QCompute-QAPP Documentation

Release 0.0.1

Institute for Quantum Computing, Baidu Inc.

Apr 13, 2023

API DOCS

CHAPTER

ONE

QCOMPUTE-QAPP PACKAGE

1.1 qcompute_qapp.algorithm

The variational quantum algorithm library

class qcompute_qapp.algorithm.**VQE**(*num: int*, *hamiltonian: list*, *ansatz:* ParameterizedCircuit, *optimizer:* BasicOptimizer, *backend: str*, *measurement: str = 'default'*)

Bases: `object`

Variational Quantum Eigensolver class

The constructor of the VQE class

Parameters

- **num** (*int*) – Number of qubits
- **hamiltonian** (*list*) – Hamiltonian whose minimum eigenvalue is to be solved
- **ansatz** (ParameterizedCircuit) – Ansatz used to search for the ground state of the Hamiltonian
- **optimizer** (BasicOptimizer) – Optimizer used to optimize the parameters in the ansatz
- **backend** (*str*) – Backend to be used in this task. Please refer to https://quantum-hub.baidu.com/quickGuide for details
- **measurement** (*str*) – Method chosen from 'default', 'ancilla', and 'SimMeasure' for measuring the expectation value, defaults to 'default'

get_measure(*shots: int = 1024*) → dict

Returns the measurement results

Parameters
shots (*int*) – Number of measurement shots, defaults to 1024

Returns
Measurement results in bitstrings with the number of counts

Return type
dict

get_gradient(*shots: int = 1024*) → ndarray

Calculates the gradient with respect to current parameters in circuit

Parameters
shots (*int*) – Number of measurement shots, defaults to 1024

Returns
Gradient with respect to current parameters

Return type
np.ndarray

`get_loss`(*shots: int = 1024*) → float

Calculates the loss with respect to current parameters in circuit

Parameters
shots (*int*) – Number of measurement shots, defaults to 1024

Returns
Loss with respect to current parameters

Return type
float

`run`(*shots: int = 1024*) → None

Searches for the minimum eigenvalue of the input Hamiltonian with the given ansatz and optimizer

Parameters
shots (*int*) – Number of measurement shots, defaults to 1024

`property minimum_eigenvalue: Union[str, float]`

The optimized minimum eigenvalue from last run

Returns
Optimized minimum eigenvalue from last run

Return type
Union[str, float]

`set_backend`(*backend: str*) → None

Sets the backend to be used

Parameters
backend (*str*) – Backend to be used

`class qcompute_qapp.algorithm.SSVQE`(*num: int, ex_num: int, hamiltonian: list, ansatz: ParameterizedCircuit, optimizer: BasicOptimizer, backend: str, measurement: str = 'default'*)

Bases: `object`

Subspace-Search Variational Quantum Eigensolver class

Please see https://journals.aps.org/prresearch/abstract/10.1103/PhysRevResearch.1.033062 for details on this algorithm.

The constructor of the SSVQE class

Parameters

- **num** (*int*) – Number of qubits
- **ex_num** (*int*) – Number of extra eignevalues to be solved. When ex_num = 0, only compute the minimum eigenvalue
- **hamiltonian** (*list*) – Hamiltonian whose eigenvalues are to be solved
- **ansatz** (ParameterizedCircuit) – Ansatz used to search for the eigenstates of the Hamiltonian
- **optimizer** (BasicOptimizer) – Optimizer used to optimize the parameters in the ansatz

- **backend** (*str*) – Backend to be used in this task. Please refer to https://quantum-hub.baidu.com/quickGuide for details
- **measurement** (*str*) – Method chosen from 'default', 'ancilla', and 'SimMeasure' for measuring the expectation value, defaults to 'default'

get_gradient(*shots: int = 1024*) → ndarray

Calculates the gradient with respect to current parameters in circuit

Parameters
shots (*int*) – Number of measurement shots, defaults to 1024

Returns
Gradient with respect to current parameters

Return type
np.ndarray

get_loss(*shots: int = 1024*) → float

Calculates the loss with respect to current parameters in circuit

Parameters
shots (*int*) – Number of measurement shots, defaults to 1024

Returns
Loss with respect to current parameters

Return type
float

run(*shots: int = 1024*) → None

Searches for the minimum eigenvalue of the input Hamiltonian with the given ansatz and optimizer

