OpenFermion-Cirq Documentation

Release 0.1a0

The OpenFermion Developers

Apr 02, 2020

Contents

CHAPTER 1

API Reference

1.1 Gates

1.1.1 Two-Qubit Gates

FermionicSwapGate
XXYYGate
YXXYGate
ZZGate
FSWAP
XXYY
YXXY
ZZ

openfermioncirq.FSWAP

openfermioncirq.XXYY

openfermioncirq.YXXY

1.1.2 Three-Qubit Gates

ControlledXXYYGate
ControlledYXXYGate
Rot111Gate
CXXYY
CYXXY
CCZ

openfermioncirq.CXXYY

openfermioncirq.CYXXY

1.1.3 Fermionic Simulation Gates

QuadraticFermionicSimulationGate(weights, ...)	(w0 |10><01| + h.c.) + w1 * |11><11| interaction.
CubicFermionicSimulationGate(weights, ...)	w0 * |110><101| + w1 * |110><011| + w2 * |101><011| + hc interaction.
QuarticFermionicSimulationGate(weights, ...)	Rotates Hamming-weight 2 states into their bitwise complements.

openfermioncirq.QuadraticFermionicSimulationGate

class openfermioncirq.**QuadraticFermionicSimulationGate**(*weights: Tuple[float, float] = (1, 1)*, ***kwargs*)

(w0 |10><01| + h.c.) + w1 * |11><11| interaction.

With weights (w_0, w_1) and exponent t, this gate's matrix is defined as

$$e^{-itH},$$

where

$$H = (w_0 |10\rangle \langle 01| + \text{h.c.}) - w_1 |11\rangle \langle 11|.$$

This corresponds to the Jordan-Wigner transform of

$$H = (w_0 a_i^\dagger a_{i+1} + \text{h.c.}) + w_1 a_i^\dagger a_{i+1}^\dagger a_i a_{i+1},$$

where a_i and a_{i+1} are the annihilation operators for the fermionic modes i and $(i + 1)$, respectively mapped to the first and second qubits on which this gate acts.

Parameters **weights** – The weights of the terms in the Hamiltonian.

__init__(*weights: Tuple[float, float] = (1, 1)*, ***kwargs*) → None

Initializes the parameters used to compute the gate's matrix.

The eigenvalue of each eigenspace of a gate is computed by

1. Starting with an angle in half turns as returned by the gate's `_eigen_components` method:

   θ

2. Shifting the angle by *global_shift*:

   θ + s

3. Scaling the angle by *exponent*:

   (θ + s) * e

4. Converting from half turns to a complex number on the unit circle:

   exp(i * pi * (θ + s) * e)

Parameters

- **exponent** – The t in gate**t. Determines how much the eigenvalues of the gate are scaled by. For example, eigenvectors phased by -1 when *gate**1* is applied will gain a relative phase of e^{i pi exponent} when *gate**exponent* is applied (relative to eigenvectors unaffected by *gate**1*).
- **global_shift** – Offsets the eigenvalues of the gate at exponent=1. In effect, this controls a global phase factor on the gate's unitary matrix. The factor is:

  exp(i * pi * global_shift * exponent)

  For example, *cirq.X**t* uses a *global_shift* of 0 but *cirq.rx(t)* uses a *global_shift* of -0.5, which is why *cirq.unitary(cirq.rx(pi))* equals -iX instead of X.

Methods

`__init__`(weights, float] = (1, 1), **kwargs)	Initializes the parameters used to compute the gate's matrix.
`controlled`(num_controls, control_values, ...)	Returns a controlled version of this gate.
`num_qubits`()	The number of qubits this gate acts on.
`on`(*qubits)	Returns an application of this gate to the given qubits.
`qubit_index_to_equivalence_group_key`(index)	Returns a key that differs between non-interchangeable qubits.
`validate_args`(qubits)	Checks if this gate can be applied to the given qubits.
`wrap_in_linear_combination`(coefficient, ...)	

Attributes

`exponent`
`global_shift`

openfermioncirq.CubicFermionicSimulationGate

class openfermioncirq.**CubicFermionicSimulationGate**(*weights: Tuple[complex, complex, complex] = (1.0, 1.0, 1.0), **kwargs*)

w0 * **|110><101|** + w1 * **|110><011|** + w2 * **|101><011|** + hc interaction.

With weights (w_0, w_1, w_2) and exponent t, this gate's matrix is defined as

$$e^{-itH},$$

where

$$H = (w_0 \left|110\right\rangle \left\langle 101\right| + \text{h.c.}) + (w_1 \left|110\right\rangle \left\langle 011\right| + \text{h.c.}) + (w_2 \left|101\right\rangle \left\langle 011\right| + \text{h.c.})$$

This corresponds to the Jordan-Wigner transform of

$$H = -\left(w_0 a_i^\dagger a_{i+1}^\dagger a_i a_{i+2} + \text{h.c.}\right) - \left(w_1 a_i^\dagger a_{i+1}^\dagger a_{i+1} a_{i+2} + \text{h.c.}\right) - \left(w_2 a_i^\dagger a_{i+2}^\dagger a_{i+1} a_{i+2} + \text{h.c.}\right),$$

where a_i, a_{i+1}, a_{i+2} are the annihilation operators for the fermionic modes i, $(i + 1)$ $(i + 2)$, respectively mapped to the three qubits on which this gate acts.

Parameters **weights** – The weights of the terms in the Hamiltonian.

__init__(*weights: Tuple[complex, complex, complex] = (1.0, 1.0, 1.0), **kwargs*) → None
Initializes the parameters used to compute the gate's matrix.

The eigenvalue of each eigenspace of a gate is computed by

1. Starting with an angle in half turns as returned by the gate's `_eigen_components` method:

   θ

2. Shifting the angle by *global_shift*:

   θ + s

3. Scaling the angle by *exponent*:

   (θ + s) * e

4. Converting from half turns to a complex number on the unit circle:

   exp(i * pi * (θ + s) * e)

Parameters

- **exponent** – The t in gate**t. Determines how much the eigenvalues of the gate are scaled by. For example, eigenvectors phased by -1 when *gate**1* is applied will gain a relative phase of e^{i pi exponent} when *gate**exponent* is applied (relative to eigenvectors unaffected by *gate**1*).
- **global_shift** – Offsets the eigenvalues of the gate at exponent=1. In effect, this controls a global phase factor on the gate's unitary matrix. The factor is:

  exp(i * pi * global_shift * exponent)

  For example, *cirq.X**t* uses a *global_shift* of 0 but *cirq.rx(t)* uses a *global_shift* of -0.5, which is why *cirq.unitary(cirq.rx(pi))* equals -iX instead of X.

