
symbol_substitution() (*SymmetryOperatorFermion-icFactorised method*), **839**
symbol_substitution() (*SymmetryOperatorPauli*
method), **864**
symbol_substitution() (*SymmetryOperatorPauliFac-*
torised method), **898**
symbol_substitution() (*TrotterAnsatz method*), **349**
SymbolDict (*class in inquanto.core*), **572**
SymbolicProtocol (*class in inquanto.protocols*), **965**
symbols (*ExpectationValueBraDerivativeImag attribute*),
465
symbols (*ExpectationValueBraDerivativeReal attribute*),
467
symbols (*ExpectationValueDerivative attribute*), **469**
symbols (*ExpectationValueKetDerivativeImag attribute*),
471
symbols (*ExpectationValueKetDerivativeReal attribute*),
472
symbols (*MetricTensorImag attribute*), **476**
symbols (*MetricTensorReal attribute*), **477**
symbols (*SymbolDict property*), **574**
symbols (*SymbolSet property*), **578**
SymbolSet (*class in inquanto.core*), **574**
symmetry_operators (*TapererZ2 attribute*), **1101**
symmetry_operators_z2() (*FermionSpace static*
method), **1032**
symmetry_operators_z2() (*ParaFermionSpace static*
method), **1053**
symmetry_operators_z2() (*QubitSpace static*

method), 1054
symmetry_operators_z2_in_sector() (Fermion- Space method), 1033
symmetry_operators_z2_in_sector() (ParaFermionSpace static method), 1053
symmetry_operators_z2_in_sector() (QubitSpace static method), 1055
symmetry_sector() (SymmetryOperatorFermionic method), 817
symmetry_sector() (SymmetryOperatorFermionicFactorised method), 839
symmetry_sector() (SymmetryOperatorPauli method), 864
symmetry_sector() (SymmetryOperatorPauliFactorised method), 898
symmetry_sectors (TapererZ2 attribute), 1101
SymmetryOperatorFermionic (class in in- quanto.operators), 798
SymmetryOperatorFermionicFactorised (class in inquanto.operators), 821
SymmetryOperatorFermionicFactorised.CompressScalarsBehavior (class in inquanto.operators), 821
SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation (class in inquanto.operators), 826
SymmetryOperatorFermionic.TrotterizeCoefficientsLocation (class in inquanto.operators), 798
SymmetryOperatorPauli (class in inquanto.operators), 842
SymmetryOperatorPauliFactorised (class in in- quanto.operators), 869
SymmetryOperatorPauliFactorised.CompressScalarsBehavior (class in inquanto.operators), 869
SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior (class in inquanto.operators), 874
SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation (class in inquanto.operators), 879
SymmetryOperatorPauli.TrotterizeCoefficientsLocation (class in inquanto.operators), 843
sympify() (FermionOperator method), 698
sympify() (FermionOperatorList method), 720
sympify() (FermionState method), 1064
sympify() (QubitOperator method), 751
sympify() (QubitOperatorList method), 785
sympify() (QubitState method), 1080
sympify() (State method), 1094
sympify() (SymmetryOperatorFermionic method), 817
sympify() (SymmetryOperatorFermionicFactorised method), 839
sympify() (SymmetryOperatorPauli method), 864
sympify() (SymmetryOperatorPauliFactorised method), 898
symplectic_representation() (QubitOperator method), 751
symplectic_representation() (SymmetryOperator-

Pauli method), 864
syndrome_interval (IcebergOptions attribute), 1013

T

T (ComputableNDArray attribute), 492
take() (ComputableNDArray method), 521
taperable_qubits (TapererZ2 attribute), 1101
tapered_operator() (TapererZ2 method), 1101
tapered_state() (TapererZ2 method), 1102
TapererZ2 (class in inquanto.symmetry), 1100
TapererZ2.XOperatorMinimalError, 1101
tapering_unitary() (TapererZ2 method), 1102
terms (FermionOperator property), 698
terms (FermionState property), 1065
terms (QubitOperator property), 752
terms (QubitState property), 1080
terms (State property), 1094
terms (SymmetryOperatorFermionic property), 817
terms (SymmetryOperatorPauli property), 865
Timer (class in inquanto.core), 583
TimerWith (class in inquanto.core), 583
title() (FermionOperatorList.CompressScalarsBehavior method), 707
title() (FermionOperatorList.FactoryCoefficientsLocation method), 711
title() (FermionOperator.TrotterizeCoefficientsLocation method), 684
title() (QubitOperatorList.CompressScalarsBehavior method), 761
title() (QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method), 766
title() (QubitOperatorList.FactoryCoefficientsLocation method), 771
title() (QubitOperator.TrotterizeCoefficientsLocation method), 734
title() (SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method), 826
title() (SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method), 830
title() (SymmetryOperatorFermionic.TrotterizeCoefficientsLocation method), 802
title() (SymmetryOperatorPauliFactorised.CompressScalarsBehavior method), 874
title() (SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method), 879
title() (SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method), 884

