HybridQ Documentation

Contents

Module `hybridq`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Sub-modules

- hybridq.base
- hybridq.circuit
- hybridq.dm
- hybridq.extras
- hybridq.gate
- hybridq.utils

Module `hybridq.base`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Sub-modules

- hybridq.base.base
- hybridq.base.property

Module `hybridq.base.base`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function compare

```
def compare(
    staticvars: {str, tuple[str, ...]},
    cmp: dict[str, any] = None
)
```

Function generate

```
def generate(
    class_name: str,
    mro: iter[type],
    methods: dict[str, any] = None,
    **staticvars
)
```

Generate new type.

Parameters

`class_name` **: str** Name of the new type. It must be a valid identified.
`mro` **: iter[type]** A series of types to derive from.
`methods` **: dict[str, any]** Extra method to add to class.

Returns

type The new type.

Function requires

```
def requires(
    names: {str, tuple[str, ...]}
)
```

Add requires static variables.

Function staticvars

```
def staticvars(
    staticvars: {str, tuple[str, ...]},
    check: dict[str, any] = None,
    transform: dict[str, any] = None,
    **defaults
)
```

Decorator for classes to add static variables to them.

Module hybridq.base.property

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Classes

Class Name

```
class Name
```

Add name to a object.

Ancestors (in MRO)

- hybridq.base.base.__Base__

Descendants

- hybridq.extras.gate.gate.MessageGate

Class Params

```
class Params(
    params: iter[any] = None,
    **kwargs
)
```

Add parameters to class.

Attributes

params : iter[any], optional

Ancestors (in MRO)

- hybridq.base.base.__Base__

Descendants

- hybridq.gate.property.ParamGate

Instance variables

Variable `params` Type: `tuple[any]`

Methods

Method `set_params`

```
def set_params(
    self,
    params: iter[any],
    *,
    inplace: bool = False
) -> Params
```

Return Params with given params. If inplace is True, Params is modified in place.

Parameters

`params` **: iter[any]** Parameters used to define the new Params.
`inplace` **: bool, optional** If True, Params is modified in place. Otherwise, a new Params is returned.

Returns

Params New Params with params. If inplace is True, Params is modified in place.

Class `Tags`

```
class Tags(
    tags: dict[any, any] = None,
    **kwargs
)
```

Add tags to a object.

Attributes

`tags` **: dict[any, any], optional** Dictionary of tags.

Ancestors (in MRO)

- hybridq.base.base.__Base__

Descendants

- hybridq.extras.gate.gate.MessageGate
- hybridq.gate.property.BaseTupleGate

Instance variables

Variable `tags` Type: `dict[any, any]`

Methods

Method `remove_tag`

```
def remove_tag(
    self,
    key: any,
    *,
    inplace: bool = False
) -> hybridq.base.property.Tags
```

Return Tags with removed tag mathcing key. If inplace is True, Tags is modified in place.

Parameters

key : any Key to remove from tags.
inplace : bool, optional If True, Tags is modified in place. Otherwise, a new Tags is returned.

Returns

Tags New Tags with key in tags removed. If inplace is True, Tags is modified in place.

Method `remove_tags`

```
def remove_tags(
    self,
    keys: iter[any],
    *,
    inplace: bool = False
) -> Tags
```

Return Tags with removed tags matching keys. If inplace is True, Tags is modified in place.

Parameters

keys : iter[any] Keys to remove from tags.
inplace : bool, optional If True, Tags is modified in place. Otherwise, a new Tags is returned.

Returns

Tags New Tags with keys in tags removed. If inplace is True, Tags is modified in place.

Method `set_tags`

```
def set_tags(
    self,
    tags: dict[any, any] = None,
    *,
    inplace: bool = False
) -> Tags
```

Return Tags with given tags. All previous tags are removed and substituted with tags. If inplace is True, Tags is modified in place.

Parameters

tags : dict[any, any] Parameters used to define the new Tags.
inplace : bool, optional If True, Tags is modified in place. Otherwise, a new Tags is returned.

Returns

Tags New Tags with tags. If inplace is True, Tags is modified in place.

Method `update_tags`

```
def update_tags(
    self,
    *args,
    inplace: bool = False,
    **kwargs
) -> hybridq.base.property.Tags
```

Return Tags with updated tags. If inplace is True, Tags is modified in place.

Parameters

`inplace` : bool, optional If True, Tags is modified in place. Otherwise, a new Tags is returned.

Returns

Tags New Tags with updated tags. If inplace is True, Tags is modified in place.

Class `Tuple`

```
class Tuple(
    elements=(),
    **kwargs
)
```

Tuple class for __Base__.

Ancestors (in MRO)

- hybridq.base.base.__Base__

Descendants

- hybridq.gate.property.BaseTupleGate

Instance variables

Variable `elements`

Methods

Method `flatten`

```
def flatten(
    self
) -> hybridq.base.property.Tuple
```

Return a flattend Tuple.

Method `index`

```
def index(
    self,
    *args,
    **kwargs
)
```

Module `hybridq.circuit`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Sub-modules

- hybridq.circuit.circuit
- hybridq.circuit.simulation
- hybridq.circuit.utils

Module `hybridq.circuit.circuit`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Classes

Class `BaseCircuit`

```
class BaseCircuit(
    gates: iter[Gate] = None,
    copy: bool = False
)
```

Class representing a circuit.

Attributes

`gates` **: iter[Gate], optional** Gates to be added to Circuit.
`copy` **: bool, optional** If True, every gate is copied using deepcopy.

Example

```
>>> c = Circuit(Gate('H', qubits=[q]) for q in range(10))
>>> c
Circuit([
        Gate(name=H, qubits=[0])
        Gate(name=H, qubits=[1])
        Gate(name=H, qubits=[2])
        Gate(name=H, qubits=[3])
        Gate(name=H, qubits=[4])
        Gate(name=H, qubits=[5])
        Gate(name=H, qubits=[6])
        Gate(name=H, qubits=[7])
        Gate(name=H, qubits=[8])
        Gate(name=H, qubits=[9])
])
>>> c + [Gate('X')]
Circuit([
        Gate(name=H, qubits=[0])
        Gate(name=H, qubits=[1])
        Gate(name=H, qubits=[2])
        Gate(name=H, qubits=[3])
        Gate(name=H, qubits=[4])
        Gate(name=H, qubits=[5])
        Gate(name=H, qubits=[6])
        Gate(name=H, qubits=[7])
        Gate(name=H, qubits=[8])
        Gate(name=H, qubits=[9])
])
>>> c[5:2:-1]
Circuit([
        Gate(name=H, qubits=[5])
        Gate(name=H, qubits=[4])
        Gate(name=H, qubits=[3])
])
```

Ancestors (in MRO)

- builtins.list

Descendants

- hybridq.circuit.circuit.Circuit
- hybridq.dm.circuit.circuit.Circuit

Methods

Method `all_tags`

```
def all_tags(
    self
) -> list[dict]
```

Return a list of all tags in each Gate.

Returns

list[dict] List of all tags in each Gate.

Example

```
>>> Circuit(Gate('X', tags={q:q}) for q in range(10)).all_tags()
[{0: 0},
 {1: 1},
 {2: 2},
 {3: 3},
 {4: 4},
 {5: 5},
 {6: 6},
 {7: 7},
 {8: 8},
 {9: 9}]
```

Method `append`

```
def append(
    self,
    gate: Gate
) -> None
```

Append Gate to existing Circuit.

Parameters

`gate` **: Gate** Gate to append.

Example

```
>>> c = Circuit()
>>> c.append(Gate('H'))
>>> c
Circuit([
        Gate(name=H)
])
```

Method `extend`

```
def extend(
    self,
    circuit: iter[Gate]
) -> None
```

Extend existing Circuit.

Parameters

circuit : iter[Gate] Extend Circuit using circuit.

Example

```
>>> c = Circuit()
>>> c.extend(Gate('X', qubits=[q]) for q in range(10))
Circuit([
        Gate(name=X, qubits=[0])
        Gate(name=X, qubits=[1])
        Gate(name=X, qubits=[2])
        Gate(name=X, qubits=[3])
        Gate(name=X, qubits=[4])
        Gate(name=X, qubits=[5])
        Gate(name=X, qubits=[6])
        Gate(name=X, qubits=[7])
        Gate(name=X, qubits=[8])
        Gate(name=X, qubits=[9])
])
```

Method `remove_all_tags`

```
def remove_all_tags(
    self,
    keys: iter[any],
    *,
    inplace: bool = False
) -> Circuit
```

Remove all tags matching keys from all Gates. If inplace is True, Circuit is modified in place.

Parameters

keys : iter[any] Keys to remove from tags.
inplace : bool, optional If True, Circuit is modified in place. Otherwise, a new Circuit is returned.

Returns

Circuit Circuit with tags matching keys from all Gates removed. If inplace is True, Circuit is modified in place.

Example

```
>>> c = Circuit(Gate('H', tags={q%4:q}) for q in range(10))
>>> c
Circuit([
        Gate(name=H, tags={0: 0})
        Gate(name=H, tags={1: 1})
        Gate(name=H, tags={2: 2})
        Gate(name=H, tags={3: 3})
        Gate(name=H, tags={0: 4})
        Gate(name=H, tags={1: 5})
        Gate(name=H, tags={2: 6})
        Gate(name=H, tags={3: 7})
```

```
        Gate(name=H, tags={0: 8})
        Gate(name=H, tags={1: 9})
])
>>> c.remove_all_tags([0, 3])
Circuit([
        Gate(name=H)
        Gate(name=H, tags={1: 1})
        Gate(name=H, tags={2: 2})
        Gate(name=H)
        Gate(name=H)
        Gate(name=H, tags={1: 5})
        Gate(name=H, tags={2: 6})
        Gate(name=H)
        Gate(name=H)
        Gate(name=H, tags={1: 9})
])
```

Method `update_all_tags`

```
def update_all_tags(
    self,
    *args,
    inplace: bool = False,
    **kwargs
) -> hybridq.circuit.circuit.Circuit
```

Update all Gates' tags in Circuit. If inplace is True, Circuit is modified in place.

Parameters

`inplace` : bool, optional If True, Circuit is modified in place. Otherwise, a new Circuit is returned.

Returns

Circuit Circuit with update tags in all Gates. If inplace is True, Circuit is modified in place.