Methods

`__init__`(weights, complex, complex] = (1.0, ...)	Initializes the parameters used to compute the gate's matrix.
`controlled`(num_controls, control_values, ...)	Returns a controlled version of this gate.
`num_qubits`()	The number of qubits this gate acts on.
`on`(*qubits)	Returns an application of this gate to the given qubits.
`validate_args`(qubits)	Checks if this gate can be applied to the given qubits.
`wrap_in_linear_combination`(coefficient, ...)	

Attributes

`exponent`
`global_shift`

openfermioncirq.QuarticFermionicSimulationGate

class openfermioncirq.**QuarticFermionicSimulationGate**(*weights: Tuple[complex, complex, complex] = (1, 1, 1)*, *absorb_exponent: bool = True*, ***, *exponent: Union[sympy.core.symbol.Symbol, float, None] = None*, *rads: Optional[float] = None*, *degs: Optional[float] = None*, *duration: Optional[float] = None*)

Rotates Hamming-weight 2 states into their bitwise complements.

With weights (w_0, w_1, w_2) and exponent t, this gate's matrix is defined as

$$e^{-itH},$$

where

$$H = (w_0 \left|1001\right\rangle \left\langle 0110\right| + \text{h.c.}) + (w_1 \left|1010\right\rangle \left\langle 0101\right| + \text{h.c.}) + (w_2 \left|1100\right\rangle \left\langle 0011\right| + \text{h.c.})$$

This corresponds to the Jordan-Wigner transform of

$$H = -\left(w_0 a_i^\dagger a_{i+3}^\dagger a_{i+1} a_{i+2} + \text{h.c.}\right) - \left(w_1 a_i^\dagger a_{i+2}^\dagger a_{i+1} a_{i+3} + \text{h.c.}\right) - \left(w_2 a_i^\dagger a_{i+1}^\dagger a_{i+2} a_{i+3} + \text{h.c.}\right),$$

where $a_i, \ldots, a_{i+3}$ are the annihilation operators for the fermionic modes $i, \ldots, (i+3)$, respectively mapped to the four qubits on which this gate acts.

Parameters **weights** – The weights of the terms in the Hamiltonian.

__init__(*weights: Tuple[complex, complex, complex] = (1, 1, 1)*, *absorb_exponent: bool = True*, ***, *exponent: Union[sympy.core.symbol.Symbol, float, None] = None*, *rads: Optional[float] = None*, *degs: Optional[float] = None*, *duration: Optional[float] = None*) → None

Initialize the gate.

At most one of exponent, rads, degs, or duration may be specified. If more are specified, the result is considered ambiguous and an error is thrown. If no argument is given, the default value of one half-turn is used.

Parameters

- **weights** – The weights of the terms in the Hamiltonian.
- **absorb_exponent** – Whether to absorb the given exponent into the weights. If true, the exponent of the returned gate is 1.
- **exponent** – The exponent angle, in half-turns.
- **rads** – The exponent angle, in radians.
- **degs** – The exponent angle, in degrees.
- **duration** – The exponent as a duration of time.

Methods

__init__(weights, complex, complex] = (1, 1, ...)	Initialize the gate.
absorb_exponent_into_weights()	
controlled(num_controls, control_values, ...)	Returns a controlled version of this gate.
num_qubits()	The number of qubits this gate acts on.
on(*qubits)	Returns an application of this gate to the given qubits.
validate_args(qubits)	Checks if this gate can be applied to the given qubits.
wrap_in_linear_combination(coefficient, ...)	

Attributes

exponent
global_shift

1.2 Primitives

bogoliubov_transform(transformation_matrix, ...)	Perform a Bogoliubov transformation.
ffft()	Performs fast fermionic Fourier transform.
prepare_gaussian_state(...)	Prepare a fermionic Gaussian state from a computational basis state.
prepare_slater_determinant(...)	Prepare a Slater determinant from a computational basis state.
swap_network(operation, int, ...)	Apply operations to pairs of qubits or modes using a swap network.

1.2.1 openfermioncirq.bogoliubov_transform

1.2.2 openfermioncirq.ffft

1.2.3 openfermioncirq.prepare_gaussian_state

1.2.4 openfermioncirq.prepare_slater_determinant

1.2.5 openfermioncirq.swap_network

1.3 Hamiltonian Simulation

simulate_trotter(hamiltonian, ...)	Simulate Hamiltonian evolution using a Trotter-Suzuki product formula.
trotter.TrotterStep(hamiltonian, ...)	A method for performing a Trotter step.
trotter.TrotterAlgorithm	An algorithm for performing a Trotter step.
trotter.LINEAR_SWAP_NETWORK	A Trotter algorithm using the "fermionic simulation gate".

Continued on next page

Table 11 – continued from previous page

trotter.SPLIT_OPERATOR	A Trotter algorithm using a split-operator approach.
trotter.LOW_RANK	A Trotter algorithm using the low rank decomposition strategy.

1.3.1 openfermioncirq.simulate_trotter

1.3.2 openfermioncirq.trotter.TrotterStep

class openfermioncirq.trotter.**TrotterStep**(*hamiltonian: Union[openfermion.ops._fermion_operator.FermionOpera openfermion.ops._qubit_operator.QubitOperator, openfermion.ops._interaction_operator.InteractionOperator, openfermion.ops._diagonal_coulomb_hamiltonian.DiagonalCoulombHa*

A method for performing a Trotter step.

This class assumes that Hamiltonian evolution using a Trotter-Suzuki product formula is performed in the following steps:

1. Perform some preparatory operations (for instance, a basis change).
2. Perform a number of Trotter steps. Each Trotter step may induce a permutation on the ordering in which qubits represent fermionic modes.
3. Perform some finishing operations.

hamiltonian
The Hamiltonian being simulated.