Parameters
shots (*int*) – Number of measurement shots, defaults to 1024

property minimum_eigenvalues: Union[str, list]

The optimized minimum eigenvalue from last run

Returns
Optimized minimum eigenvalues from last run

Return type
Union[str, list]

set_backend(*backend: str*) → None

Sets the backend to be used

Parameters
backend (*str*) – Backend to be used

class qcompute_qapp.algorithm.**QAOA**(*num: int*, *hamiltonian: list*, *ansatz:* QAOAAnsatz, *optimizer:* BasicOptimizer, *backend: str*, *measurement: str = 'default'*, *delta: float = 0.1*)

Bases: object

Quantum Approximate Optimization Algorithm class

The constructor of the QAOA class

Parameters

- **num** (*int*) – Number of qubits

- **hamiltonian** (*list*) – Hamiltonian used to construct the QAOA ansatz
- **ansatz** (QAOAAnsatz) – QAOA ansatz used to search for the maximum eigenstate of the Hamiltonian optimizer (BasicOptimizer): Optimizer used to optimize the parameters in the ansatz
- **backend** (*str*) – Backend to be used in this task. Please refer to https://quantum-hub.baidu.com/quickGuide for details
- **measurement** (*str*) – Method chosen from 'default', 'ancilla', and 'SimMeasure' for measuring the expectation value, defaults to 'default'
- **delta** (*float*) – Parameter used to calculate gradients, defaults to 0.1

`get_measure`(*shots: int = 1024*) → dict

Returns the measurement results

Parameters
shots (*int*) – Number of measurement shots, defaults to 1024

Returns
Measurement results in bitstrings with the number of counts

Return type
float

`get_gradient`(*shots: int = 1024*) → ndarray

Calculates the gradient with respect to current parameters in circuit

Parameters
shots (*int*) – Number of measurement shots, defaults to 1024

Returns
Gradient with respect to current parameters

Return type
np.ndarray

`get_loss`(*shots: int = 1024*) → float

Calculates the loss with respect to current parameters in circuit

Parameters
shots (*int*) – Number of measurement shots, defaults to 1024

Returns
Loss with respect to current parameters

Return type
float

`run`(*shots: int = 1024*) → None

Searches for the maximum eigenvalue of the input Hamiltonian with the given ansatz and optimizer

Parameters
shots (*int*) – Number of measurement shots, defaults to 1024

`property maximum_eigenvalue: Union[str, float]`

The optimized maximum eigenvalue from last run

Returns
Optimized maximum eigenvalue from last run

Return type
Union[str, float]

set_backend(*backend: str*) → None

Sets the backend to be used

Parameters
backend – Backend to be used

class qcompute_qapp.algorithm.**KernelClassifier**(*backend: str, encoding_style: str = 'IQP', kernel_type: str = 'qke', shots: int = 1024*)

Bases: object

Kernel Classifier class

The constructor of the KernelClassifier class

Parameters

- **backend** (*str*) – Backend to be used in this task. Please refer to https://quantum-hub.baidu.com/quickGuide for details
- **encoding_style** (*str*) – Encoding scheme to be used, defaults to 'IQP', which uses the default encoding scheme
- **kernel_type** (*str*) – Type of kernel to be used, defaults to 'qke', i.e., <x1|x2>
- **shots** (*int*) – Number of measurement shots, defaults to 1024

fit(*X: ndarray, y: ndarray*) → None

Trains the classifier with known data

Parameters

- **X** (*np.ndarray*) – Set of classical data vectors as the training data
- **y** (*np.ndarray*) – Known labels of the training data

predict(*x: ndarray*) → ndarray

Predicts labels of new data

Parameters
x (*np.ndarray*) – Set of data vectors with unknown labels

Returns
Predicted labels of the input data

Return type
np.ndarray

1.2 qcompute_qapp.application

1.2.1 qcompute_qapp.application.chemistry

The function about quantum chemistry

class qcompute_qapp.application.chemistry.**MolecularGroundStateEnergy**(*num_qubits: int = 0, hamiltonian: Optional[list] = None*)