Example

```
>>> c = Circuit(Gate('H', tags={q%4:q}) for q in range(10))
>>> c
Circuit([
        Gate(name=H, tags={0: 0})
        Gate(name=H, tags={1: 1})
        Gate(name=H, tags={2: 2})
        Gate(name=H, tags={3: 3})
        Gate(name=H, tags={0: 4})
        Gate(name=H, tags={1: 5})
        Gate(name=H, tags={2: 6})
        Gate(name=H, tags={3: 7})
        Gate(name=H, tags={0: 8})
        Gate(name=H, tags={1: 9})
])
>>> c.update_all_tags({-1:'x', '42': 1.23})
Circuit([
        Gate(name=H, tags={0: 0, -1: 'x', '42': 1.23})
        Gate(name=H, tags={1: 1, -1: 'x', '42': 1.23})
```

```
        Gate(name=H, tags={2: 2, -1: 'x', '42': 1.23})
        Gate(name=H, tags={3: 3, -1: 'x', '42': 1.23})
        Gate(name=H, tags={0: 4, -1: 'x', '42': 1.23})
        Gate(name=H, tags={1: 5, -1: 'x', '42': 1.23})
        Gate(name=H, tags={2: 6, -1: 'x', '42': 1.23})
        Gate(name=H, tags={3: 7, -1: 'x', '42': 1.23})
        Gate(name=H, tags={0: 8, -1: 'x', '42': 1.23})
        Gate(name=H, tags={1: 9, -1: 'x', '42': 1.23})
])
```

Class `Circuit`

```
class Circuit(
    gates: iter[Gate] = (),
    *args,
    **kwargs
)
```

Class representing a circuit.

Attributes

`gates` **: iter[Gate], optional** Gates to be added to Circuit.
`copy` **: bool, optional** If True, every gate is copied using deepcopy.

Example

```
>>> c = Circuit(Gate('H', qubits=[q]) for q in range(10))
>>> c
Circuit([
        Gate(name=H, qubits=[0])
        Gate(name=H, qubits=[1])
        Gate(name=H, qubits=[2])
        Gate(name=H, qubits=[3])
        Gate(name=H, qubits=[4])
        Gate(name=H, qubits=[5])
        Gate(name=H, qubits=[6])
        Gate(name=H, qubits=[7])
        Gate(name=H, qubits=[8])
        Gate(name=H, qubits=[9])
])
>>> c + [Gate('X')]
Circuit([
        Gate(name=H, qubits=[0])
        Gate(name=H, qubits=[1])
        Gate(name=H, qubits=[2])
        Gate(name=H, qubits=[3])
        Gate(name=H, qubits=[4])
        Gate(name=H, qubits=[5])
        Gate(name=H, qubits=[6])
        Gate(name=H, qubits=[7])
        Gate(name=H, qubits=[8])
        Gate(name=H, qubits=[9])
])
>>> c[5:2:-1]
Circuit([
```

```
        Gate(name=H, qubits=[5])
        Gate(name=H, qubits=[4])
        Gate(name=H, qubits=[3])
])
```

Ancestors (in MRO)

- hybridq.circuit.circuit.BaseCircuit
- builtins.list

Methods

Method `T`

```
def T(
    self
) -> hybridq.circuit.circuit.Circuit
```

Return the transposed circuit of Circuit.

Returns

Circuit Transposition of Circuit.

Method `adj`

```
def adj(
    self
) -> hybridq.circuit.circuit.Circuit
```

Return the adjoint circuit of Circuit.

Returns

Circuit Adjoint of Circuit.

Method `all_qubits`

```
def all_qubits(
    self,
    *,
    ignore_missing_qubits: bool = False
) -> list[any]
```

Get all qubits in Circuit. It raises a ValueError if qubits in Gate are missing, unless ignore_missing_qubits is True. The returned qubits are always sorted using hybridq.utils.sort for consistency.

Parameters

`ignore_missing_qubits` : bool, optional If True, ignore gates without specified qubits. Otherwise, raise ValueError.

Returns

list[any] Sorted list of all qubits in Circuit.

Example

```
>>> Circuit([Gate('H', qubits=[2]), Gate('X', qubits=[1])]).all_qubits()
[1, 2]
>>> Circuit([Gate('H', qubits=[2]), Gate('X', qubits=[1]), Gate('H')]).all_qubits()
ValueError: Circuit contains virtual gates with no qubits.
>>> Circuit([Gate('H', qubits=[2]), Gate('X', qubits=[1]), Gate('H')]).all_qubits(ignore_missing_qubit
[1, 2]
```

Method `conj`

```
def conj(
    self
) -> hybridq.circuit.circuit.Circuit
```

Return the conjugate circuit of Circuit.

Returns

Circuit Conjugation of Circuit.

Method `inv`

```
def inv(
    self
) -> hybridq.circuit.circuit.Circuit
```

Return the inverse circuit of Circuit.

Returns

Circuit Inverse of Circuit.

Example

```
>>> from numpy.random import random
>>> from hybridq.circuit.utils import simplify
>>> c = Circuit(Gate('RX', qubits=[q], params=[random()]) for q in range(10))
>>> simplify(c + c.inv())
Circuit([
])
```

Module `hybridq.circuit.simulation`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Sub-modules

- hybridq.circuit.simulation.clifford
- hybridq.circuit.simulation.simulation
- hybridq.circuit.simulation.simulation_mpi
- hybridq.circuit.simulation.utils

Module `hybridq.circuit.simulation.clifford`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function `expectation_value`

```
def expectation_value(
    circuit: Circuit,
    op: Circuit,
    initial_state: str,
    **kwargs
) -> float
```

Compute the expectation value of op for the given circuit, using initial_state as initial state.

Parameters

`circuit` : Circuit Circuit to simulate.

`op` : Circuit Operator used to compute the expectation value. op must be a valid Circuit containing only Pauli gates (that is, either I, X, Y or Z gates) acting on different qubits.

`initial_state` : str Initial state used to compute the expectation value. Valid tokens for initial_state are:

- 0: qubit is set to 0 in the computational basis,
- 1: qubit is set to 1 in the computational basis,
- +: qubit is set to + state in the computational basis,
- −: qubit is set to − state in the computational basis.

Returns

float [, dict[any, any]] The expectation value of the operator op. If `return_info=True`, information gathered during the simulation are also returned.

Other Parameters

expectation_value() uses all valid parameters for update_pauli_string().

See Also

update_pauli_string()

Example

```
>>> # Define circuit
>>> circuit = Circuit(
>>>     [Gate('X', qubits=[0])**1.2,
>>>      Gate('ISWAP', qubits=[0, 1])**2.3])
>>>
>>> # Define operator
>>> op = Circuit([Gate('Z', qubits=[1])])
>>>
>>> # Get expectation value
>>> clifford.expectation_value(circuit=circuit,
>>>                            op=op,
>>>                            initial_state='11',
>>>                            float_type='float64')
-0.6271482580325515
```

Function `update_pauli_string`

```
def update_pauli_string(
    circuit: Circuit,
    pauli_string: {Circuit, dict[str, float]},
    phase: float = 1,
    parallel: {bool, int} = False,
    return_info: bool = False,
    use_mpi: bool = None,
    compress: int = 4,
    simplify: bool = True,
    remove_id_gates: bool = True,
    float_type: any = 'float32',
    verbose: bool = False,
    **kwargs
) -> defaultdict
```

Evolve density matrix accordingly to circuit using pauli_string as initial product state. The evolved density matrix will be represented as a set of different Pauli strings, each of them with a different phase, such that their sum corresponds to the evolved density matrix. The number of branches depends on the number of non-Clifford gates in circuit.

Parameters

`circuit` : Circuit Circuit to use to evolve pauli_string.

`pauli_string` : `{Circuit, dict[str, float]}` Pauli string to be evolved. pauli_string must be a Circuit composed of single qubit Pauli Gates (that is, either `Gate('I')`, `Gate('X')`, `Gate('Y')` or `Gate('Z')`), each one acting on every qubit of circuit. If a dictionary is provided, every key of pauli_string must be a valid Pauli string. The size of each Pauli string must be equal to the number of qubits in circuit. Values in pauli_string will be used as inital phase for the given string.

`phase` : float, optional Initial phase for pauli_string.

`atol` : float, optional Discard all Pauli strings that have an absolute amplitude smaller than atol.

`parallel` : int, optional Parallelize simulation (where possible). If True, the number of available cpus is used. Otherwise, a parallel number of threads is used.

`return_info` : bool Return extra information collected during the evolution.

`use_mpi` : bool, optional Use MPI if available. Unless `use_mpi=False`, MPI will be used if detected (for instance, if mpiexec is used to called HybridQ). If `use_mpi=True`, force the use of MPI (in case MPI is not automatically detected).

`compress` : **int, optional** Compress Circuit using utils.compress prior the simulation.
`simplify` : **bool, optional** Simplify Circuit using utils.simplify prior the simulation.
`remove_id_gates` : **bool, optional** Remove ID gates prior the simulation.
`float_type` : **any, optional** Float type to use for the simulation.
`verbose` : **bool, optional** Verbose output.

Returns

dict[str, float] [, dict[any, any]] If `return_info=False`, update_pauli_string() returns a dict of Pauli strings and the corresponding amplitude. The full density matrix can be reconstructed by resumming over all the Pauli string, weighted with the corresponding amplitude. If `return_info=True`, information gathered during the simulation are also returned.

Other Parameters

`eps` : **float, optional** `(default: auto)` Do not branch if the branch weight for the given non-Clifford operation is smaller than eps. `atol=1e-7` if `float_type=float32`, otherwise `atol=1e-8` if `float_type=float64`.
`atol` : **float, optional** `(default: auto)` Remove elements from final state if such element as an absolute amplitude smaller than atol. `atol=1e-8` if `float_type=float32`, otherwise `atol=1e-12` if `float_type=float64`.
`branch_atol` : **float, optional** Stop branching if the branch absolute amplitude is smaller than branch_atol. If not specified, it will be equal to atol.
`max_breadth_first_branches` : `int (default: auto)` Max number of branches to collect using breadth first search. The number of branches collect during the breadth first phase will be split among the different threads (or nodes if using MPI).
`n_chunks` : `int (default: auto)` Number of chunks to divide the branches obtained during the breadth first phase. The default value is twelve times the number of threads.
`max_virtual_memory` : `float (default: 80)` Max virtual memory (%) that can be using during the simulation. If the used virtual memory is above max_virtual_memory, update_pauli_string() will raise an error.
`sleep_time` : `float (default: 0.1)` Completition of parallel processes is checked every sleep_time seconds.