__init__(*hamiltonian: Union[openfermion.ops._fermion_operator.FermionOperator, openfermion.ops._qubit_operator.QubitOperator, openfermion.ops._interaction_operator.InteractionOperator, openfermion.ops._diagonal_coulomb_hamiltonian.DiagonalCoulombHamiltonian]*) → None
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(hamiltonian, ...)	Initialize self.
finish(qubits, n_steps, control_qubit, ...)	Operations to perform after all Trotter steps are done.
prepare(qubits, control_qubit)	Operations to perform before doing the Trotter steps.
step_qubit_permutation(qubits, control_qubit)	The qubit permutation induced by a single Trotter step.
trotter_step(qubits, time, control_qubit)	Yield operations to perform a Trotter step.

1.3.3 openfermioncirq.trotter.TrotterAlgorithm

class openfermioncirq.trotter.**TrotterAlgorithm**
An algorithm for performing a Trotter step.

A Trotter step algorithm contains methods for performing a symmetric or asymmetric Trotter step and their controlled versions. It does not need to support all the possibilities; for instance, it may support only symmetric Trotter steps with no control qubit. Support for a kind of Trotter step is implemented by overriding the methods of this class.

supported_types
A set containing types of Hamiltonian representations that can be simulated using this Trotter step algorithm. For example, {DiagonalCoulombHamiltonian, InteractionOperator}.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

asymmetric(hamiltonian, ...)
controlled_asymmetric(hamiltonian, ...)
controlled_symmetric(hamiltonian, ...)
symmetric(hamiltonian, ...)

Attributes

supported_types

1.3.4 openfermioncirq.trotter.LINEAR_SWAP_NETWORK

openfermioncirq.trotter.**LINEAR_SWAP_NETWORK = <openfermioncirq.trotter.algorithms.linear_sw**
A Trotter algorithm using the "fermionic simulation gate".

This algorithm simulates a DiagonalCoulombHamiltonian. It uses layers of fermionic swap networks to simultaneously simulate the one- and two-body interactions.

This algorithm is described in arXiv:1711.04789.

1.3.5 openfermioncirq.trotter.SPLIT_OPERATOR

openfermioncirq.trotter.**SPLIT_OPERATOR = <openfermioncirq.trotter.algorithms.split_operato**
A Trotter algorithm using a split-operator approach.

This algorithm simulates a DiagonalCoulombHamiltonian. It uses Bogoliubov transformations to switch between a basis in which the one-body terms are convenient to simulate and a basis in which the two-body terms are convenient to simulate. The Bogoliubov transformations are implemented using Givens rotations.

This algorithm is described in arXiv:1706.00023.

1.3.6 openfermioncirq.trotter.LOW_RANK

openfermioncirq.trotter.**LOW_RANK = <openfermioncirq.trotter.algorithms.low_rank.LowRankTrot**
A Trotter algorithm using the low rank decomposition strategy.

This algorithm simulates an InteractionOperator with real coefficients. The one-body terms are simulated in their diagonal basis; the basis change is accomplished with a Bogoliubov transformation. To simulate the two-body terms, the two-body tensor is decomposed into singular components and possibly truncating. Then, each singular component is simulated in the appropriate basis using a (non-fermionic) swap network. The general idea is based on expressing the two-body operator as $\sum_{pqrs} h_{pqrs} a_p^\dagger a_q^\dagger a_r a_s = \sum_{j=0}^{J-1} \lambda_j (\sum_{pq} g_{jpq} a_p^\dagger a_q)^2$ One can then diagonalize the squared one-body component as math:*sum_{pq} g_{pqj} a^dagger_p a_q = R_j (sum_{p} f_{pj} n_p) R_j^dagger* Then, a 'low rank' Trotter step of the two-body tensor can be simulated as $\prod_{j=0}^{J-1} R_j e^{-i\lambda_j \sum_{pq} f_{pj} f_{qj} n_p n_q} R_j^\dagger$. The R_j are Bogoliubov transformations, and one can use a swap network to simulate the diagonal $n_p n_q$ terms. The value of J is either fully the square of the number of qubits, which would imply no truncation, or it is specified by the user, or it is chosen so that $\sum_{l=0}^{L-1} (\sum_{pq} |g_{lpq}|)^2 |\lambda_l| < x$ where x is a truncation threshold specified by user.

1.3.7 Trotter Algorithms

trotter.LinearSwapNetworkTrotterAlgorithm	A Trotter algorithm using the "fermionic simulation gate".
trotter.SplitOperatorTrotterAlgorithm	A Trotter algorithm using a split-operator approach.
trotter.LowRankTrotterAlgorithm(... [, ...])	A Trotter algorithm using the low rank decomposition strategy.

openfermioncirq.trotter.LinearSwapNetworkTrotterAlgorithm

class openfermioncirq.trotter.**LinearSwapNetworkTrotterAlgorithm**

A Trotter algorithm using the "fermionic simulation gate".

This algorithm simulates a DiagonalCoulombHamiltonian. It uses layers of fermionic swap networks to simultaneously simulate the one- and two-body interactions.

This algorithm is described in arXiv:1711.04789.

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

asymmetric(hamiltonian, ...)	
controlled_asymmetric(hamiltonian, ...)	
controlled_symmetric(hamiltonian, ...)	
symmetric(hamiltonian, ...)	

Attributes

supported_types	

openfermioncirq.trotter.SplitOperatorTrotterAlgorithm

class openfermioncirq.trotter.**SplitOperatorTrotterAlgorithm**

A Trotter algorithm using a split-operator approach.

This algorithm simulates a DiagonalCoulombHamiltonian. It uses Bogoliubov transformations to switch between a basis in which the one-body terms are convenient to simulate and a basis in which the two-body terms are convenient to simulate. The Bogoliubov transformations are implemented using Givens rotations.

This algorithm is described in arXiv:1706.00023.

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

asymmetric(hamiltonian, ...)	
controlled_asymmetric(hamiltonian, ...)	