Bases: `object`

Molecular Ground State Energy class

The constructor of the MolecularGroundStateEnergy class

Parameters

- **num_qubits** (*int*) – Number of qubits, defaults to 0
- **hamiltonian** (*list*) – Hamiltonian of the molecular system, defaults to None

property num_qubits: int

The number of qubits used to encoding this molecular system

Returns
Number of qubits

Return type
int

property hamiltonian: list

The Hamiltonian of this molecular system

Returns
Hamiltonian of this molecular system

Return type
list

compute_ground_state_energy() → float

Analytically computes the ground state energy

Returns
minimum real part of eigenvalues

Return type
float

load_hamiltonian_from_file(*filename: str*, *separator: str = ', '*) → None

Loads Hamiltonian from a file

Parameters

- **filename** (*str*) – Path to the file storing the Hamiltonian in Pauli terms
- **separator** (*str*) – Delimiter between coefficient and Pauli string, defaults to ', '

1.2.2 qcompute_qapp.application.optimization

The function about optimization

class qcompute_qapp.application.optimization.**MaxCut**(*num_qubits: int = 0*, *hamiltonian: Optional[list] = None*)

Bases: `object`

Max Cut Problem class

The constructor of the MaxCut class

Parameters

- **num_qubits** (*int*) – Number of qubits, defaults to 0
- **hamiltonian** (*list*) – Hamiltonian of the target graph of the Max Cut problem, defaults to None

property num_qubits: int

The number of qubits used to encoding this target graph

Returns
Number of qubits used to encoding this target graph

Return type
int

property hamiltonian: list

The Hamiltonian of this target graph

Returns
Hamiltonian of this target graph

Return type
list

graph_to_hamiltonian(*graph: Graph*) → None

Constructs Hamiltonian from the target graph of the Max Cut problem

Parameters
graph (*nx.Graph*) – Undirected graph without weights

decode_bitstring(*bitstring: str*) → dict

Decodes the measurement result into problem solution, i.e., set partition

Parameters
bitstring (*str*) – Measurement result with the largest probability

Returns
Solution to the Max Cut problem

Return type
dict

1.3 qcompute_qapp.circuit

The pre-defined quantum circuit

class qcompute_qapp.circuit.**BasicCircuit**(*num: int*)

Bases: ABC

Basic Circuit class

The constructor of the BasicCircuit class

Parameters
num (*int*) – Number of qubits

abstract add_circuit(*q: QRegPool*) → None

Adds circuit to the register.

Parameters
q (*QRegPool*) – Quantum register to which this circuit is added

class qcompute_qapp.circuit.**IQPEncodingCircuit**(*num: int*, *inverse: bool = False*)

Bases: *BasicCircuit*

IQP Encoding Circuit class

The constructor of the IQPEncodingCircuit class

Parameters

- **num** (*int*) – Number of qubits
- **inverse** (*bool*) – Whether the encoding circuit will be inverted, i.e. U^dagger(x) if True, defaults to False

add_circuit(*q: QRegPool*, *x: ndarray*) → None

Adds the encoding circuit used to map a classical data vector into its quantum feature state

Parameters

- **q** (*QRegPool*) – Quantum register to which this circuit is added
- **x** (*np.ndarray*) – Classical data vector to be encoded

class qcompute_qapp.circuit.**BasisEncodingCircuit**(*num: int*, *bit_string: str*)