title() (SymmetryOperator method), 334
Pauli.TrotterizeCoefficientsLocation method), 847
 to_angstrom() (*GeometryMolecular* method), 614
 to_angstrom() (*GeometryPeriodic* method), 625
 to_array() (*SymbolDict* method), 574
 to_arrays() (*FCIDumpRestricted* method), 678
 to_bohr() (*GeometryMolecular* method), 614
 to_bohr() (*GeometryPeriodic* method), 625
 to_bytes() (*CacheLevels* method), 581
 to_bytes() (*CacheSizeUnit* method), 582
 to_bytes() (*CompilationLevel* method), 1015
 to_bytes() (*CtrluStrat* method), 1017
 to_ChemistryRestrictedIntegralOperator() (*FCIDumpRestricted* method), 678
 to_ChemistryRestrictedIntegralOperator() (*FermionOperator* method), 698
 to_ChemistryRestrictedIntegralOperator() (*PySCFChemistryRestrictedIntegralOperator* method), 1252
 to_ChemistryRestrictedIntegralOperator() (*SymmetryOperatorFermionic* method), 818
 to_ChemistryUnrestrictedIntegralOperator() (*FermionOperator* method), 699
 to_ChemistryUnrestrictedIntegralOperator() (*PySCFChemistryUnrestrictedIntegralOperator* method), 1257
 to_ChemistryUnrestrictedIntegralOperator() (*SymmetryOperatorFermionic* method), 818
 to_circuit() (*QubitOperatorString* method), 794
 to_CircuitAnsatz() (*CircuitAnsatz* method), 339
 to_CircuitAnsatz() (*ComposedAnsatz* method), 344
 to_CircuitAnsatz() (*FermionSpaceAnsatzChemicallyAwareUCCSD* method), 378
 to_CircuitAnsatz() (*FermionSpaceAnsatzkUpCCGD* method), 383
 to_CircuitAnsatz() (*FermionSpaceAnsatzkUpCCGSD* method), 389
 to_CircuitAnsatz() (*FermionSpaceAnsatzkUpCCGS-DSinglet* method), 394
 to_CircuitAnsatz() (*FermionSpaceAnsatzUCCD* method), 367
 to_CircuitAnsatz() (*FermionSpaceAnsatzUCCGD* method), 400
 to_CircuitAnsatz() (*FermionSpaceAnsatzUCCGSD* method), 405
 to_CircuitAnsatz() (*FermionSpaceAnsatzUCCSD* method), 362
 to_CircuitAnsatz() (*FermionSpaceAnsatzUCCSDSinglet* method), 411
 to_CircuitAnsatz() (*FermionSpaceStateExp* method), 356
 to_CircuitAnsatz() (*FermionSpaceStateExpChemicallyAware* method), 372
 to_CircuitAnsatz() (*GeneralAnsatz* method), 334
 to_CircuitAnsatz() (*HamiltonianVariationalAnsatz* method), 451
 to_CircuitAnsatz() (*HardwareEfficientAnsatz* method), 462
 to_CircuitAnsatz() (*LayeredAnsatz* method), 457
 to_CircuitAnsatz() (*MultiConfigurationAnsatz* method), 417
 to_CircuitAnsatz() (*MultiConfigurationState* method), 422
 to_CircuitAnsatz() (*MultiConfigurationStateBox* method), 427
 to_CircuitAnsatz() (*RealGeneralizedBasisRotationAnsatz* method), 433
 to_CircuitAnsatz() (*RealRestrictedBasisRotationAnsatz* method), 438
 to_CircuitAnsatz() (*RealUnrestrictedBasisRotationAnsatz* method), 444
 to_CircuitAnsatz() (*TrotterAnsatz* method), 350
 to_compact_integral_operator() (*ChemistryRestrictedIntegralOperator* method), 657
 to_compact_integral_operator() (*ChemistryUnrestrictedIntegralOperator* method), 667
 to_dict() (*QubitOperatorString* method), 794
 to_dict() (*SymbolDict* method), 574
 to_FermionOperator() (*ChemistryRestrictedIntegralOperator* method), 657
 to_FermionOperator() (*ChemistryRestrictedIntegralOperatorCompact* method), 662
 to_FermionOperator() (*ChemistryUnrestrictedIntegralOperator* method), 667
 to_FermionOperator() (*ChemistryUnrestrictedIntegralOperatorCompact* method), 672
 to_FermionOperator() (*PySCFChemistryRestrictedIntegralOperator* method), 1253
 to_FermionOperator() (*PySCFChemistryUnrestrictedIntegralOperator* method), 1257
 to_index() (*FermionStateString* method), 1069
 to_index() (*QubitStateString* method), 1084
 to_index() (*StateString* method), 1098
 to_latex() (*FermionOperator* method), 699
 to_latex() (*FermionOperatorString* method), 725
 to_latex() (*QubitOperator* method), 752
 to_latex() (*QubitOperatorString* method), 794
 to_latex() (*SymmetryOperatorFermionic* method), 818
 to_latex() (*SymmetryOperatorPauli* method), 865
 to_list() (*QubitOperator* method), 753
 to_list() (*QubitOperatorString* method), 794
 to_list() (*SymmetryOperatorPauli* method), 866
 to_ndarray() (*FermionState* method), 1065
 to_ndarray() (*QubitState* method), 1080
 to_ndarray() (*State* method), 1094
 to_QubitPauliOperator() (*QubitOperator* method), 752