Example

```
>>> from hybridq.circuit import utils
>>> import numpy as np
>>>
>>> # Define circuit
>>> circuit = Circuit(
>>>     [Gate('X', qubits=[0])**1.2,
>>>      Gate('ISWAP', qubits=[0, 1])**2.3])
>>>
>>> # Define Pauli string
>>> pauli_string = Circuit([Gate('Z', qubits=[1])])
>>>
>>> # Get density matrix decomposed in Pauli strings
>>> dm = clifford.update_pauli_string(circuit=circuit,
>>>                                    pauli_string=pauli_string,
>>>                                    float_type='float64')
>>>
>>> dm
defaultdict(<function hybridq.circuit.simulation.clifford.update_pauli_string.<locals>._db_init.<loca
            {'IZ': 0.7938926261462365,
             'YI': -0.12114687473997318,
             'ZI': -0.166744368113685,
             'ZX': 0.2377641290737882,
             'YX': -0.3272542485937367,
             'XY': -0.40450849718747345})
```

```
>>> # Reconstruct density matrix
>>> U = sum(phase * np.kron(Gate(g1).matrix(),
>>>                         Gate(g2).matrix()) for (g1, g2), phase in dm.items())
>>>
>>> U
array([[ 0.62714826+0.j        ,  0.23776413+0.j        ,
         0.        +0.12114687j,  0.        +0.73176275j],
       [ 0.23776413+0.j        , -0.96063699+0.j        ,
         0.        -0.07725425j,  0.        +0.12114687j],
       [ 0.        -0.12114687j,  0.        +0.07725425j,
         0.96063699+0.j        , -0.23776413+0.j        ],
       [ 0.        -0.73176275j,  0.        -0.12114687j,
        -0.23776413+0.j        , -0.62714826+0.j        ]])
>>> np.allclose(utils.matrix(circuit + pauli_string + circuit.inv()),
>>>             U,
>>>             atol=1e-8)
True
>>> U[0b11, 0b11]
(-0.6271482580325515+0j)
```

Module `hybridq.circuit.simulation.simulation`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Types

Array: numpy.ndarray

TensorNetwork: quimb.tensor.TensorNetwork

ContractionInfo: (opt_einsum.contract.PathInfo, cotengra.hyper.HyperOptimizer)

Functions

Function `expectation_value`

```
def expectation_value(
    state: Array,
    op: Circuit,
    qubits_order: iter[any],
    complex_type: any = 'complex64',
    backend: any = 'numpy',
    verbose: bool = False,
    **kwargs
) -> complex
```

Compute expectation value of an operator given a quantum state.

Parameters

`state` : Array Quantum state to use to compute the expectation value of the operator op.
`op` : Circuit Quantum operator to use to compute the expectation value.
`qubits_order` : iter[any] Order of qubits used to map Circuit.qubits to state.
`complex_type` : any, optional Complex type to use to compute the expectation value.
`backend` : any, optional Backend used to compute the quantum state. Backend must have tensordot, transpose and einsum methods.
`verbose` : bool, optional Verbose output.

Returns

complex The expectation value of the operator op given state.

Other Parameters

expectation_value() accepts all valid parameters for simulate().

See Also

simulate()

Example

```
>>> op = Circuit([
>>>     Gate('H', qubits=[32]),
>>>     Gate('CX', qubits=[32, 42]),
>>>     Gate('RX', qubits=[12], params=[1.32])
>>> ])
>>> expectation_value(
>>>     state=prepare_state('+0-'),
>>>     op=op,
>>>     qubits_order=[12, 42, 32],
>>> )
array(0.55860883-0.43353909j)
```

Function `simulate`

```
def simulate(
    circuit: {Circuit, TensorNetwork},
    initial_state: any = None,
    final_state: any = None,
    optimize: any = 'evolution',
    backend: any = 'numpy',
    complex_type: any = 'complex64',
    tensor_only: bool = False,
    simplify: {bool, dict} = True,
    remove_id_gates: bool = True,
    use_mpi: bool = None,
    atol: float = 1e-08,
    verbose: bool = False,
    **kwargs
) -> any
```

Frontend to simulate Circuit using different optimization models and backends.

Parameters

`circuit` : `{Circuit, TensorNetwork}` Circuit to simulate.

`initial_state` : any, optional Initial state to use.

`final_state` : any, optional Final state to use (only valid for `optimize='tn'`).

`optimize` : any, optional Optimization to use. At the moment, HybridQ supports two optimizations: `optimize='evolution'` (equivalent to `optimize='evolution-hybridq'`) and `optimize='tn'` (equivalent to `optimize='cotengra'`). `optimize='evolution'` takes an initial_state (it can either be a string, which is processed using `prepare_state` or an Array) and evolve the quantum state accordingly to Circuit. Alternatives are:

- `optimize='evolution-hybridq'`: use internal C++ implementation for quantum state evolution that uses vectorization instructions (such as AVX instructions for Intel processors). This optimization method is best suitable for CPUs.
- `optimize='evolution-einsum'`: use einsum to perform the evolution of the quantum state (via opt_einsum). It is possible to futher specify optimization for opt_einsum by using `optimize='evolution-einsum-opt'` where opt is one of the available optimization in opt_einsum.contract (default: auto). This optimization is best suitable for GPUs and TPUs (using `backend='jax'`).

`optimize='tn'` (or, equivalently, `optimize='cotengra'`) performs the tensor contraction of Circuit given an initial_state and a final_state (both must be a str). Valid tokens for both initial_state and final_state are:

- 0: qubit is set to 0 in the computational basis,
- 1: qubit is set to 1 in the computational basis,
- +: qubit is set to + state in the computational basis,
- −: qubit is set to − state in the computational basis,
- .: qubit is left uncontracted.

Before the actual contraction, cotengra is called to identify an optimal contraction. Such contraction is then used to perform the tensor contraction.

If Circuit is a TensorNetwork, optimize must be a valid contraction (see tensor_only parameter).

`backend` : any, optional Backend used to perform the simulation. Backend must have tensordot, transpose and einsum methods.

`complex_type` : any, optional Complex type to use for the simulation.

`tensor_only` : bool, optional If True and `optimize=None`, simulate() will return a TensorNetwork representing Circuit. Otherwise, if `optimize='cotengra'`, simulate() will return the tuple `(TensorNetwork, ContractionInfo)`. `TensorNetwork` and and `ContractionInfo` can be respectively used as values for circuit and optimize to perform the actual contraction.

`simplify` : `{bool, dict}`, optional Circuit is simplified before the simulation using circuit.utils.simplify. If non-empty dict is provided, simplify is passed as arguments for circuit.utils.simplity.

`remove_id_gates` : bool, optional Identity gates are removed before to perform the simulation. If False, identity gates are kept during the simulation.

`use_mpi` : bool, optional Use MPI if available. Unless `use_mpi=False`, MPI will be used if detected (for instance, if mpiexec is used to called HybridQ). If `use_mpi=True`, force the use of MPI (in case MPI is not automatically detected).

`atol` : float, optional Use atol as absolute tollerance.

`verbose` : bool, optional Verbose output.

Returns

Output of simulate() depends on the chosen parameters.

Other Parameters

parallel : int (default: False) Parallelize simulation (where possible). If True, the number of available cpus is used. Otherwise, a parallel number of threads is used.

compress : {int, dict} (default: auto) Select level of compression for `circuit.utils.compress`, which is run on Circuit prior to perform the simulation. If non-empty dict is provided, compress is passed as arguments for circuit.utils.compress. If `optimize=evolution`, compress is set to 4 by default. Otherwise, if `optimize=tn`, compress is set to 2 by default.

allow_sampling : bool (default: False) If True, Gates that provide the method sample will not be sampled.

sampling_seed : int (default: None) If provided, numpy.random state will be saved before sampling and sampling_seed will be used to sample Gates. numpy.random state will be restored after sampling.

block_until_ready : bool (default: True) When `backend='jax'`, wait till the results are ready before returning.

return_numpy_array : bool (default: True) When `optimize='hybridq'` and return_numpy_array is False, a tuple` of two np.ndarray is returned, corresponding to the real and imaginary part of the quantu state. If True, the real and imaginary part are copied to a single np.ndarray of complex numbers.

return_info : bool (default: False) Return extra information collected during the simulation.

simplify_tn : str (default: 'RC') Simplification to apply to TensorNetwork. Available simplifications as specified in quimb.tensor.TensorNetwork.full_simplify.

max_largest_intermediate : int (default: 226)** Largest intermediate which is allowed during simulation. If `optimize='evolution'`, simulate() will raise an error if the largest intermediate is larger than max_largest_intermediate. If `optimize='tn'`, slicing will be applied to fit the contraction in memory.

target_largest_intermediate : int (default: 0) Stop cotengra if a contraction having the largest intermediate smaller than target_largest_intermediate is found.

max_iterations : int (default: 1) Number of cotengra iterations to find optimal contration.

max_time : int (default: 120) Maximum number of seconds allowed to cotengra to find optimal contraction for each iteration.

max_repeats : int (default: 16) Number of cotengra steps to find optimal contraction for each iteration.

temperatures : list[float] (default: [1.0, 0.1, 0.01]) Temperatures used by cotengra to find optimal slicing of the tensor network.

max_n_slices : int (default: None) If specified, simulate() will raise an error if the number of slices to fit the tensor contraction in memory is larger than max_n_slices.

minimize : str (default: 'combo') Cost function to minimize while looking for the best contraction (see cotengra for more information).

methods : list[str] (default: ['kahypar', 'greedy']) Heuristics used by cotengra to find optimal contraction.

optlib : str (default: 'baytune') Library used by cotengra to tune hyper-parameters while looking for the best contraction.

sampler : str (default: 'GP') Sampler used by cotengra while looking for the contraction.

cotengra : dict[any, any] (default: {}) Extra parameters to pass to cotengra.

Module `hybridq.circuit.simulation.simulation_mpi`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

MPI implementation of hybridq.circuit.simulation.simulate.

See Also

hybridq.circuit.simulate Simulate quantum circuit.

Module `hybridq.circuit.simulation.utils`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Types

Array: numpy.ndarray

Functions

Function `prepare_state`

```
def prepare_state(
    state: str,
    d: {int, iter[int]} = 2,
    complex_type: any = 'complex64'
)
```

Prepare initial state accordingly to state.