Bases: *BasicCircuit*

Basis Encoding Circuit class

The constructor of the BasisEncodingCircuit class

Parameters

- **num** (*int*) – Number of qubits
- **bit_string** (*str*) – Bit string to be encoded as a quantum state

add_circuit(*q: QRegPool*) → None

Adds the basis encoding circuit to the register

Parameters

q (*QRegPool*) – Quantum register to which this circuit is added

class qcompute_qapp.circuit.**KernelEstimationCircuit**(*num: int*, *encoding_style: str*)

Bases: *BasicCircuit*

Kernel Estimation Circuit class

The constructor of the KernelEstimationCircuit class

Parameters

- **num** (*int*) – Number of qubits
- **encoding_style** (*str*) – Encoding circuit, only accepts `'IQP'` for now

add_circuit(*q: QRegPool*, *x1: ndarray*, *x2: ndarray*) → None

Adds the kernel estimation circuit used to evaluate the kernel entry value between two classical data vectors

Parameters

- **q** (*QRegPool*) – Quantum register to which this circuit is added
- **x1** (*np.ndarray*) – First classical vector
- **x2** (*np.ndarray*) – Second classical vector

class qcompute_qapp.circuit.**ParameterizedCircuit**(*num: int*, *parameters: ndarray*)

Bases: *BasicCircuit*

Parameterized Circuit class

The constructor of the BasicCircuit class

Parameters

- **num** (*int*) – Number of qubits
- **parameters** (*np.ndarray*) – Parameters of parameterized gates

property parameters: ndarray

Parameters of the circuit

Returns
Parameters of the circuit

Return type
np.ndarray

set_parameters(*parameters: ndarray*) → None

Sets parameters of the circuit

Parameters
parameters (*np.ndarray*) – New parameters of the circuit

abstract add_circuit(*q: QRegPool*) → None

Adds the circuit to the register

Parameters
q (*QRegPool*) – Quantum register to which this circuit is added

class qcompute_qapp.circuit.**PauliMeasurementCircuit**(*num: int*, *pauli_terms: str*)

Bases: *BasicCircuit*

Pauli Measurement Circuit class

The constructor of the PauliMeasurementCircuit class

Parameters

- **num** (*int*) – Number of qubits
- **pauli_terms** (*str*) – Pauli terms to be measured

add_circuit(*q: QRegPool*, *pauli_str: str*) → None

Adds the pauli measurement circuit to the register

Parameters

- **q** (*QRegPool*) – Quantum register to which this circuit is added
- **pauli_str** (*str*) – Pauli string to be measured

get_expectation(*preceding_circuits: List[BasicCircuit]*, *shots: int*, *backend: str*) → float

Computes the expectation value of the Pauli terms

Parameters

- **preceding_circuits** (*List[BasicCircuit]*) – Circuit precedes the measurement circuit
- **shots** (*int*) – Number of measurement shots

- **backend** (*str*) – Backend to be used in this task

Returns
Expectation value of the Pauli terms

Return type
float

class qcompute_qapp.circuit.**PauliMeasurementCircuitWithAncilla**(*num: int*, *pauli_terms: str*)

Bases: *BasicCircuit*

Pauli Measurement Circuit with Ancilla class

The constructor of the PauliMeasurementCircuitWithAncilla class

Parameters

- **num** (*int*) – Number of qubits
- **pauli_terms** (*str*) – Pauli terms to be measured

add_circuit(*q: QRegPool*, *pauli_str: str*) → None

Adds the pauli measurement circuit to the register

Parameters

- **q** (*QRegPool*) – Quantum register to which this circuit is added
- **pauli_str** (*str*) – Pauli string to be measured

get_expectation(*preceding_circuits: List[BasicCircuit]*, *shots: int*, *backend: str*) → float

Computes the expectation value of the Pauli terms

Parameters

- **preceding_circuits** (*List[BasicCircuit]*) – Circuit precedes the measurement circuit
- **shots** (*int*) – Number of measurement shots
- **backend** (*str*) – Backend to be used in this task

Returns
Expectation value of the Pauli terms

Return type
float

class qcompute_qapp.circuit.**SimultaneousPauliMeasurementCircuit**(*num: int*, *pauli_terms: list*)