to_QubitPauliOperator() (*SymmetryOperatorPauli method*), 865
 to_QubitPauliString() (*QubitOperatorString method*), 794
 to_QubitState() (*CircuitAnsatz method*), 339
 to_QubitState() (*ComposedAnsatz method*), 345
 to_QubitState() (*FermionSpaceAnsatzChemicallyAwareUCCSD method*), 378
 to_QubitState() (*FermionSpaceAnsatzkUpCCGD method*), 383
 to_QubitState() (*FermionSpaceAnsatzkUpCCGSD method*), 389
 to_QubitState() (*FermionSpaceAnsatzkUpCCGSDSinglet method*), 395
 to_QubitState() (*FermionSpaceAnsatzUCCD method*), 367
 to_QubitState() (*FermionSpaceAnsatzUCCGD method*), 400
 to_QubitState() (*FermionSpaceAnsatzUCCGSD method*), 406
 to_QubitState() (*FermionSpaceAnsatzUCCSD method*), 362
 to_QubitState() (*FermionSpaceAnsatzUCCSDSinglet method*), 411
 to_QubitState() (*FermionSpaceStateExp method*), 356
 to_QubitState() (*FermionSpaceStateExpChemicallyAware method*), 373
 to_QubitState() (*GeneralAnsatz method*), 334
 to_QubitState() (*HamiltonianVariationalAnsatz method*), 451
 to_QubitState() (*HardwareEfficientAnsatz method*), 462
 to_QubitState() (*LayeredAnsatz method*), 457
 to_QubitState() (*MultiConfigurationAnsatz method*), 417
 to_QubitState() (*MultiConfigurationState method*), 423
 to_QubitState() (*MultiConfigurationStateBox method*), 428
 to_QubitState() (*RealGeneralizedBasisRotationAnsatz method*), 433
 to_QubitState() (*RealRestrictedBasisRotationAnsatz method*), 439
 to_QubitState() (*RealUnrestrictedBasisRotationAnsatz method*), 444
 to_QubitState() (*TrotterAnsatz method*), 350
 to_QubitState_direct() (*FermionSpaceAnsatzkUpCCGD method*), 384
 to_QubitState_direct() (*FermionSpaceAnsatzkUpCCGSD method*), 389
 to_QubitState_direct() (*FermionSpaceAnsatzkUpCCGSDSinglet method*), 395
 to_QubitState_direct() (*FermionSpaceAnsatzUCCD method*), 368
 to_QubitState_direct() (*FermionSpaceAnsatzUC-CGD method*), 400
 to_QubitState_direct() (*FermionSpaceAnsatzUC-CGSD method*), 406
 to_QubitState_direct() (*FermionSpaceAnsatzUCCSD method*), 362
 to_QubitState_direct() (*FermionSpaceAnsatzUCCSDSinglet method*), 412
 to_QubitState_direct() (*FermionSpaceStateExp method*), 357
 to_QubitState_direct() (*HamiltonianVariationalAnsatz method*), 451
 to_QubitState_direct() (*TrotterAnsatz method*), 350
 to_sparray() (*FermionState method*), 1066
 to_sparray() (*QubitState method*), 1081
 to_sparray() (*State method*), 1095
 to_sparse_matrices() (*QubitOperatorList method*), 785
 to_sparse_matrices() (*SymmetryOperatorPauliFactorised method*), 898
 to_sparse_matrix() (*QubitOperator method*), 753
 to_sparse_matrix() (*QubitOperatorString method*), 795
 to_sparse_matrix() (*SymmetryOperatorPauli method*), 866
 to_symmetry_operator_fermionic() (*SymmetryOperatorFermionicFactorised method*), 839
 to_symmetry_operator_pauli() (*SymmetryOperatorPauliFactorised method*), 899
 to_uncompacted_integral_operator() (*ChemistryRestrictedIntegralOperatorCompact method*), 663
 to_uncompacted_integral_operator() (*ChemistryUnrestrictedIntegralOperatorCompact method*), 672
 to_zmatrix() (*GeometryMolecular method*), 614
 tobytes() (*ComputableNDArray method*), 522
 toeplitz_decomposition() (*QubitOperator method*), 753
 toeplitz_decomposition() (*SymmetryOperatorPauli method*), 867
 tofile() (*ComputableNDArray method*), 522
 TOLERANCE (*ChemistryRestrictedIntegralOperator attribute*), 652
 TOLERANCE (*ChemistryRestrictedIntegralOperatorCompact attribute*), 658
 TOLERANCE (*ChemistryUnrestrictedIntegralOperator attribute*), 663
 TOLERANCE (*ChemistryUnrestrictedIntegralOperatorCompact attribute*), 668
 TOLERANCE (*HadamardTestDerivativeOverlap attribute*), 967
 TOLERANCE (*PySCFChemistryRestrictedIntegralOperator attribute*), 1249