Parameters

`state` : str State used to prepare the quantum state. If state is a string, a quantum state of len(str) is created using the following notation:

- 0: qubit is set to 0 in the computational basis,
- 1: qubit is set to 1 in the computational basis,
- +: qubit is set to + state in the computational basis,
- −: qubit is set to − state in the computational basis.

`d` : `{int, iter[int]}` Dimensions of qubits.

`complex_type` : any, optional Complex type to use to prepare the quantum state.

Returns

Array Quantum state prepared from state.

Example

```
>>> prepare_state('+-+')
array([ 0.35355338+0.j,  0.35355338+0.j, -0.35355338+0.j, -0.35355338+0.j,
        0.35355338+0.j,  0.35355338+0.j, -0.35355338+0.j, -0.35355338+0.j],
      dtype=complex64)
```

Module `hybridq.circuit.utils`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function `compress`

```
def compress(
    circuit: iter[BaseGate],
    max_n_qubits: int = 2,
    *,
    exclude_qubits: iter[any] = None,
    use_matrix_commutation: bool = True,
    max_n_qubits_matrix: int = 10,
    skip_compression: iter[{type, str}] = None,
    skip_commutation: iter[{type, str}] = None,
    atol: float = 1e-08,
    verbose: bool = False
) -> list[Circuit]
```

Compress gates together up to the specified number of qubits. compress() is deterministic, so it can be reused elsewhere.

Parameters

`circuit` **: iter[BaseGate]** Circuit to compress.
`max_n_qubits` **: int, optional** Maximum number of qubits that a compressed gate may have.
`exclude_qubits` **: list[any], optional** Exclude gates which act on exclude_qubits to be compressed.
`use_matrix_commutation` **: bool, optional** If True, use commutation to maximize compression.
`max_n_qubits_matrix` **: int, optional** Limit the size of unitaries when checking for commutation.
`skip_compression` **: `iter[{type, str}]`, optional** If BaseGate is either an instance of any types in skip_compression, it provides any methods in skip_compression, or BaseGate name will match any names in skip_compression, BaseGate will not be compressed. However, if use_matrix_commutation is True, commutation will be checked against BaseGate.
`skip_commutation` **: `iter[{type, str}]`, optional** If BaseGate is either an instance of any types in skip_commutation, it provides any methods in skip_commutation, or BaseGate name will match any names in skip_commutation, BaseGate will not be checked against commutation.
`atol` **: float** Absolute tollerance for commutation.
`verbose` **: bool, optional** Verbose output.

Returns

list[Circuit] A list of Circuits, with each Circuit representing a compressed BaseGate.

See Also

hybridq.gate.commutes_with

Example

```
>>> # Define circuit
>>> circuit = Circuit(
>>>     [Gate('X', qubits=[0])**1.2,
>>>      Gate('ISWAP', qubits=[0, 1])**2.3,
>>>      Gate('ISWAP', qubits=[0, 2])**2.3])
>>>
>>> # Compress circuit up to 1-qubit gates
>>> utils.compress(circuit, 1)
[Circuit([
    Gate(name=X, qubits=[0])**1.2
 ]),
 Circuit([
    Gate(name=ISWAP, qubits=[0, 1])**2.3
 ]),
 Circuit([
    Gate(name=ISWAP, qubits=[0, 2])**2.3
 ])]
>>> # Compress circuit up to 2-qubit gates
>>> utils.compress(circuit, 2)
[Circuit([
    Gate(name=X, qubits=[0])**1.2
    Gate(name=ISWAP, qubits=[0, 1])**2.3
 ]),
 Circuit([
    Gate(name=ISWAP, qubits=[0, 2])**2.3
 ])]
>>> # Compress circuit up to 3-qubit gates
>>> utils.compress(circuit, 3)
[Circuit([
    Gate(name=X, qubits=[0])**1.2
    Gate(name=ISWAP, qubits=[0, 1])**2.3
    Gate(name=ISWAP, qubits=[0, 2])**2.3
 ])]
```

Function `expand_iswap`

```
def expand_iswap(
    circuit: Circuit
) -> hybridq.circuit.circuit.Circuit
```

Expand ISWAP's by iteratively replacing with SWAP's, CZ's and Phases.

Function `filter`

```
def filter(
    circuit: iter,
    names: list[str] = <built-in function any>,
    qubits: list[any] = <built-in function any>,
    params: list[any] = <built-in function any>,
    n_qubits: int = <built-in function any>,
    n_params: int = <built-in function any>,
    virtual: bool = <built-in function any>,
    exact_match: bool = False,
    atol: float = 1e-08,
    **kwargs
) -> iter
```

Function `flatten`

```
def flatten(
    a: Circuit
) -> hybridq.circuit.circuit.Circuit
```

Return a flattened circuit.

Function `insert_from_left`

```
def insert_from_left(
    circuit: iter[BaseGate],
    gate: BaseGate,
    atol: float = 1e-08,
    *,
    use_matrix_commutation: bool = True,
    simplify: bool = True,
    pop: bool = False,
    pinned_qubits: list[any] = None,
    inplace: bool = False
) -> Circuit
```

Add a gate to circuit starting from the left, commuting with existing gates if necessary.

Function `isclose`

```
def isclose(
    a: Circuit,
    b: Circuit,
    use_matrix_commutation: bool = True,
    atol: float = 1e-08,
    verbose: bool = False
)
```

Check if a is close to b within the absolute tollerance atol.

Parameters

`circuit` : Circuit[BaseGate] Circuit to compare with.
`use_matrix_commutation` : bool Use commutation rules. See simplify().
`atol` : float, optional Absolute tollerance.

Returns

bool True if the two circuits are close within the absolute tollerance atol, and False otherwise.

See Also

simplify()

Example

```
>>> c = Circuit(Gate('H', [q]) for q in range(10))
>>> c.isclose(Circuit(g**1.1 for g in c))
False
>>> c.isclose(Circuit(g**1.1 for g in c), atol=1e-1)
True
```

Function `matrix`

```
def matrix(
    circuit: iter[BaseGate],
    order: iter[any] = None,
    complex_type: any = 'complex64',
    max_compress: int = 4,
    verbose: bool = False
) -> numpy.ndarray
```

Return matrix representing circuit.

Parameters

`circuit` : iter[BaseGate] Circuit to get the matrix from.
`order` : iter[any], optional If specified, a matrix is returned following the order given by order. Otherwise, circuit.all_qubits() is used.
`max_compress` : int, optional To reduce the computational cost, circuit is compressed prior to compute the matrix.
`complex_type` : any, optional Complex type to use to compute the matrix.
`verbose` : bool, optional Verbose output.

Returns

numpy.ndarray Unitary matrix of circuit.

Example

```
>>> # Define circuit
>>> circuit = Circuit([Gate('CX', [1, 0])])
>>> # Show qubits
[0, 1]
>>> circuit.all_qubits()
>>> # Get matrix without specifying any qubits order
>>> # (therefore using circuit.all_qubits() == [0, 1])
>>> utils.matrix()
array([[1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
       [0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j],
       [0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j],
       [0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j]], dtype=complex64)
>>> # Get matrix with a specific order of qubits
>>> utils.matrix(Circuit([Gate('CX', [1, 0])]), order=[1, 0])
array([[1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
       [0.+0.j, 1.+0.j, 0.+0.j, 0.+0.j],
       [0.+0.j, 0.+0.j, 0.+0.j, 1.+0.j],
       [0.+0.j, 0.+0.j, 1.+0.j, 0.+0.j]], dtype=complex64)
```

Function `moments`

```
def moments(
    circuit: iter[{BaseGate, Circuit}]
) -> list[list[{BaseGate, Circuit}]]
```

Split circuit in moments.

Function `pop`

```
def pop(
    circuit: list[BaseGate],
    direction: str,
    pinned_qubits: list[any],
    atol: float = 1e-08,
    use_matrix_commutation: bool = True,
    simplify: bool = True,
    verbose: bool = False
) -> Circuit
```

Remove gates outside the lightcone created by pinned_qubits.

Function `popleft`

```
def popleft(
    circuit: list[BaseGate],
    pinned_qubits: list[any],
    atol: float = 1e-08,
    use_matrix_commutation: bool = True,
    simplify: bool = True,
    verbose: bool = False
) -> Circuit
```

Remove gates outside the lightcone created by pinned_qubits (starting from the right).

Function `popright`

```
def popright(
    circuit: list[BaseGate],
    pinned_qubits: list[any],
    atol: float = 1e-08,
    use_matrix_commutation: bool = True,
    simplify: bool = True,
    verbose: bool = False
) -> Circuit
```

Remove gates outside the lightcone created by pinned_qubits.

Function `remove_swap`

```
def remove_swap(
    circuit: Circuit
) -> tuple[Circuit, dict[any, any]]
```

Iteratively remove SWAP's from circuit by actually swapping qubits. The output map will have the form new_qubit -> old_qubit.

Function `simplify`

```
def simplify(
    circuit: list[BaseGate],
    atol: float = 1e-08,
```

```
        use_matrix_commutation: bool = True,
        remove_id_gates: bool = True,
        verbose: bool = False
    ) -> Circuit
```

Compress together gates up to the specified number of qubits.

Function `to_matrix_gate`

```
    def to_matrix_gate(
        circuit: iter[BaseGate],
        complex_type: any = 'complex64',
        **kwargs
    ) -> BaseGate
```

Convert circuit to a matrix BaseGate.

Parameters

`circuit` : iter[BaseGate] Circuit to convert to BaseGate.
`complex_type` : any, optional Float type to use while converting to BaseGate.

Returns

Gate BaseGate representing circuit.

Example

```
>>> # Define circuit
>>> circuit = Circuit(
>>>     [Gate('X', qubits=[0])**1.2,
>>>      Gate('ISWAP', qubits=[0, 1])**2.3])
>>>
>>> gate = utils.to_matrix_gate(circuit)
>>> gate
Gate(name=MATRIX, qubits=[0, 1], U=np.array(shape=(4, 4), dtype=complex64))
>>> gate.U
array([[ 0.09549151-0.29389262j,  0.        +0.j        ,
         0.9045085 +0.29389262j,  0.        +0.j        ],
       [ 0.13342446-0.41063824j, -0.08508356+0.26186025j,
        -0.13342446-0.04335224j, -0.8059229 -0.26186025j],
       [-0.8059229 -0.26186025j, -0.13342446-0.04335224j,
        -0.08508356+0.26186025j,  0.13342446-0.41063824j],
       [ 0.        +0.j        ,  0.9045085 +0.29389262j,
         0.        +0.j        ,  0.09549151-0.29389262j]],
      dtype=complex64)
```

Function `to_nx`

```
    def to_nx(
        circuit: iter[BaseGate],
        add_final_nodes: bool = True,
        node_tags: dict = None,
        edge_tags: dict = None,
        return_qubits_map: bool = False,
        leaves_prefix: str = 'q'
    ) -> networkx.Graph
```

Return graph representation of circuit. to_nx() is deterministic, so it can be reused elsewhere.