Bases: *BasicCircuit*

Simultaneous Pauli Measurement Circuit for Qubitwise Commute Pauli Terms

The constructor of the SimultaneousPauliMeasurementCircuit class

Parameters

- **num** (*int*) – Number of qubits
- **pauli_terms** (*list*) – Pauli terms to be measured

add_circuit(*q: QRegPool*, *clique: list*) → None

Adds the simultaneous pauli measurement circuit to the register

Parameters

- **q** (*QRegPool*) – Quantum register to which this circuit is added

- **clique** (*list*) – Clique of Pauli terms to be measured together

get_expectation(*preceding_circuits: List[BasicCircuit], shots: int, backend: str*) → float

Computes the expectation value of the Pauli terms

Parameters

- **preceding_circuits** (*List[BasicCircuit]*) – Circuit precedes the measurement circuit
- **shots** (*int*) – Number of measurement shots
- **backend** (*str*) – Backend to be used in this task

Returns
Expectation value of the Pauli terms

Return type
float

class qcompute_qapp.circuit.**QAOAAnsatz**(*num: int, parameters: ndarray, hamiltonian: list, layer: int*)

Bases: *ParameterizedCircuit*

QAOA Ansatz class

The constructor of the QAOAAnsatz class

Parameters

- **num** (*int*) – Number of qubits in this ansatz
- **parameters** (*np.ndarray*) – Parameters of parameterized gates in this ansatz
- **hamiltonian** (*list*) – Hamiltonian used to construct the QAOA ansatz
- **layer** (*int*) – Number of layers for this Ansatz

add_circuit(*q: QRegPool*) → None

Adds circuit to the register according to the given hamiltonian

Parameters
q (*QRegPool*) – Quantum register to which this circuit is added

class qcompute_qapp.circuit.**UniversalCircuit**(*num: int, parameters: ndarray*)

Bases: *ParameterizedCircuit*

Universal Circuit class

The constructor of the UniversalCircuit class

Parameters

- **num** (*int*) – Number of qubits in this ansatz
- **parameters** (*np.ndarray*) – Parameters of parameterized gates in this circuit, whose shape should be `(3,)` for single-qubit cases and should be `(15,)` for 2-qubit cases

add_circuit(*q: QRegPool*) → None

Adds the universal circuit to the register. Only support single-qubit and 2-qubit cases

Parameters
q (*QRegPool*) – Quantum register to which this circuit is added

class qcompute_qapp.circuit.**RealEntangledCircuit**(*num: int*, *layer: int*, *parameters: ndarray*)

Bases: *ParameterizedCircuit*

Real Entangled Circuit class

The constructor of the RealEntangledCircuit class

Parameters

- **num** (*int*) – Number of qubits in this ansatz
- **layer** (*int*) – Number of layers for this ansatz
- **parameters** (*np.ndarray*) – Parameters of parameterized gates in this circuit, whose shape should be `(num * layer,)`

add_circuit(*q: QRegPool*) → None

Adds the real entangled circuit to the register

Parameters

q (*QRegPool*) – Quantum register to which this circuit is added

class qcompute_qapp.circuit.**ComplexEntangledCircuit**(*num: int*, *layer: int*, *parameters: ndarray*)

Bases: *ParameterizedCircuit*

Complex Entangled Circuit class

The constructor of the ComplexEntangledCircuit class

Parameters

- **num** (*int*) – Number of qubits in this Ansatz
- **layer** (*int*) – Number of layer for this Ansatz
- **parameters** (*np.ndarray*) – Parameters of parameterized gates in this circuit, whose shape should be `(num * layer * 2,)`

add_circuit(*q: QRegPool*) → None

Adds the complex entangled circuit to the register

Parameters

q (*QRegPool*) – Quantum register to which this circuit is added

class qcompute_qapp.circuit.**RealAlternatingLayeredCircuit**(*num: int*, *layer: int*, *parameters: ndarray*)