Continued on next page

Table 18 – continued from previous page

controlled_symmetric(hamiltonian, ...)
symmetric(hamiltonian, ...)

Attributes

supported_types

openfermioncirq.trotter.LowRankTrotterAlgorithm

class openfermioncirq.trotter.**LowRankTrotterAlgorithm**(*truncation_threshold: Optional[float] = 1e-08*, *final_rank: Optional[int] = None*, *spin_basis=True*)

A Trotter algorithm using the low rank decomposition strategy.

This algorithm simulates an InteractionOperator with real coefficients. The one-body terms are simulated in their diagonal basis; the basis change is accomplished with a Bogoliubov transformation. To simulate the two-body terms, the two-body tensor is decomposed into singular components and possibly truncating. Then, each singular component is simulated in the appropriate basis using a (non-fermionic) swap network. The general idea is based on expressing the two-body operator as $\sum_{pqrs} h_{pqrs} a_p^\dagger a_q^\dagger a_r a_s = \sum_{j=0}^{J-1} \lambda_j (\sum_{pq} g_{jpq} a_p^\dagger a_q)^2$ One can then diagonalize the squared one-body component as math:*sum_{pq} g_{pqj} a^dagger_p a_q = R_j (sum_{p} f_{pj} n_p) R_j^dagger* Then, a 'low rank' Trotter step of the two-body tensor can be simulated as $\prod_{j=0}^{J-1} R_j e^{-i\lambda_j \sum_{pq} f_{pj} f_{qj} n_p n_q} R_j^\dagger$. The R_j are Bogoliubov transformations, and one can use a swap network to simulate the diagonal $n_p n_q$ terms. The value of J is either fully the square of the number of qubits, which would imply no truncation, or it is specified by the user, or it is chosen so that $\sum_{l=0}^{L-1} (\sum_{pq} |g_{lpq}|)^2 |\lambda_l| < x$ where x is a truncation threshold specified by user.

__init__(*truncation_threshold: Optional[float] = 1e-08*, *final_rank: Optional[int] = None*, *spin_basis=True*) → None

Parameters

- **truncation_threshold** – The value of x from the docstring of this class.
- **final_rank** – If provided, this specifies the value of J at which to truncate.
- **spin_basis** – Whether the Hamiltonian is given in the spin orbital (rather than spatial orbital) basis.

Methods

__init__(truncation_threshold, final_rank[, ...])
param truncation_threshold The value of x from the docstring of
asymmetric(hamiltonian, ...)
controlled_asymmetric(hamiltonian, ...)
controlled_symmetric(hamiltonian, ...)
symmetric(hamiltonian, ...)

Attributes

supported_types	

1.4 Variational Algorithms

VariationalAnsatz(qubits)	A variational ansatz.
VariationalStudy(name, ansatz, objective, ...)	The results from optimizing a variational ansatz.
HamiltonianVariationalStudy	

1.4.1 openfermioncirq.VariationalAnsatz

class openfermioncirq.**VariationalAnsatz**(*qubits: Optional[Sequence[cirq.ops.raw_types.Qid]] = None*)

A variational ansatz.

A variational ansatz is a parameterized circuit. The VariationalAnsatz class stores parameters as instances of the Symbol class. A Symbol is simply a named object that can be used in a circuit and whose numerical value is determined at run time. The Symbols are stored in a dictionary whose keys are the names of the corresponding parameters. For instance, the Symbol corresponding to the parameter 'theta_0' would be obtained with the expression *self.params['theta_0']*.

params
A dictionary storing the parameters by name. Key is the string name of a parameter and the corresponding value is a Symbol with the same name.

circuit
The ansatz circuit.

qubits
A list containing the qubits used by the ansatz circuit.

__init__(*qubits: Optional[Sequence[cirq.ops.raw_types.Qid]] = None*) → None

Parameters **qubits** – Qubits to be used by the ansatz circuit. If not specified, then qubits will automatically be generated by the *_generate_qubits* method.

Methods

__init__(qubits)	**param qubits** Qubits to be used by the ansatz circuit. If not specified,
default_initial_params()	Suggested initial parameter settings.
operations(qubits)	Produce the operations of the ansatz circuit.
param_bounds()	Optional bounds on the parameters.
param_resolver(param_values)	Interprets parameters input as an array of real numbers.
param_scale_factors()	Coefficients to scale parameters by during optimization.
params()	The parameters of the ansatz.
qubit_permutation(qubits)	The qubit permutation induced by the ansatz circuit.

1.4.2 openfermioncirq.VariationalStudy

class openfermioncirq.**VariationalStudy**(*name: str*, *ansatz: openfermioncirq.variational.ansatz.VariationalAnsatz*, *objective: openfermioncirq.variational.objective.VariationalObjective*, *preparation_circuit: Optional[cirq.circuits.circuit.Circuit] = None*, *initial_state: Union[int, numpy.ndarray] = 0*, *target: Optional[float] = None*, *black_box_type: Type[openfermioncirq.variational.variational_black_box.VariationalBlackBo = <class 'openfermioncirq.variational.variational_black_box.UnitarySimulateVariationalBlackBox datadir: Optional[str] = None*)

The results from optimizing a variational ansatz.

A VariationalStudy is used to facilitate optimizing the parameters of a variational ansatz. It contains methods for performing optimizations and saving and loading the results.

Example:: ansatz = SomeVariationalAnsatz() objective = SomeVariationalObjective() study = SomeVariationalStudy('my_study', ansatz, objective) optimization_params = OptimizationParams(

algorithm=openfermioncirq.optimization.COBYLA, initial_guess=numpy.zeros(5))

result = study.optimize(optimization_params, identifier='run0') print(result.optimal_value) # prints a number print(result.params.initial_guess) # prints the initial guess used study.save() # saves the study with all results obtained so far

name
The name of the study.

circuit
The circuit of the study, which is the preparation circuit, if any, followed by the ansatz circuit.

ansatz
The ansatz being studied.

objective
The objective function of interest.

target
An optional target value one wants to achieve during optimization.

trial_results
A dictionary of OptimizationTrialResults from optimization runs of the study. Key is the identifier used to label the run.

num_params
The number of parameters in the circuit.