Parameters

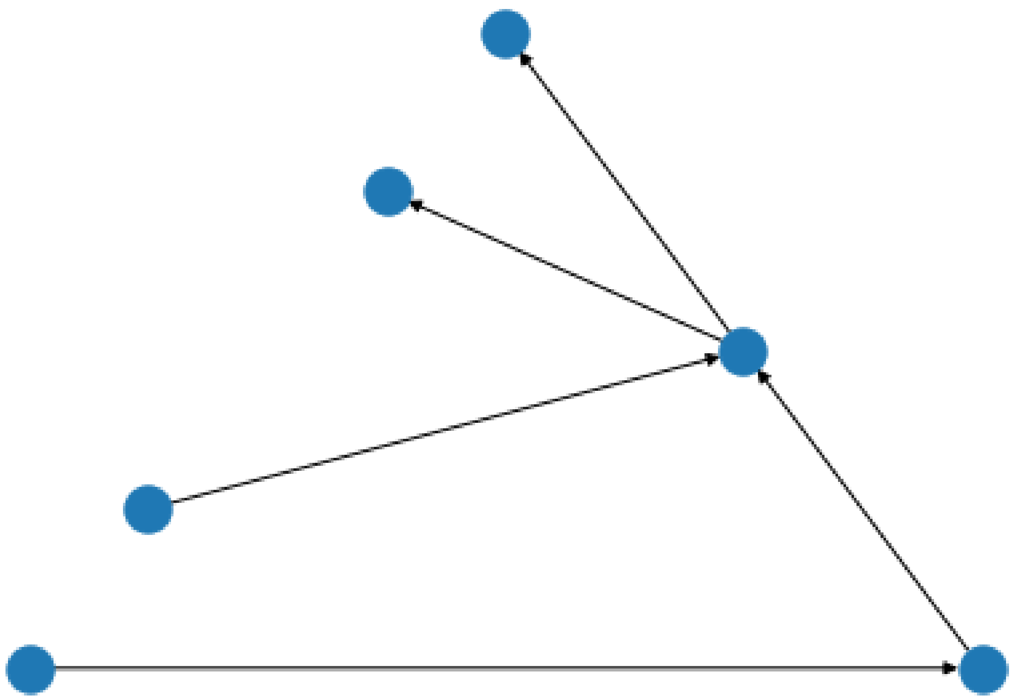
`circuit` : **iter[BaseGate]** Circuit to get graph representation from.
`add_final_nodes` : **bool, optional** Add final nodes for each qubit to the graph representation of circuit.
`node_tags` : **dict, optional** Add specific tags to nodes.
`edge_tags` : **dict, optional** Add specific tags to edges.
`return_qubits_map` : **bool, optional** Return map associated to the Circuit qubits.
`leaves_prefix` : **str, optional** Specify prefix to use for leaves.

Returns

networkx.Graph Graph representing circuit.

Example

```
>>> import networkx as nx
>>>
>>> # Define circuit
>>> circuit = Circuit(
>>>     [Gate('X', qubits=[0])**1.2,
>>>      Gate('ISWAP', qubits=[0, 1])**2.3], Gate('H', [1]))
>>>
>>> # Draw graph
>>> nx.draw_planar(utils.to_nx(circuit))
```



Function `to_tn`

```
def to_tn(
```

```
    circuit: iter[BaseGate],
    complex_type: any = 'complex64',
    return_qubits_map: bool = False,
    leaves_prefix: str = 'q_'
) -> quimb.tensor.TensorNetwork
```

Return quimb.tensor.TensorNetwork representing circuit. to_tn() is deterministic, so it can be reused elsewhere.

Parameters

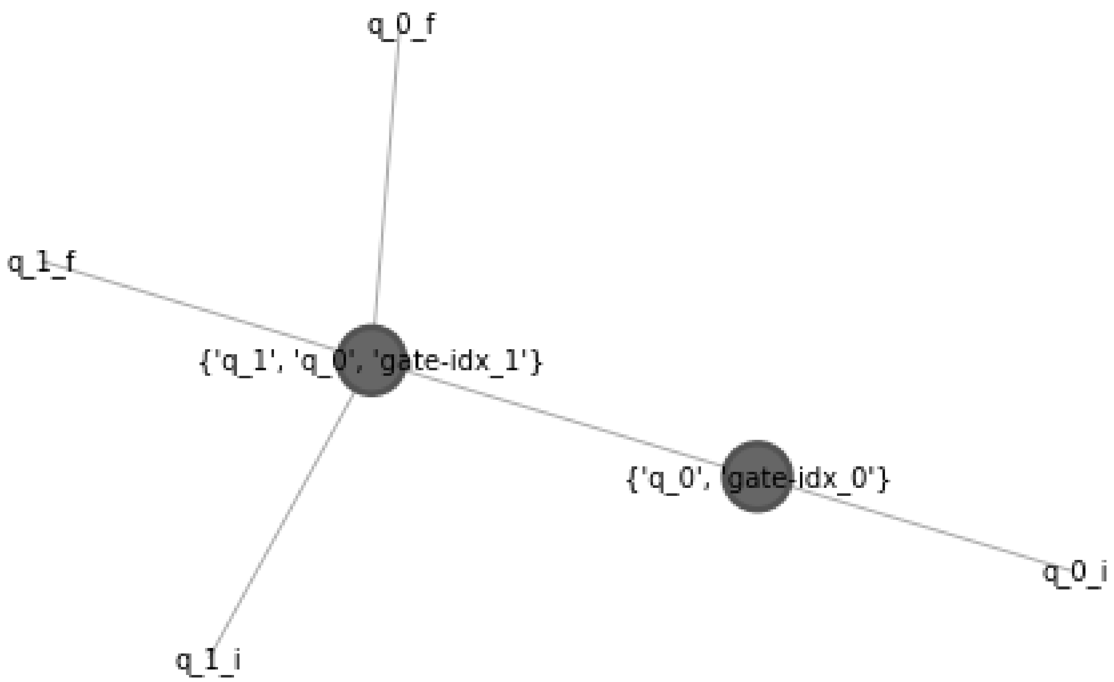
circuit : iter[BaseGate] Circuit to get quimb.tensor.TensorNetwork representation from.
complex_type : any, optional Complex type to use while getting the quimb.tensor.TensorNetwork representation.
return_qubits_map : bool, optional Return map associated to the Circuit qubits.
leaves_prefix : str, optional Specify prefix to use for leaves.

Returns

quimb.tensor.TensorNetwork Tensor representing circuit.

Example

```
>>> # Define circuit
>>> circuit = Circuit(
>>>     [Gate('X', qubits=[0])**1.2,
>>>      Gate('ISWAP', qubits=[0, 1])**2.3], Gate('H', [1]))
>>>
>>> # Draw graph
>>> utils.to_tn(circuit).graph()
```



Function `unitary`

```
def unitary(
    *args,
    **kwargs
)
```

Alias for utils.matrix.

Module `hybridq.dm`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Sub-modules

- hybridq.dm.circuit
- hybridq.dm.gate

Module `hybridq.dm.circuit`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Sub-modules

- hybridq.dm.circuit.circuit
- hybridq.dm.circuit.simulation

Module `hybridq.dm.circuit.circuit`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Classes

Class `Circuit`

```
class Circuit(
    gates: iter[Gate] = (),
    *args,
    **kwargs
)
```

Class representing a circuit.

Attributes

`gates` **: iter[Gate], optional** Gates to be added to Circuit.
`copy` **: bool, optional** If True, every gate is copied using deepcopy.

Example

```
>>> c = Circuit(Gate('H', qubits=[q]) for q in range(10))
>>> c
Circuit([
        Gate(name=H, qubits=[0])
        Gate(name=H, qubits=[1])
        Gate(name=H, qubits=[2])
        Gate(name=H, qubits=[3])
        Gate(name=H, qubits=[4])
        Gate(name=H, qubits=[5])
        Gate(name=H, qubits=[6])
        Gate(name=H, qubits=[7])
        Gate(name=H, qubits=[8])
        Gate(name=H, qubits=[9])
])
>>> c + [Gate('X')]
Circuit([
        Gate(name=H, qubits=[0])
        Gate(name=H, qubits=[1])
        Gate(name=H, qubits=[2])
        Gate(name=H, qubits=[3])
        Gate(name=H, qubits=[4])
        Gate(name=H, qubits=[5])
        Gate(name=H, qubits=[6])
        Gate(name=H, qubits=[7])
        Gate(name=H, qubits=[8])
        Gate(name=H, qubits=[9])
])
>>> c[5:2:-1]
Circuit([
```

```
        Gate(name=H, qubits=[5])
        Gate(name=H, qubits=[4])
        Gate(name=H, qubits=[3])
])
```

Ancestors (in MRO)

- hybridq.circuit.circuit.BaseCircuit
- builtins.list

Methods

Method `all_qubits`

```
def all_qubits(
    self,
    *,
    ignore_missing_qubits: bool = False
) -> tuple[list[any], list[any]]
```

Module `hybridq.dm.circuit.simulation`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function `simulate`

```
def simulate(
    circuit: SuperCircuit,
    initial_state: any,
    final_state: any = None,
    optimize: any = 'evolution',
    **kwargs
)
```

Frontend to simulate rho using different optimization models and backends.

Parameters

`circuit` : `Circuit` Circuit to simulate.
`initial_state` : `{str, Circuit, array_like}` Initial density matrix to evolve.
`final_state` : `{str, Circuit, array_like}` Final density matrix to project to.

optimize : any Method to use to perform the simulation. The available methods are: - evolution: Evolve the density matrix using state vector evolution - tn: Evolve the density matrix using tensor contraction - clifford: Evolve the density matrix using Clifford expansion

See Also

hybridq.circuit.simulation and hybridq.circuit.simulation.clifford

Module `hybridq.dm.gate`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Sub-modules

- hybridq.dm.gate.gate
- hybridq.dm.gate.property
- hybridq.dm.gate.utils

Module `hybridq.dm.gate.gate`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function `Gate`

```
def Gate(
    name: str,
    **kwargs
)
```

Function `KrausSuperGate`

```
def KrausSuperGate(
    gates: {iter[Gate], tuple[iter[Gate], iter[Gate]]},
    s: any = 1,
    tags: dict[any, any] = None,
    copy: bool = True
) -> KrausSuperGate
```

Return a KrausSuperGate.