Bases: *ParameterizedCircuit*

Real Alternating Layered Circuit class

The constructor of the RealAlternatingLayeredCircuit class

Parameters

- **num** (*int*) – Number of qubits in this Ansatz
- **layer** (*int*) – Number of layer for this Ansatz
- **parameters** (*np.ndarray*) – Parameters of parameterized gates in this circuit, whose shape should be `((2 * num - 2) * layer,)`

add_circuit(*q: QRegPool*) → None

Adds the real alternating layered circuit to the register

Parameters

q (*QRegPool*) – Quantum register to which this circuit is added

class qcompute_qapp.circuit.**ComplexAlternatingLayeredCircuit**(*num: int*, *layer: int*, *parameters: ndarray*)

Bases: *ParameterizedCircuit*

Complex Alternating Layered Circuit class

The constructor of the ComplexAlternatingLayeredCircuit class

Parameters

- **num** (*int*) – Number of qubits in this Ansatz
- **layer** (*int*) – Number of layer for this Ansatz
- **parameters** (*np.ndarray*) – Parameters of parameterized gates in this circuit, whose shape should be `((4 * num - 4) * layer,)`

add_circuit(*q: QRegPool*) → None

Adds the complex alternating layered circuit to the register

Parameters

q (*QRegPool*) – Quantum register to which this circuit is added

1.4 qcompute_qapp.optimizer

The provided optimizer

class qcompute_qapp.optimizer.**BasicOptimizer**(*iterations: int*, *circuit: ParameterizedCircuit*)

Bases: `ABC`

Basic Optimizer class

The constructor of the BasicOptimizer class

Parameters

- **iterations** (*int*) – Number of iterations
- **circuit** (ParameterizedCircuit) – Circuit whose parameters are to be optimized

set_circuit(*circuit: ParameterizedCircuit*) → None

Sets the parameterized circuit to be optimized

Parameters

circuit (ParameterizedCircuit) – Parameterized Circuit to be optimized

abstract minimize(*shots: int*, *loss_func: Callable[[ndarray, int], float]*, *grad_func: Callable[[ndarray, int], ndarray]*) → None

Minimizes the given loss function

Parameters

- **shots** (*int*) – Number of measurement shots
- **loss_func** (*Callable[[np.ndarray, int], float]*) – Loss function to be minimized
- **grad_func** (*Callable[[np.ndarray, int], np.ndarray]*) – Function for calculating gradients

class `qcompute_qapp.optimizer.`**SGD**(*iterations: int*, *circuit:* BasicCircuit, *learning_rate: float*)

Bases: *BasicOptimizer*

SGD Optimizer class

The constructor of the SGD class

Parameters

- **iterations** (*int*) – Number of iterations
- **circuit** (BasicCircuit) – Circuit whose parameters are to be optimized
- **learning_rate** (*float*) – Learning rate

minimize(*shots: int*, *loss_func: Callable[[ndarray, int], float]*, *grad_func: Callable[[ndarray, int], ndarray]*) → None

Minimizes the given loss function

Parameters

- **shots** (*int*) – Number of measurement shots
- **loss_func** (*Callable[[np.ndarray, int], float]*) – Loss function to be minimized
- **grad_func** (*Callable[[np.ndarray, int], np.ndarray]*) – Function for calculating gradients

class `qcompute_qapp.optimizer.`**SLSQP**(*iterations: int*, *circuit:* BasicCircuit)

Bases: *BasicOptimizer*

SLSQP Optimizer class

The constructor of the SLSQP class

Parameters

- **iterations** (*int*) – Number of iterations
- **circuit** (BasicCircuit) – Circuit whose parameters are to be optimized

minimize(*shots: int*, *loss_func: Callable[[ndarray, int], float]*, *grad_func: Callable[[ndarray, int], ndarray]*) → None