__init__(*name: str*, *ansatz: openfermioncirq.variational.ansatz.VariationalAnsatz*, *objective: openfermioncirq.variational.objective.VariationalObjective*, *preparation_circuit: Optional[cirq.circuits.circuit.Circuit] = None*, *initial_state: Union[int, numpy.ndarray] = 0*, *target: Optional[float] = None*, *black_box_type: Type[openfermioncirq.variational.variational_black_box.VariationalBlackBox] = <class 'openfermioncirq.variational.variational_black_box.UnitarySimulateVariationalBlackBox'>*, *datadir: Optional[str] = None*) → None

Parameters

- **name** – The name of the study.

- **ansatz** – The ansatz to study.
- **objective** – The objective function.
- **preparation_circuit** – A circuit to apply prior to the ansatz circuit. It should use the qubits belonging to the ansatz.
- **initial_state** – An initial state to use if the study circuit is run on a simulator.
- **target** – The target value one wants to achieve during optimization.
- **black_box_type** – The type of VariationalBlackBox to use for optimization.
- **datadir** – The directory to use when saving the study. The default behavior is to use the current working directory.

Methods

__init__(name, ansatz, objective, ...)	**param name** The name of the study.
extend_result(identifier, ...)	Extend a result by repeating the run with the same parameters.
load(name, datadir)	Load a study from disk.
optimize(optimization_params, identifier, ...)	Perform an optimization run and save the results.
optimize_sweep(param_sweep, identifiers, ...)	Perform multiple optimization runs and save the results.
save()	Save the study to disk.
value_of(params)	Determine the value of some parameters.

Attributes

ansatz	The ansatz associated with the study.
circuit	The preparation circuit followed by the ansatz circuit.
num_params	The number of parameters of the ansatz.
objective	The objective associated with the study.

1.4.3 Variational Ansatzes

SwapNetworkTrotterAnsatz(hamiltonian, ...)	An ansatz based on the fermionic swap network.
SplitOperatorTrotterAnsatz(hamiltonian, ...)	An ansatz based on a split-operator Trotter step.

openfermioncirq.SwapNetworkTrotterAnsatz

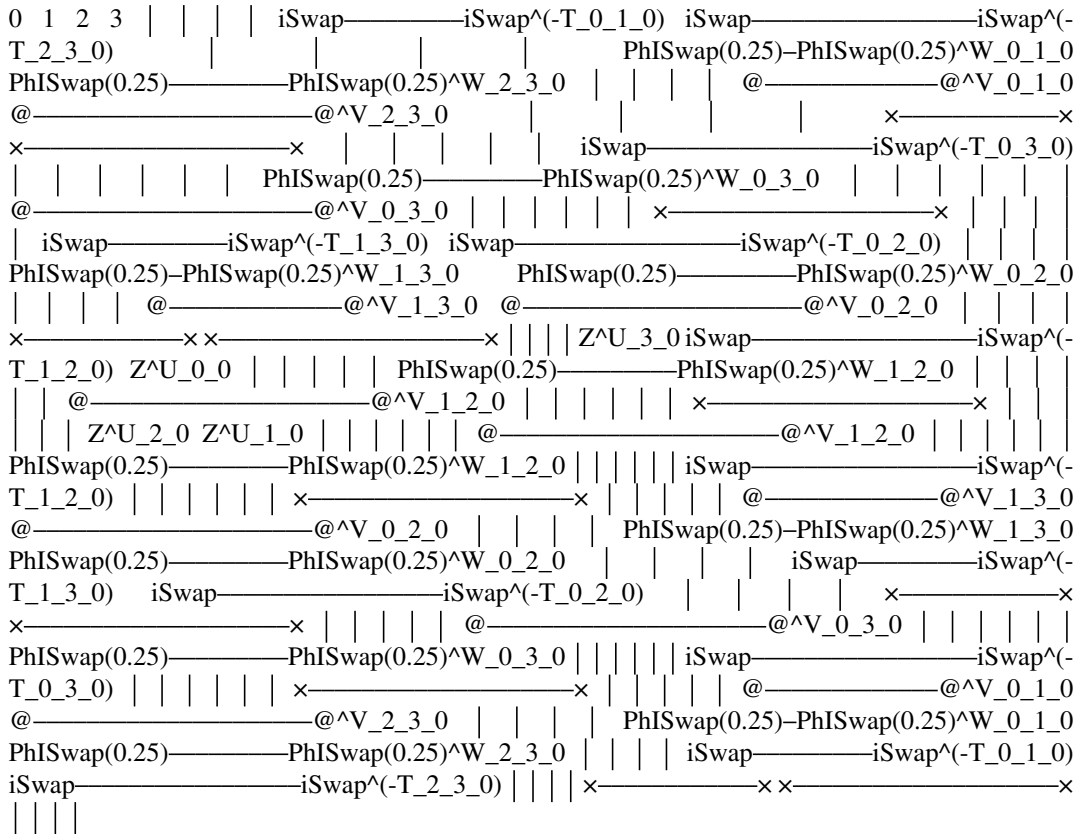
class openfermioncirq.**SwapNetworkTrotterAnsatz**(*hamiltonian: openfermion.ops._diagonal_coulomb_hamiltonian.DiagonalCoulomb iterations: int = 1, include_all_xxyy: bool = False, include_all_yxxy: bool = False, include_all_cz: bool = False, include_all_z: bool = False, adiabatic_evolution_time: Optional[float] = None, qubits: Optional[Sequence[cirq.ops.raw_types.Qid]] = None*)

An ansatz based on the fermionic swap network.

This ansatz uses as a template the form of a second-order Trotter step based on the "fermionic swap network" described in arXiv:1711.04789. The ansatz circuit and default initial parameters are determined by an instance of the DiagonalCoulombHamiltonian class.

Example: The ansatz on 4 qubits with one iteration and all gates included has the circuit:

```
# pylint: disable=line-too-long
```



pylint: enable=line-too-long

The Hamiltonian associated with the ansatz determines which ISWAP, PhasedISWAP, CZ, and Z gates are included. This basic template can be repeated, with each iteration introducing a new set of parameters.