TOLERANCE (*PySCFChemistryUnrestrictedIntegralOperator attribute*), 1253
tolist() (*ComputableNDArray method*), 523
tostring() (*ComputableNDArray method*), 524
totally_commuting_decomposition() (*QubitOperator method*), 754
totally_commuting_decomposition() (*SymmetryOperatorPauli method*), 867
trace() (*ComputableNDArray method*), 524
trace() (*RestrictedOneBodyRDM method*), 797
trace() (*UnrestrictedOneBodyRDM method*), 903
transf() (*AVAS method*), 1106
transf() (*CASSCF method*), 1107
transform() (*OrbitalOptimizer method*), 728
transform() (*OrbitalTransformer method*), 729
translate() (*FermionOperatorList.CompressScalarsBehavior method*), 707
translate() (*FermionOperatorList.FactoryCoefficientsLocation method*), 712
translate() (*FermionOperator.TrotterizeCoefficientsLocation method*), 684
translate() (*QubitOperatorList.CompressScalarsBehavior method*), 761
translate() (*QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method*), 766
translate() (*QubitOperatorList.FactoryCoefficientsLocation method*), 771
translate() (*QubitOperator.TrotterizeCoefficientsLocation method*), 735
translate() (*SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method*), 826
translate() (*SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method*), 830
translate() (*SymmetryOperatorFermionic.TrotterizeCoefficientsLocation method*), 803
translate() (*SymmetryOperatorPauliFactorised.CompressScalarsBehavior method*), 874
translate() (*SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method*), 879
translate() (*SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method*), 884
translate() (*SymmetryOperatorPauli.TrotterizeCoefficientsLocation method*), 847
translate_by_vector() (*GeometryMolecular method*), 614
translate_by_vector() (*GeometryPeriodic method*), 625
translate_operator() (*FermionSpaceSupercell method*), 1050
transpose() (*ComputableNDArray method*), 524
TrotterAnsatz (*class in inquanto.ansatzes*), 345
trotterize() (*FermionOperator method*), 700
trotterize() (*QubitOperator method*), 754
trotterize() (*SymmetryOperatorFermionic method*), 819
trotterize() (*SymmetryOperatorPauli method*), 867
trotterize_as_linear_combination() (*FermionOperatorList method*), 720
trotterize_as_linear_combination() (*QubitOperatorList method*), 785
trotterize_as_linear_combination() (*SymmetryOperatorFermionicFactorised method*), 840
trotterize_as_linear_combination() (*SymmetryOperatorPauliFactorised method*), 899
truncated() (*FermionOperator method*), 701
truncated() (*SymmetryOperatorFermionic method*), 820
two_body_iijj() (*ChemistryRestrictedIntegralOperator method*), 657
two_body_iijj() (*ChemistryRestrictedIntegralOperatorCompact method*), 663
two_body_to_tensor() (*FCIDumpRestricted method*), 678