Minimizes the given loss function

Parameters

- **shots** (*int*) – Number of measurement shots
- **loss_func** (*Callable[[np.ndarray, int], float]*) – Loss function to be minimized
- **grad_func** (*Callable[[np.ndarray, int], np.ndarray]*) – Function for calculating gradients

class `qcompute_qapp.optimizer.`**SPSA**(*iterations: int*, *circuit:* BasicCircuit, *a: float = 1.0*, *c: float = 1.0*)

Bases: *BasicOptimizer*

SPSA Optimizer class

The constructor of the SPSA class

Parameters

- **iterations** (*int*) – Number of iterations

- **circuit** (BasicCircuit) – Circuit whose parameters are to be optimized
- **a** (*float*) – Scaling parameter for step size, defaults to 1.0
- **c** (*float*) – Scaling parameter for evaluation step size, defaults to 1.0

minimize(*shots: int*, *loss_func: Callable[[ndarray, int], float]*, *grad_func: Callable[[ndarray, int], ndarray]*) → None

Minimizes the given loss function

Parameters

- **shots** (*int*) – Number of measurement shots
- **loss_func** (*Callable[[np.ndarray, int], float]*) – Loss function to be minimized
- **grad_func** (*Callable[[np.ndarray, int], np.ndarray]*) – Function for calculating gradients

class qcompute_qapp.optimizer.**SMO**(*iterations: int*, *circuit:* BasicCircuit)

Bases: *BasicOptimizer*

SMO Optimizer class

Please see https://arxiv.org/abs/1903.12166 for details on this optimization method.

The constructor of the SMO class

Parameters

- **iterations** (*int*) – Number of iterations
- **circuit** (BasicCircuit) – Circuit whose parameters are to be optimized

minimize(*shots: int*, *loss_func: Callable[[ndarray, int], float]*, *grad_func: Callable[[ndarray, int], ndarray]*) → None

Minimizes the given loss function

Parameters

- **shots** (*int*) – Number of measurement shots
- **loss_func** (*Callable[[np.ndarray, int], float]*) – Loss function to be minimized
- **grad_func** (*Callable[[np.ndarray, int], np.ndarray]*) – Function for calculating gradients

class qcompute_qapp.optimizer.**Powell**(*iterations: int*, *circuit:* BasicCircuit)

Bases: *BasicOptimizer*

Powell Optimizer class

The constructor of the Powell class

Parameters

- **iterations** (*int*) – Number of iterations
- **circuit** (BasicCircuit) – Circuit whose parameters are to be optimized

minimize(*shots: int*, *loss_func: Callable[[ndarray, int], float]*, *grad_func: Callable[[ndarray, int], ndarray]*) → None

Minimizes the given loss function

Parameters

- **shots** (*int*) – Number of measurement shots
- **loss_func** (*Callable[[np.ndarray, int], float]*) – Loss function to be minimized
- **grad_func** (*Callable[[np.ndarray, int], np.ndarray]*) – Function for calculating gradients

1.5 qcompute_qapp.utils

Some auxiliary functions

qcompute_qapp.utils.**grouping_hamiltonian**(*hamiltonian: list*, *coloring_strategy: str = 'largest_first'*) → List[List[str]]

Finds the minimum clique cover of the Hamiltonian graph, which is used for simultaneous Pauli measurement

Parameters

- **hamiltonian** (*list*) – Hamiltonian of the target system
- **coloring_strategy** (*str*) – Graph coloring strategy chosen from the following: 'largest_first', 'random_sequential', 'smallest_last', 'independent_set', 'connected_sequential_bfs', 'connected_sequential_dfs', 'connected_sequential', 'saturation_largest_first', and 'DSATUR'; defaults to 'largest_first'

Returns
List of cliques consisting of Pauli strings to be measured together

Return type
List[List[str]]

qcompute_qapp.utils.**pauli_terms_to_matrix**(*pauli_terms: list*) → ndarray

Converts Pauli terms to a matrix

Parameters
pauli_terms (*list*) – Pauli terms whose matrix is to be computed

Returns
Matrix form of the Pauli terms

Return type
np.ndarray

PYTHON MODULE INDEX

q

INDEX

A

B

C

D

F

G

S

U

V