Parameters

`gates` : `{iter[Gate], tuple[iter[Gate], iter[Gate]]}` List of valid Gate()s representing the KrausSuperGate(). If gates is a pair of list of Gate()s, the first list is used for the left-hand side Gate()s of the KrausSuperGate(), while the second list is used for the right-hand side Gate()s of KrausSuperGate(). If gates is a single list of Gate()'s, left/right-hand side Gates of the KrausSuperGate() are assumed to be the same.

`s` : np.ndarray Correlation matrix between left/right-hand side Gate()s of the KrausSuperOperator. More precisely, KrausSuperOperator will act on a dentity matrix ρ as:

```
K( ) = \sum_{ij} s_ij L_i   R_j^+
```

s can be a single scalar, a vector or a matrix consistent with the number of gates.

`tags` : dict[any, any], optional Dictionary of tags.

`copy` : bool, optional A copy of s is used instead of a reference if copy is True (default: True).

Returns

KrausSuperGate()

Function `MatrixSuperGate`

```
def MatrixSuperGate(
    Map: np.ndarray,
    l_qubits: iter[any],
    r_qubits: iter[any] = None,
    tags: dict[any, any] = None,
    copy: bool = True
) -> MatrixSuperGate
```

Return a MatrixSuperGate.

Parameters

`Map` : np.ndarray Map representing the SuperGate.
`l_qubits, r_qubits` : iter[any], iter[any] Left (right respectively) qubits for the Map. If r_qubits is not provided, r_qubits is assumed to be equal to l_qubits.
`tags` : dict[any, any], optional Dictionary of tags.
`copy` : bool, optional A copy of Map is used instead of a reference if copy is True (default: True).

Returns

MatrixSuperGate()

Classes

Class BaseSuperGate

```
class BaseSuperGate
```

Common type for all gates.

Ancestors (in MRO)

- hybridq.base.base.__Base__

Descendants

- hybridq.dm.gate.gate._MatrixSuperGate

Module hybridq.dm.gate.property

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Classes

Class Map

```
class Map
```

Basic features.

Ancestors (in MRO)

- hybridq.base.base.__Base__

Methods

Method commutes_with

```
def commutes_with(
    self,
    gate: Map,
    atol: float = 1e-07
)
```

Method `isclose`

```
def isclose(
    self,
    gate: Map,
    atol: float = 1e-08
)
```

Method `map`

```
def map(
    self,
    order: iter[any] = None
)
```

Return map.

Module `hybridq.dm.gate.utils`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function `to_matrix_supergate`

```
def to_matrix_supergate(
    gate: SuperGate,
    copy: bool = True
)
```

Convert gate to MatrixSuperGate.

Parameters

`gate` **: SuperGate** SuperGate to convert.m
`copy` **: bool, optional** A copy of Map is used instead of a reference if copy is True (default: True).

Returns

MatrixSuperGate

Module `hybridq.extras`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Sub-modules

- hybridq.extras.architecture
- hybridq.extras.gate
- hybridq.extras.io
- hybridq.extras.random
- hybridq.extras.simulation

Module `hybridq.extras.architecture`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Sub-modules

- hybridq.extras.architecture.plot
- hybridq.extras.architecture.sycamore

Module `hybridq.extras.architecture.plot`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Types

Qubit: tuple[int, int]

QpuLayout: list[Qubit]

Functions

Function `plot_qubits`

```
def plot_qubits(
    qpu_layout: QpuLayout,
    layout: list[Coupling] = None,
    subset: list[Qubit] = None,
    selection: list[Qubit] = None,
    figsize: tuple[int, int] = (6, 6),
    draw_border: bool = True,
    title: str = None
) -> None
```

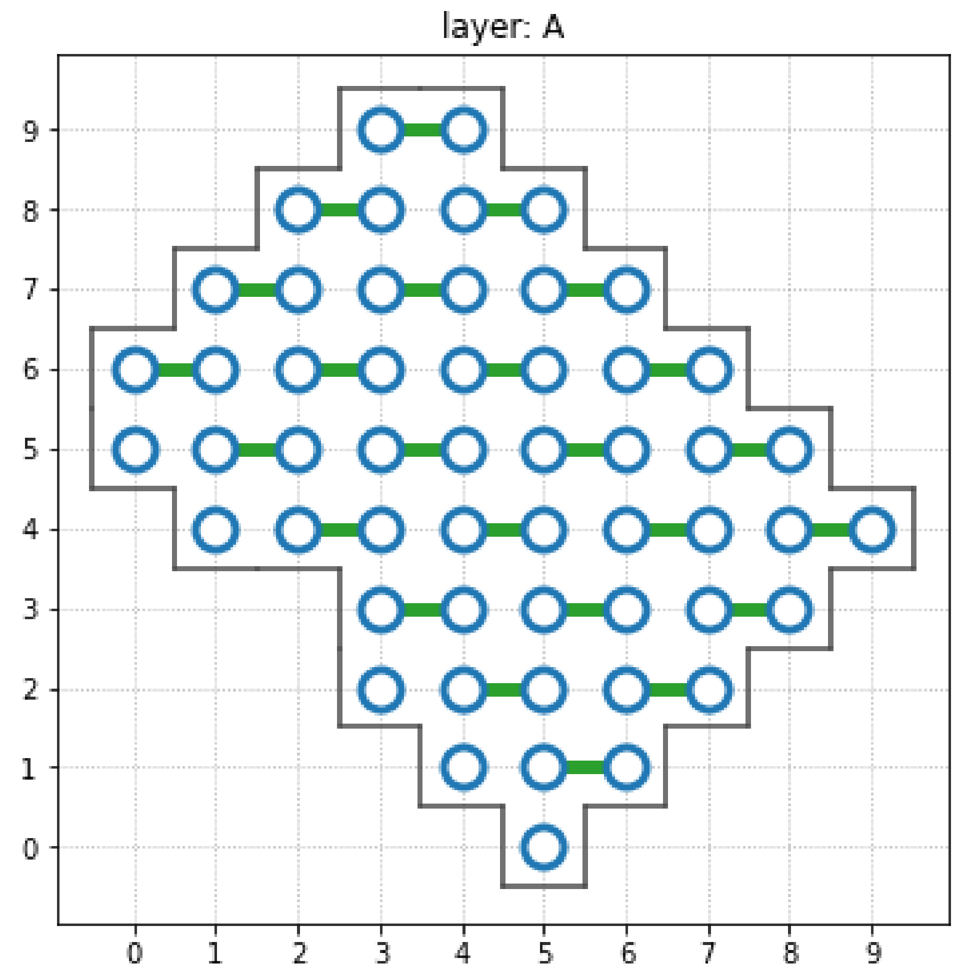
Plot qubits for 2D architectures.

Parameters

`qpu_layout` **: QpuLayout** List of qubits, with each qubits represented by the 2D coordinate, (x, y).
`layout` **: list[Coupling], optional** List of couplings between qubits.
`subset` **: list[Qubit], optional** Qubits in subset are highlated.
`selection` **: list[Qubit], optional** Box are added to qubits in selection.
`figsize` **: tuple[int, int], optional** Size of figure.
`draw_border` **: bool, optional** Draw border around qubits in qpu_layout.
`title` **: str, optional** Add title to plot.

Example

```
>>> from hybridq.architecture import sycamore
>>> qpu_layout = sycamore.gmon54
>>> layer = sycamore.get_layers(qpu_layout=qpu_layout)['A']
>>> plot_qubits(qpu_layout, layout=layer)
```



Module `hybridq.extras.architecture.sycamore`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Types

Qubit: tuple[int, int]

QpuLayout: list[Qubit]

Coupling: tuple[Qubit, Qubit]

Attributes

`gmon54` **: QpuLayout** Qubits available in Google Sycamore QPU.

Functions

Function `get_all_couplings`

```
def get_all_couplings(
    qpu_layout: QpuLayout
) -> list[Coupling]
```

Given qpu_layout of Qubits, return all couplings between nearest neighbors.

Parameters

`qpu_layout` : QpuLayout List of Qubits to use as QpuLayout.

Returns

list[Coupling] List of all possible couplings between nearest neighbor Qubits.

Example

```
>>> get_all_couplings(qpu_layout=((0, 0), (0, 1), (1, 0), (1, 1)))
[((0, 0), (0, 1)), ((0, 0), (1, 0)), ((0, 1), (1, 1)), ((1, 0), (1, 1))]
```

Function `get_layers`

```
def get_layers(
    qpu_layout: list[Qubit]
) -> dict[str, list[Coupling]]
```

Return layers used in Google Quantum Supremacy Paper [Nature 574 (7779), 505-510].