The default initial parameters of the ansatz are chosen so that the ansatz circuit consists of a sequence of second-order Trotter steps approximating the dynamics of the time-dependent Hamiltonian H(t) = T + (t/A)V, where T

is the one-body term and V is the two-body term of the Hamiltonian used to generate the ansatz circuit, and t ranges from 0 to A and A is an adjustable value that defaults to the sum of the absolute values of the coefficients of the Jordan-Wigner transformed two-body operator V. The number of Trotter steps is equal to the number of iterations in the ansatz. This choice is motivated by the idea of state preparation via adiabatic evolution. The dynamics of H(t) are approximated as follows. First, the total evolution time of A is split into segments of length A / r, where r is the number of Trotter steps. Then, each Trotter step simulates H(t) for a time length of A / r, where t is the midpoint of the corresponding time segment. As an example, suppose A is 100 and the ansatz has two iterations. Then the approximation is achieved with two Trotter steps. The first Trotter step simulates H(25) for a time length of 50, and the second Trotter step simulates H(75) for a time length of 50.

__init__(*hamiltonian: openfermion.ops._diagonal_coulomb_hamiltonian.DiagonalCoulombHamiltonian*, *iterations: int = 1*, *include_all_xxyy: bool = False*, *include_all_yxxy: bool = False*, *include_all_cz: bool = False*, *include_all_z: bool = False*, *adiabatic_evolution_time: Optional[float] = None*, *qubits: Optional[Sequence[cirq.ops.raw_types.Qid]] = None*) → None

Parameters

- **hamiltonian** – The Hamiltonian used to generate the ansatz circuit and default initial parameters.
- **iterations** – The number of iterations of the basic template to include in the circuit. The number of parameters grows linearly with this value.
- **include_all_xxyy** – Whether to include all possible ISWAP-type parameterized gates in the ansatz (irrespective of the ansatz Hamiltonian)
- **include_all_yxxy** – Whether to include all possible PhasedISWAP-type parameterized gates in the ansatz (irrespective of the ansatz Hamiltonian)
- **include_all_cz** – Whether to include all possible CZ-type parameterized gates in the ansatz (irrespective of the ansatz Hamiltonian)
- **include_all_z** – Whether to include all possible Z-type parameterized gates in the ansatz (irrespective of the ansatz Hamiltonian)
- **adiabatic_evolution_time** – The time scale for Hamiltonian evolution used to determine the default initial parameters of the ansatz. This is the value A from the docstring of this class. If not specified, defaults to the sum of the absolute values of the entries of the two-body tensor of the Hamiltonian.
- **qubits** – Qubits to be used by the ansatz circuit. If not specified, then qubits will automatically be generated by the *_generate_qubits* method.

Methods

__init__(hamiltonian, iterations, ...)	**param hamiltonian** The Hamiltonian used to generate the ansatz
default_initial_params()	Approximate evolution by H(t) = T + (t/A)V.
operations(qubits)	Produce the operations of the ansatz circuit.
param_bounds()	Bounds on the parameters.
param_resolver(param_values)	Interprets parameters input as an array of real numbers.
param_scale_factors()	Coefficients to scale parameters by during optimization.

Continued on next page

Table 27 – continued from previous page

params()	The parameters of the ansatz.
qubit_permutation(qubits)	The qubit permutation induced by the ansatz circuit.

openfermioncirq.SplitOperatorTrotterAnsatz

class openfermioncirq.**SplitOperatorTrotterAnsatz**(*hamiltonian: openfermion.ops._diagonal_coulomb_hamiltonian.DiagonalCoulo iterations: int = 1, include_all_cz: bool = False, include_all_z: bool = False, adiabatic_evolution_time: Optional[float] = None, qubits: Optional[Sequence[cirq.ops.raw_types.Qid]] = None*)

An ansatz based on a split-operator Trotter step.

This ansatz uses as a template the form of a second-order Trotter step based on the split-operator simulation method described in arXiv:1706.00023. The ansatz circuit and default initial parameters are determined by an instance of the DiagonalCoulombHamiltonian class.

Example: The ansatz for a spinless jellium Hamiltonian on a 2x2 grid with one iteration has the circuit:

```
# pylint: disable=line-too-long
```

```
0    1    2    3    │    │    │    │    │    │    PhISwap(0.25)————PhISwap(0.25)^0.5    │
│    │    │    │    PhISwap(0.25)————PhISwap(0.25)^0.608    │    │    │    │    │    │
PhISwap(0.25)————PhISwap(0.25)^(-1/3)    │    │    │    PhISwap(0.25)—PhISwap(0.25)^(2/3)
│    │    │    │    │    │    Rz(-π)    PhISwap(0.25)————PhISwap(0.25)    │    │    │    │    │    Rz(π)    Rz(-π)
PhISwap(0.25)————PhISwap(0.25)^-0.392    │    │    │    │    Z^U_1_0    │    Rz(-π)    │    │    │    │    │    Rz(π)
Z^U_2_0 Z^U_3_0    │    │    │    │    │    │    Rz(π)    │    │    │    │    │    PhISwap(0.25)————PhISwap(0.25)^0.392
│    │    │    │    │    PhISwap(0.25)————PhISwap(0.25)^-1    │    │    │    │    PhISwap(0.25)—PhISwap(0.25)^(-
2/3)    │    │    │    │    │    │    │    PhISwap(0.25)————PhISwap(0.25)^(1/3)    │    │    │    │    │
PhISwap(0.25)————PhISwap(0.25)^-0.608    │    │    │    │    @————————@^V_0_1_0
│    │    │    │    │    │    ×————————×    PhISwap(0.25)————PhISwap(0.25)^-0.5    │    │    │    │    │    │
@————————@^V_2_3_0    │    │    │    │    │    ×————————×    │    │    │    │    │
@————————@^V_0_3_0    │    │    │    │    │    ×————————×    │    │    │    │    │
@————————@^V_1_3_0 @————————@^V_0_2_0    │    │    │    ×————————×
×————————×    │    │    │    │    @————————@^V_1_2_0    │    │    │    │    │    │
×————————×    │    │    │    │    PhISwap(0.25)—PhISwap(0.25)^0.5    │    │    │    │    │    │
PhISwap(0.25)————PhISwap(0.25)^0.608    │    │    │    │    PhISwap(0.25)—PhISwap(0.25)^(-
1/3)    │    │    │    │    │    │    │    PhISwap(0.25)————PhISwap(0.25)^(2/3)    │    │    │    │    │
PhISwap(0.25)————PhISwap(0.25) Rz(-π)    │    │    │    PhISwap(0.25)—PhISwap(0.25)^-0.392 Rz(-
π) Rz(π)    │    │    │    Rz(-π)    Z^U_1_0    │    │    │    Z^U_3_0 Z^U_2_0 Rz(π)    │    │    │    Rz(π)    │    │    │    │    │    │
PhISwap(0.25)—PhISwap(0.25)^0.392    │    │    │    │    │    PhISwap(0.25)————PhISwap(0.25)^-1    │    │    │
│    │    │    PhISwap(0.25)————PhISwap(0.25)^(-2/3)    │    │    │    PhISwap(0.25)—PhISwap(0.25)^(1/3)
│    │    │    │    │    │    PhISwap(0.25)————PhISwap(0.25)^-0.608    │    │    │    │    │
PhISwap(0.25)—PhISwap(0.25)^-0.5    │    │    │    │    │
```