U

UNRESTRICTED (*IntegralType attribute*), 726
unrestricted_basis_rotation_to_circuit() (*in module inquanto.ansatzes*), 445
UnrestrictedOneBodyRDM (*class in inquanto.operators*), 902
UnrestrictedOneBodyRDMComputable (*class in inquanto.computables.composite*), 569
UnrestrictedOneBodyRDMRealComputable (*class in inquanto.computables.composite*), 570
UnrestrictedTwoBodyRDM (*class in inquanto.operators*), 903
unsympify() (*FermionOperator method*), 701
unsympify() (*FermionOperatorList method*), 721
unsympify() (*FermionState method*), 1066
unsympify() (*QubitOperator method*), 755
unsympify() (*QubitOperatorList method*), 786
unsympify() (*QubitState method*), 1081
unsympify() (*State method*), 1096
unsympify() (*SymmetryOperatorFermionic method*), 820

unsympify() (*SymmetryOperatorFermionicFactorised method*), 841
 unsympify() (*SymmetryOperatorPauli method*), 869
 unsympify() (*SymmetryOperatorPauliFactorised method*), 900
 untrotterize() (*FermionOperatorList method*), 721
 untrotterize() (*QubitOperatorList method*), 787
 untrotterize() (*SymmetryOperatorFermionicFactorised method*), 841
 untrotterize() (*SymmetryOperatorPauliFactorised method*), 900
 untrotterize_partitioned() (*FermionOperatorList method*), 722
 untrotterize_partitioned() (*QubitOperatorList method*), 787
 untrotterize_partitioned() (*SymmetryOperatorFermionicFactorised method*), 841
 untrotterize_partitioned() (*SymmetryOperatorPauliFactorised method*), 901
 update() (*SymbolDict method*), 574
 update() (*SymbolSet method*), 578
 update_k_and_beta() (*IterativePhaseEstimation method*), 991
 update_k_and_beta() (*IterativePhaseEstimationQuantinuum method*), 996
 update_k_and_beta() (*IterativePhaseEstimationStatevector method*), 999
 update_set() (*QubitMapping class method*), 629
 update_set() (*QubitMappingBravyiKitaev method*), 636
 update_set() (*QubitMappingJordanWigner method*), 632
 update_set() (*QubitMappingParaparticular method*), 643
 update_set() (*QubitMappingParity class method*), 640
 upper() (*FermionOperatorList.CompressScalarsBehavior method*), 707
 upper() (*FermionOperatorList.FactoryCoefficientsLocation method*), 712
 upper() (*FermionOperator.TrotterizeCoefficientsLocation method*), 684
 upper() (*QubitOperatorList.CompressScalarsBehavior method*), 761
 upper() (*QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method*), 766
 upper() (*QubitOperatorList.FactoryCoefficientsLocation method*), 771
 upper() (*QubitOperator.TrotterizeCoefficientsLocation method*), 735
 upper() (*SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method*), 826
 upper() (*SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method*), 831
 upper() (*SymmetryOperatorFermionic.TrotterizeCoefficientsLocation method*), 803
 upper() (*SymmetryOperatorPauliFactorised.CompressScalarsBehavior method*), 874
 upper() (*SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method*), 879
 upper() (*SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method*), 884
 upper() (*SymmetryOperatorPauli.TrotterizeCoefficientsLocation method*), 847

V

value (*ComputableInt attribute*), 488
 values() (*SymbolDict method*), 574
 var() (*ComputableNDArray method*), 525
 vdot() (*FermionState method*), 1066
 vdot() (*QubitState method*), 1082
 vdot() (*State method*), 1096
 vector_to_dict() (*in module inquanto.core*), 585
 view() (*ComputableNDArray method*), 525
 visualize_fragmentation() (*VisualizerNGL method*), 1264
 visualize_molecule() (*VisualizerNGL method*), 1265
 visualize_orbitals() (*VisualizerNGL method*), 1265
 visualize_unit_cell() (*VisualizerNGL method*), 1265
 VisualizerNGL (*class in inquanto.extensions.nglview*), 1264