Parameters

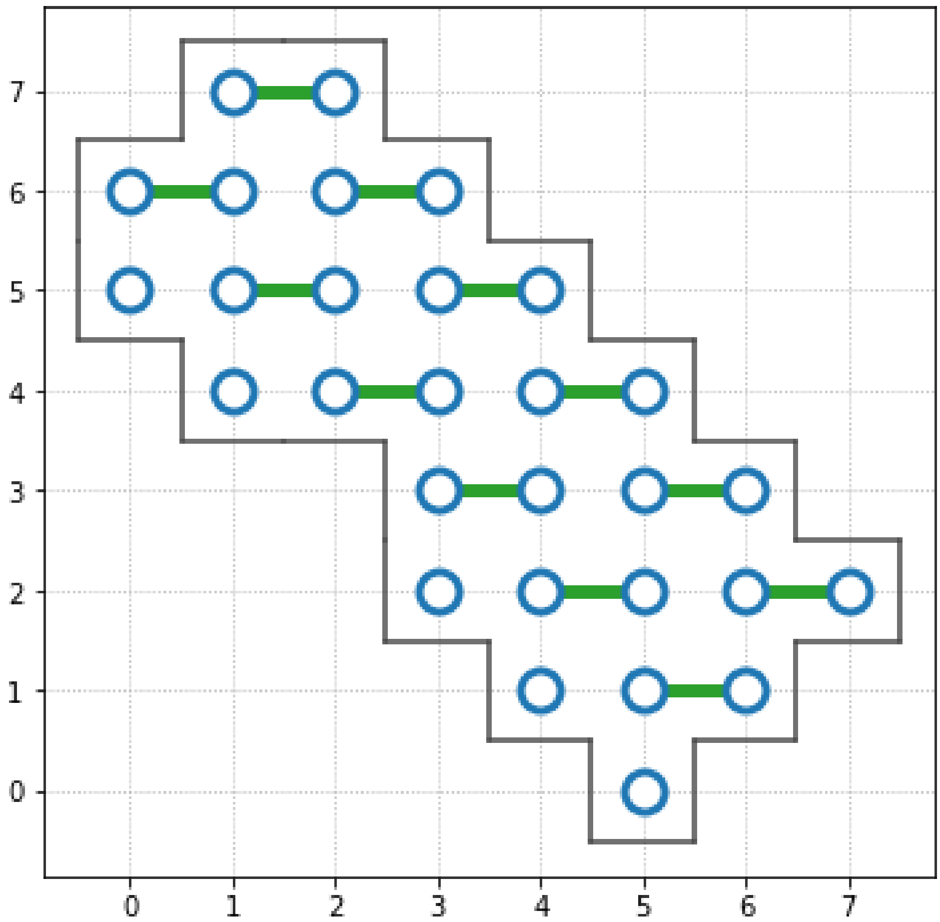
`qpu_layout` : QpuLayout List of Qubits to use as QpuLayout.

Returns

dict[str, list[Coupling]] Map between layer name and the list of corresponding Couplings.

Example

```
>>> from hybridq.architecture.plot import plot_qubits
>>> qpu_layout = [(x, y) for x, y in gmon54 if x + y < 10]
>>> layers = get_layers(qpu_layout=qpu_layout)
>>> layers.keys()
dict_keys(['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'])
>>> layers['A']
[((0, 6), (1, 6)),
 ((1, 5), (2, 5)),
 ((1, 7), (2, 7)),
 ((2, 4), (3, 4)),
 ((2, 6), (3, 6)),
 ((3, 3), (4, 3)),
 ((3, 5), (4, 5)),
 ((4, 2), (5, 2)),
 ((4, 4), (5, 4)),
 ((5, 1), (6, 1)),
 ((5, 3), (6, 3)),
 ((6, 2), (7, 2))]
>>> plot_qubits(qpu_layout=qpu_layout, layout=layers['A'])
```



Function `index_to_xy`

```
def index_to_xy(
    qpu_layout: QpuLayout
) -> dict[int, Qubit]
```

Given qpu_layout of Qubits, return a one-to-one between indexes and Qubits.

Parameters

`qpu_layout` **:** **QpuLayout** List of Qubits to use as QpuLayout.

Returns

dict[int, Qubit] One-to-one map between indexes and Qubits.

Note

`xy_to_index` and `index_to_xy` are by construction one the inverse of the other.

Example

```
>>> sum(q != sycamore.index_to_xy(qpu_layout=sycamore.gmon54)[x]
>>>     for q, x in sycamore.xy_to_index(qpu_layout=sycamore.gmon54).items())
0
```

Function `xy_to_index`

```
def xy_to_index(
    qpu_layout: QpuLayout
) -> dict[Qubit, int]
```

Given qpu_layout of Qubits, return a one-to-one between Qubits and indexes.

Parameters

`qpu_layout` : QpuLayout List of Qubits to use as QpuLayout.

Returns

dict[Qubit, int] One-to-one map between Qubits and indexes.

Note

`xy_to_index` and `index_to_xy` are by construction one the inverse of the other.

Example

```
>>> sum(q != sycamore.index_to_xy(qpu_layout=sycamore.gmon54)[x]
>>>     for q, x in sycamore.xy_to_index(qpu_layout=sycamore.gmon54).items())
0
```

Module `hybridq.extras.gate`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Sub-modules

- hybridq.extras.gate.gate

Module `hybridq.extras.gate.gate`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function `Gate`

```
def Gate(
    name: str,
    message: str,
    qubits: iter[any] = None,
    file: any = sys.stdout
)
```

Classes

Class `MessageGate`

```
class MessageGate(
    qubits: iter[any] = None,
    **kwargs
)
```

FunctionalGate to manipulate state.

Ancestors (in MRO)

- hybridq.gate.property.FunctionalGate
- hybridq.gate.property.QubitGate
- hybridq.base.property.Tags
- hybridq.base.property.Name
- hybridq.base.base.__Base__

Methods

Method `apply`

```
def apply(
    self,
    psi,
    order,
    *args,
    **kwargs
)
```

Module `hybridq.extras.io`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Sub-modules

- hybridq.extras.io.cirq
- hybridq.extras.io.qasm

Module `hybridq.extras.io.cirq`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function `to_cirq`

```
def to_cirq(
    circuit: Circuit,
    qubits_map: dict[any, any] = None,
    verbose: bool = False
) -> cirq.Circuit
```

Convert Circuit to cirq.Circuit.

Parameters

`circuit` **: Circuit** Circuit to convert to cirq.Circuit.
`qubits_map` **: dict[any, any], optional** How to map qubits in Circuit to cirq.Circuit. if not provided, cirq.LineQubits are used and automatically mapped to Circuit's qubits.
`verbose` **: bool, optional** Verbose output.

Returns

cirq.Circuit cirq.Circuit obtained from Circuit.

Example

```
>>> from hybridq.extras.cirq import to_cirq
>>> c = Circuit(Gate('H', qubits=[q]) for q in range(3))
>>> c.append(Gate('CX', qubits=[0, 1]))
>>> c.append(Gate('CX', qubits=[2, 0]))
>>> c.append(Gate('CX', qubits=[1, 2]))
>>> to_cirq(c)
0:   H  @  X

1:   H  X     @

2:   H     @  X
```

Module `hybridq.extras.io.qasm`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function `from_qasm`

```
def from_qasm(
    qasm_string: str
) -> hybridq.circuit.circuit.Circuit
```

Convert a QASM circuit to Circuit.

Parameters

`qasm_string` : str QASM circuit to convert to Circuit.

Returns

Circuit QAMS circuit converted to Circuit.

Notes

The QASM language used in HybridQ is compatible with the standard QASM. However, HybridQ introduces few extensions, which are recognized by the parser using `#@` at the beginning of the line (# at the beginning of the line represent a general comment in QASM). At the moment, the following QAMS extensions are supported:

- **qubits**, used to store qubits_map,
- **power**, used to store the power of the gate,
- **tags**, used to store the tags associated to the gate,
- **U**, used to store the matrix reprentation of the gate if gate name is MATRIX

If Gate.qubits are not specified, a single `.` is used to represent the missing qubits. If Gate.params are missing, parameters are just omitted.

Example

```
>>> from hybridq.extras.qasm import from_qasm
>>> qasm_str = """
>>> 1
>>> #@ qubits =
>>> #@ {
>>> #@   "0": "42"
>>> #@ }
>>> #@ tags =
>>> #@ {
```

```
>>> #@   "params": false,
>>> #@   "qubits": false
>>> #@ }
>>> rx .
>>> #@ tags =
>>> #@ {
>>> #@   "params": true,
>>> #@   "qubits": false
>>> #@ }
>>> ry . 1.23
>>> #@ tags =
>>> #@ {
>>> #@   "params": false,
>>> #@   "qubits": true
>>> #@ }
>>> #@ power = 1.23
>>> rz 0
>>> #@ U =
>>> #@ [
>>> #@   [
>>> #@     "0.7071067811865475",
>>> #@     "0.7071067811865475"
>>> #@   ],
>>> #@   [
>>> #@     "0.7071067811865475",
>>> #@     "-0.7071067811865475"
>>> #@   ]
>>> #@ ]
>>> matrix .
>>> """
>>> from_qasm(qasm_str)
Circuit([
        Gate(name=RX, tags={'params': False, 'qubits': False})
        Gate(name=RY, params=[1.23], tags={'params': True, 'qubits': False})
        Gate(name=RZ, qubits=[42], tags={'params': False, 'qubits': True})**1.23
        Gate(name=MATRIX, U=np.array(shape=(2, 2), dtype=float64))
])
```

Function `to_qasm`

```
def to_qasm(
    circuit: Circuit,
    qubits_map: dict[any, int] = None
) -> str
```

Convert a Circuit to QASM language.

Parameters

`circuit` **: Circuit** Circuit to convert to QASM language.
`qubits_map` **: dict[any, int], optional** If provided, qubits_map map qubit indexes in Circuit to qubit indexes in QASM. Otherwise, indexes are assigned to QASM qubits by using Circuit.all_qubits() order.

Returns

str String representing the QAMS circuit.

Notes

The QASM language used in HybridQ is compatible with the standard QASM. However, HybridQ introduces few extensions, which are recognized by the parser using #@ at the beginning of the line (# at the beginning of the line represent a general comment in QASM). At the moment, the following QAMS extensions are supported:

- **qubits**, used to store qubits_map,
- **power**, used to store the power of the gate,
- **tags**, used to store the tags associated to the gate,
- **U**, used to store the matrix reprentation of the gate if gate name is MATRIX

If Gate.qubits are not specified, a single . is used to represent the missing qubits. If Gate.params are missing, parameters are just omitted.

Example

```
>>> from hybridq.extras.qasm import to_qasm
>>> print(to_qasm(Circuit(Gate('H', [q]) for q in range(10))))
10
#@ qubits =
#@ {
#@   "0": "0",
#@   "1": "1",
#@   "2": "2",
#@   "3": "3",
#@   "4": "4",
#@   "5": "5",
#@   "6": "6",
#@   "7": "7",
#@   "8": "8",
#@   "9": "9"
#@ }
h 0
h 1
h 2
h 3
h 4
h 5
h 6
h 7
h 8
h 9
>>> c = Circuit()
>>> c.append(Gate('RX', tags={'params': False, 'qubits': False}))
>>> c.append(Gate('RY', params=[1.23], tags={'params': True, 'qubits': False}))
>>> c.append(Gate('RZ', qubits=[42], tags={'params': False, 'qubits': True})**1.23)
>>> c.append(Gate('MATRIX', U=Gate('H').matrix()))
>>> print(to_qasm(c))
1
#@ qubits =
#@ {
#@   "0": "42"
#@ }
#@ tags =
#@ {
#@   "params": false,
#@   "qubits": false
#@ }
rx .
```

```
#@ tags =
#@ {
#@   "params": true,
#@   "qubits": false
#@ }
ry . 1.23
#@ tags =
#@ {
#@   "params": false,
#@   "qubits": true
#@ }
#@ power = 1.23
rz 0
#@ U =
#@ [
#@   [
#@     "0.7071067811865475",
#@     "0.7071067811865475"
#@   ],
#@   [
#@     "0.7071067811865475",
#@     "-0.7071067811865475"
#@   ]
#@ ]
matrix .
```

Module `hybridq.extras.random`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function `get_random_gate`

```
def get_random_gate(
    randomize_power: bool = True,
    use_clifford_only: bool = False,
    use_unitary_only: bool = True
)
```

Generate random gate.