pylint: enable=line-too-long

This basic template can be repeated, with each iteration introducing a new set of parameters.

The default initial parameters of the ansatz are chosen so that the ansatz circuit consists of a sequence of second-order Trotter steps approximating the dynamics of the time-dependent Hamiltonian H(t) = T + (t/A)V, where T is the one-body term and V is the two-body term of the Hamiltonian used to generate the ansatz circuit, and t ranges from 0 to A and A is an adjustable value that defaults to the sum of the absolute values of the coefficients

of the Jordan-Wigner transformed two-body operator V. The number of Trotter steps is equal to the number of iterations in the ansatz. This choice is motivated by the idea of state preparation via adiabatic evolution. The dynamics of H(t) are approximated as follows. First, the total evolution time of A is split into segments of length A / r, where r is the number of Trotter steps. Then, each Trotter step simulates H(t) for a time length of A / r, where t is the midpoint of the corresponding time segment. As an example, suppose A is 100 and the ansatz has two iterations. Then the approximation is achieved with two Trotter steps. The first Trotter step simulates H(25) for a time length of 50, and the second Trotter step simulates H(75) for a time length of 50.

__init__(*hamiltonian: openfermion.ops._diagonal_coulomb_hamiltonian.DiagonalCoulombHamiltonian*, *iterations: int = 1*, *include_all_cz: bool = False*, *include_all_z: bool = False*, *adiabatic_evolution_time: Optional[float] = None*, *qubits: Optional[Sequence[cirq.ops.raw_types.Qid]] = None*) → None

Parameters

- **hamiltonian** – The Hamiltonian used to generate the ansatz circuit and default initial parameters.
- **iterations** – The number of iterations of the basic template to include in the circuit. The number of parameters grows linearly with this value.
- **include_all_cz** – Whether to include all possible CZ-type parameterized gates in the ansatz (irrespective of the ansatz Hamiltonian)
- **include_all_z** – Whether to include all possible Z-type parameterized gates in the ansatz (irrespective of the ansatz Hamiltonian)
- **adiabatic_evolution_time** – The time scale for Hamiltonian evolution used to determine the default initial parameters of the ansatz. This is the value A from the docstring of this class. If not specified, defaults to the sum of the absolute values of the entries of the two-body tensor of the Hamiltonian.
- **qubits** – Qubits to be used by the ansatz circuit. If not specified, then qubits will automatically be generated by the *_generate_qubits* method.

Methods

__init__(hamiltonian, iterations, ...)	**param hamiltonian** The Hamiltonian used to generate the ansatz
default_initial_params()	Approximate evolution by H(t) = T + (t/A)V.
operations(qubits)	Produce the operations of the ansatz circuit.
param_bounds()	Bounds on the parameters.
param_resolver(param_values)	Interprets parameters input as an array of real numbers.
param_scale_factors()	Coefficients to scale parameters by during optimization.
params()	The names of the parameters of the ansatz.
qubit_permutation(qubits)	The qubit permutation induced by the ansatz circuit.

1.5 Optimization

optimization.OptimizationAlgorithm(options)	An optimization algorithm for black-box objective functions.
optimization.OptimizationParams(algorithm, ...)	Parameters for an optimization run.
optimization.OptimizationResult(...)	A result from optimizing a black-box objective function.
optimization.OptimizationTrialResult(...)	The results from multiple repetitions of an optimization run.
optimization.ScipyOptimizationAlgorithm(...)	An optimization algorithm from the scipy.optimize module.
optimization.COBYLA	An optimization algorithm from the scipy.optimize module.
optimization.L_BFGS_B	An optimization algorithm from the scipy.optimize module.
optimization.NELDER_MEAD	An optimization algorithm from the scipy.optimize module.
optimization.SLSQP	An optimization algorithm from the scipy.optimize module.

1.5.1 openfermioncirq.optimization.OptimizationAlgorithm

class openfermioncirq.optimization.**OptimizationAlgorithm**(*options: Optional[Any] = None*)

An optimization algorithm for black-box objective functions.

We use the convention that the optimization algorithm should try to minimize (rather than maximize) the value of the objective function.

In order to work with some routines that save optimization results, instances of this class must be picklable. See https://docs.python.org/3/library/pickle.html for details.

options
Options for the algorithm.

__init__(*options: Optional[Any] = None*) → None

Parameters **options** – Options for the algorithm.

Methods

__init__(options)	**param options** Options for the algorithm.
default_options()	Default options for the algorithm.
optimize(black_box, initial_guess, ...)	Perform the optimization and return the result.

Attributes

name	A name for the optimization algorithm.