W

walk() (*CommutatorComputable method*), 533
 walk() (*ComputableFunction method*), 487
 walk() (*ComputableInt method*), 488
 walk() (*ComputableList method*), 491
 walk() (*ComputableNDArray method*), 527
 walk() (*ComputableNode method*), 529
 walk() (*ComputableSingleChild method*), 530
 walk() (*ExpectationValue method*), 464
 walk() (*ExpectationValueBraDerivativeImag method*), 466
 walk() (*ExpectationValueBraDerivativeReal method*), 467
 walk() (*ExpectationValueDerivative method*), 469
 walk() (*ExpectationValueKetDerivativeImag method*), 471
 walk() (*ExpectationValueKetDerivativeReal method*), 472
 walk() (*ExpectationValueNonHermitian method*), 474
 walk() (*ExpectationValueSumComputable method*), 535
 walk() (*HoleGFComputable method*), 536

walk() (*KrylovSubspaceComputable method*), 540
 walk() (*LanczosCoefficientsComputable method*), 542
 walk() (*LanczosMatrixComputable method*), 544
 walk() (*ManyBodyGFCcomputable method*), 546
 walk() (*MetricTensorImag method*), 476
 walk() (*MetricTensorReal method*), 477
 walk() (*NonOrthogonalMatricesComputable method*), 548
 walk() (*Overlap method*), 479
 walk() (*OverlapImag method*), 481
 walk() (*OverlapMatrixComputable method*), 549
 walk() (*OverlapReal method*), 483
 walk() (*OverlapSquared method*), 485
 walk() (*ParticleGFComputable method*), 553
 walk() (*PDM1234RealComputable method*), 551
 walk() (*QCM4Computable method*), 555
 walk() (*QSEMatricesComputable method*), 557
 walk() (*RDM1234RealComputable method*), 559
 walk() (*RestrictedOneBodyRDMComputable method*), 560
 walk() (*RestrictedOneBodyRDMRealComputable method*), 562
 walk() (*SCEOMMatrixComputable method*), 564
 walk() (*SpinlessNBodyPDMArrayRealComputable method*), 566
 walk() (*SpinlessNBodyRDMArrayRealComputable method*), 568
 walk() (*UnrestrictedOneBodyRDMComputable method*), 570
 walk() (*UnrestrictedOneBodyRDMRealComputable method*), 571
 write() (*FCIDumpRestricted method*), 678
 WRITEABLE (*ComputableNDArray attribute*), 503
 WRITEBACKIFCOPY (*ComputableNDArray attribute*), 503

zfill() (*QubitOperatorList.CompressScalarsBehavior method*), 761
 zfill() (*QubitOperatorList.ExpandExponentialProductCoefficientsBehavior method*), 766
 zfill() (*QubitOperatorList.FactoryCoefficientsLocation method*), 771
 zfill() (*QubitOperator.TrotterizeCoefficientsLocation method*), 735
 zfill() (*SymmetryOperatorFermionicFactorised.CompressScalarsBehavior method*), 826
 zfill() (*SymmetryOperatorFermionicFactorised.FactoryCoefficientsLocation method*), 831
 zfill() (*SymmetryOperatorFermionicFactorised.TrotterizeCoefficientsLocation method*), 803
 zfill() (*SymmetryOperatorPauliFactorised.CompressScalarsBehavior method*), 874
 zfill() (*SymmetryOperatorPauliFactorised.ExpandExponentialProductCoefficientsBehavior method*), 879
 zfill() (*SymmetryOperatorPauliFactorised.FactoryCoefficientsLocation method*), 884
 zfill() (*SymmetryOperatorPauli.TrotterizeCoefficientsLocation method*), 847
 zmatrix (*GeometryMolecular property*), 614
 zmatrix_to_df() (*GeometryMolecular method*), 615

X

xyz (*GeometryMolecular property*), 614
 xyz (*GeometryPeriodic property*), 625
 xyz_to_df() (*GeometryMolecular method*), 614
 xyz_to_df() (*GeometryPeriodic method*), 625

Z

zero() (*FermionOperator class method*), 701
 zero() (*FermionState class method*), 1067
 zero() (*QubitOperator class method*), 756
 zero() (*QubitState class method*), 1082
 zero() (*State class method*), 1096
 zero() (*SymmetryOperatorFermionic class method*), 821
 zero() (*SymmetryOperatorPauli class method*), 869
 zfill() (*FermionOperatorList.CompressScalarsBehavior method*), 707
 zfill() (*FermionOperatorList.FactoryCoefficientsLocation method*), 712
 zfill() (*FermionOperator.TrotterizeCoefficientsLocation method*), 684