1.5.2 openfermioncirq.optimization.OptimizationParams

class openfermioncirq.optimization.**OptimizationParams**(*algorithm: openfermioncirq.optimization.algorithm.OptimizationAlgorithm*, *initial_guess: Optional[numpy.ndarray] = None*, *initial_guess_array: Optional[numpy.ndarray] = None*, *cost_of_evaluate: Optional[float] = None*)

Parameters for an optimization run.

algorithm
The algorithm to use.

initial_guess
An initial guess for the algorithm to use.

initial_guess_array
An array of initial guesses for the algorithm to use. This is a 2d numpy array with each row representing one initial point.

cost_of_evaluate
A cost value associated with the *evaluate* method of the BlackBox to be optimized. For use with black boxes with a noise and cost model.

__init__(*algorithm: openfermioncirq.optimization.algorithm.OptimizationAlgorithm*, *initial_guess: Optional[numpy.ndarray] = None*, *initial_guess_array: Optional[numpy.ndarray] = None*, *cost_of_evaluate: Optional[float] = None*) → None
Construct a parameters object by setting its attributes.

Methods

__init__(algorithm, initial_guess, ...)	Construct a parameters object by setting its attributes.

1.5.3 openfermioncirq.optimization.OptimizationResult

class openfermioncirq.optimization.**OptimizationResult**(*optimal_value: float, optimal_parameters: numpy.ndarray, num_evaluations: Optional[int] = None, cost_spent: Optional[float] = None, function_values: Optional[List[Tuple[float, Optional[float], Optional[numpy.ndarray]]]] = None, wait_times: Optional[List[float]] = None, time: Optional[int] = None, seed: Optional[int] = None, status: Optional[int] = None, message: Optional[str] = None*)

A result from optimizing a black-box objective function.

optimal_value
The best value of the objective function found by the optimizer.

optimal_parameters
The inputs to the objective function which yield the optimal value.

num_evaluations
The number of times the objective function was evaluated in the course of the optimization.

cost_spent
For objective functions with a cost model, the total cost spent on function evaluations.

function_values
A list of tuples storing function values of evaluated points. The tuples contain three objects. The first is a function value, the second is the cost that was used for the evaluation (or None if there was no cost), and the third is the point that was evaluated (or None if the black box was initialized with *save_x_vals* set to False).

wait_times
A list of floats. The i-th float float represents the time elapsed between the i-th and (i+1)-th times that the black box was queried. Time is recorded using `time.time()`.

time
The time, in seconds, it took to obtain the result.

seed
A random number generator seed used to produce the result.

status
A status flag set by the optimizer.

message
A message returned by the optimizer.

__init__(*optimal_value: float, optimal_parameters: numpy.ndarray, num_evaluations: Optional[int] = None, cost_spent: Optional[float] = None, function_values: Optional[List[Tuple[float, Optional[float], Optional[numpy.ndarray]]]] = None, wait_times: Optional[List[float]] = None, time: Optional[int] = None, seed: Optional[int] = None, status: Optional[int] = None, message: Optional[str] = None*) → None

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(optimal_value, optimal_parameters, ...)	Initialize self.

1.5.4 openfermioncirq.optimization.OptimizationTrialResult

class openfermioncirq.optimization.**OptimizationTrialResult**(*results: Iterable[openfermioncirq.optimization.result.Optim params: OptimizationParams*)

The results from multiple repetitions of an optimization run.

data_frame
A pandas DataFrame storing the results of each repetition of the optimization run. It has the following columns:

optimal_value: The optimal value found. optimal_parameters: The function input corresponding to the

optimal value.

num_evaluations: The number of function evaluations used by the optimization algorithm.

cost_spent: The total cost spent on function evaluations. seed: A random number generator seed used by the repetition. status: A status returned by the optimization algorithm. message: A message returned by the optimization algorithm. time: The time it took for the repetition to complete. average_wait_time: The average time used by the optimizer to

decide on the next evaluation point.

params
An OptimizationParams object storing the optimization parameters used to obtain the results.

repetitions
The number of times the optimization run was repeated.

optimal_value
The optimal value over all repetitions of the run.

optimal_parameters
The function parameters corresponding to the optimal value.

__init__(*results: Iterable[openfermioncirq.optimization.result.OptimizationResult], params: OptimizationParams*) → None
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(results, params)	Initialize self.
extend(results)	

Attributes

optimal_parameters
optimal_value
repetitions

1.5.5 openfermioncirq.optimization.ScipyOptimizationAlgorithm

class openfermioncirq.optimization.**ScipyOptimizationAlgorithm**(*options: Optional[Dict] = None*, *kwargs: Optional[Dict] = None*, *uses_bounds: bool = True*)

An optimization algorithm from the scipy.optimize module.

__init__(*options: Optional[Dict] = None*, *kwargs: Optional[Dict] = None*, *uses_bounds: bool = True*) → None

Parameters

- **options** – The *options* dictionary passed to scipy.optimize.minimize.
- **kwargs** – Other keyword arguments passed to scipy.optimize.minimize. This should NOT include the *bounds* or *options* keyword arguments.
- **uses_bounds** – Whether the algorithm uses bounds on the input variables. Set this to False to prevent scipy.optimize.minimize from raising a warning if the chosen method does not use bounds.

Methods

__init__(options, kwargs, uses_bounds)	**param options** The *options* dictionary passed to scipy.optimize.minimize.
default_options()	Default options for the algorithm.
optimize(black_box, initial_guess, ...)	Perform the optimization and return the result.

Attributes

name	A name for the optimization algorithm.

1.5.6 openfermioncirq.optimization.COBYLA

openfermioncirq.optimization.**COBYLA = <openfermioncirq.optimization.scipy.ScipyOptimizatio**

An optimization algorithm from the scipy.optimize module.

1.5.7 openfermioncirq.optimization.L_BFGS_B

openfermioncirq.optimization.**L_BFGS_B = <openfermioncirq.optimization.scipy.ScipyOptimizat**
An optimization algorithm from the scipy.optimize module.

1.5.8 openfermioncirq.optimization.NELDER_MEAD

openfermioncirq.optimization.**NELDER_MEAD = <openfermioncirq.optimization.scipy.ScipyOptimi**
An optimization algorithm from the scipy.optimize module.

1.5.9 openfermioncirq.optimization.SLSQP

openfermioncirq.optimization.**SLSQP = <openfermioncirq.optimization.scipy.ScipyOptimization**
An optimization algorithm from the scipy.optimize module.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

C

D

F

T

V

W