Function `get_random_indexes`

```
def get_random_indexes(
```

```
        n_qubits: int,
        *,
        use_random_indexes: bool = False
    )
```

Function `get_rqc`

```
    def get_rqc(
        n_qubits: int,
        n_gates: int,
        *,
        indexes: list[int] = None,
        randomize_power: bool = True,
        use_clifford_only: bool = False,
        use_unitary_only: bool = True,
        use_random_indexes: bool = False
    )
```

Generate random quantum circuit.

Module `hybridq.extras.simulation`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Sub-modules

- hybridq.extras.simulation.otoc

Module `hybridq.extras.simulation.otoc`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function `generate_OTOC`

```
def generate_OTOC(
    layout: dict[any, list[Coupling]],
    depth: int,
    sequence: list[any],
    one_qb_gates: iter[Gate],
    two_qb_gates: iter[Gate],
    butterfly_op: str,
    ancilla: Qubit,
    targets: list[Qubit],
    qubits_order: list[Qubit] = None
) -> Circuit
```

Function `generate_U`

```
def generate_U(
    layout: dict[any, list[Coupling]],
    qubits_order: list[Qubit],
    depth: int,
    sequence: list[any],
    one_qb_gates: iter[Gate],
    two_qb_gates: iter[Gate],
    exclude_qubits: iter[Qubit] = None
) -> Circuit
```

Generate U at a given depth.

Module `hybridq.gate`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Sub-modules

- hybridq.gate.gate
- hybridq.gate.measure
- hybridq.gate.projection
- hybridq.gate.property
- hybridq.gate.utils

Module hybridq.gate.gate

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function Control

```
def Control(
    c_qubits: iter[any],
    gate: BaseGate,
    power: any = 1,
    tags: dict[any, any] = None,
    copy: bool = True
)
```

Generate a controlled Gate().

Parameters

`c_qubits` **: iter[any]** List of controlling qubits.
`gate` **: BaseGate** Gate to control.
`power` **: float, optional** The power the matrix of Gate() is elevated to.
`tags` **: dict[any, any], optional** Dictionary of tags.
`copy` **: bool, optional** A copy of gate is used instead of a reference if copy is True (default: True).

Returns

MatrixGate()

Function FunctionalGate

```
def FunctionalGate(
    f: callable,
    qubits: iter[any] = None,
    n_qubits: int = None,
    tags: dict[any, any] = None
) -> FunctionalGate
```

Generator of gates.

Parameters

`f` **: `callable[self, psi={np.ndarray, tuple[np.ndarray, np.ndarray]}, order=list[any]]`, optional**
Function used to manipulate the quantum state. f must be a callable function which accepts three parameters: self, the gate being called, psi, the quantum state, and order, the order of qubits in the quantum state. psi can either be a single array of complex numbers, or a tuple of two real-valued array representing the real and imaginary part of psi respectively. order is the ordered list of qubits in psi. Finally, f must change psi in place and return the new order.

qubits : iter[any], optional List of qubits Gate() is acting on.
n_qubits : int, optional Specify the number of qubits if qubits is unspecified.
tags : dict[any, any], optional Dictionary of tags.

Returns

FunctionalGate()

Function Gate

```
def Gate(
    name: str,
    qubits: iter[any] = None,
    params: iter[any] = None,
    n_qubits: int = None,
    power: any = 1,
    tags: dict[any, any] = None,
    **kwargs
) -> Gate
```

Generator of gates.

Parameters

name : str Name of Gate().
qubits : iter[any], optional List of qubits Gate() is acting on.
params : iter[any], optional List of parameters to define Gate().
n_qubits : int, optional Specify the number of qubits if qubits is unspecified.
power : float, optional The power the matrix of Gate() is elevated to.
tags : dict[any, any], optional Dictionary of tags.

See Also

NamedGate(), MatrixGate(), TupleGate(), StochasticGate(), FunctionalGate(), Projection, Measure

Returns

Gate()

Function MatrixGate

```
def MatrixGate(
    U: np.ndarray,
    qubits: iter[any] = None,
    n_qubits: int = None,
    power: any = 1,
    tags: dict[any, any] = None,
    copy: bool = True,
    check_if_unitary: bool = True,
    atol: float = 1e-08
) -> MatrixGate
```

Generate matrix gates.

Parameters

U : list[list[any]], optional The matrix representing the matrix gate.

qubits : iter[any], optional List of qubits Gate() is acting on.
n_qubits : int, optional Specify the number of qubits if qubits is unspecified.
power : float, optional The power the matrix of Gate() is elevated to.
tags : dict[any, any], optional Dictionary of tags.
copy : bool, optional A copy of U is used instead of a reference if copy is True (default: True).
check_if_unitary : bool, optional Check if U is unitary and use UnitaryGate instead of PowerMatrixGate accordingly.
atol : float, optional Use atol as absolute precision for checks.

Returns

MatrixGate()

Function NamedGate

```
def NamedGate(
    name: str,
    qubits: iter[any] = None,
    params: iter[any] = None,
    n_qubits: int = None,
    power: any = 1,
    tags: dict[any, any] = None
) -> NamedGate
```

Generate named gates.

Parameters

name : str Name of Gate().
qubits : iter[any], optional List of qubits Gate() is acting on.
params : iter[any], optional List of parameters to define Gate().
n_qubits : int, optional Specify the number of qubits if qubits is unspecified.
power : float, optional The power the matrix of Gate() is elevated to.
tags : dict[any, any], optional Dictionary of tags.

Returns

NamedGate()

Function SchmidtGate

```
def SchmidtGate(
    gates: {iter[Gate], tuple[iter[Gate], iter[Gate]]},
    s=None,
    tags: dict[any, any] = None,
    copy: bool = True
) -> SchmidtGate
```

Return a SchmidtGate.

Parameters

gates : tuple[iter[Gate()], iter[Gate()]] Pair of lists of Gate()s. Gate()s must provide qubits and matrix and cannot have common qubits.

s : np.ndarray Correlation matrix between Gates. More precisely, SchmidtGate.Matrix is built as follows:

```
U = \sum_{ij} s_ij G_i G_j.
```

s can be a single scalar, a vector or a matrix consistent with the number of gates.

`tags` **: dict[any, any], optional** Dictionary of tags.

`copy` **: bool, optional** A copy of s is used instead of a reference if copy is True (default: True).

Returns

SchmidtGate()

Function `StochasticGate`

```
def StochasticGate(
    gates: iter[BaseGate],
    p: iter[float],
    tags: dict[any, any] = None
) -> StochasticGate
```

Generator of gates.

Parameters

`name` **: str** Name of Gate().
`gates` **: iter[BaseGate]** If name is STOCHASTIC, gates are used to initialize `Gate('STOCHASTIC')`.
`p` **: iter[float]** If name is STOCHASTIC, p will be used as probabilities to sample from `Gate('STOCHASTIC')`.

See Also

BaseTupleGate, TagGate, NameGate

Function `TupleGate`

```
def TupleGate(
    gates: iter[BaseGate] = None,
    tags: dict[any, any] = None
) -> TupleGate
```

Generate a tuple gate.

Parameters

`gates` **: iter[BaseGate]** gates used to initialize the TupleGate().
`tags` **: dict[any, any], optional** Dictionary of tags.

Returns

TupleGate()

Classes

Class `BaseGate`

```
class BaseGate
```

Common type for all gates.

Ancestors (in MRO)

- hybridq.base.base.__Base__

Descendants

- hybridq.gate.gate._StochasticGate
- hybridq.gate.projection.ProjectionGate

Module `hybridq.gate.measure`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function `Measure`

```
def Measure(
    qubits: iter[any] = None,
    n_qubits: int = None,
    tags: dict[any, any] = None
) -> pr.BaseGate
```

Generator of measurement gates.

Parameters

`state` : str, State to project to.
`qubits` : iter[any], optional List of qubits Projection is acting on.
`tags` : dict[any, any], optional Dictionary of tags.

See Also

MeasureGate, FunctionalGate

Module `hybridq.gate.projection`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function `Projection`

```
def Projection(
    state: iter[any],
    qubits: iter[any] = None,
    tags: dict[any, any] = None
) -> pr.BaseGate
```

Generator of projectors.

Parameters

`state` : str, State to project to.
`qubits` : iter[any], optional List of qubits Projection() is acting on.
`tags` : dict[any, any], optional Dictionary of tags.

See Also

ProjectionGate, FunctionalGate

Classes

Class `ProjectionGate`

```
class ProjectionGate
```

Common type for all gates.

Ancestors (in MRO)

- hybridq.gate.gate.BaseGate
- hybridq.base.base.__Base__

Module `hybridq.gate.property`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Classes

Class `BaseTupleGate`

```
class BaseTupleGate(
    elements=(),
    tags=None,
    **kwargs
)
```

Gate defined as a tuple of gates.

Ancestors (in MRO)

- hybridq.base.property.Tags
- hybridq.base.property.Tuple
- hybridq.base.base.__Base__

Instance variables

Variable `n_qubits` Type: `int`

Variable `qubits` Type: `tuple[any, ...]`

Class `CliffordGate`

```
class CliffordGate
```

Basic features.

Ancestors (in MRO)

- hybridq.base.base.__Base__

Class `ControlledGate`

```
class ControlledGate
```

Basic features.

Ancestors (in MRO)

- hybridq.base.base.__Base__

Instance variables

Variable `n_qubits`

Variable `qubits`

Class FunctionalGate

```
class FunctionalGate(
    qubits: iter[any] = None,
    **kwargs
)
```

FunctionalGate to manipulate state.

Ancestors (in MRO)

- hybridq.gate.property.QubitGate
- hybridq.base.base.__Base__

Descendants

- hybridq.extras.gate.gate.MessageGate

Class MatrixGate

```
class MatrixGate
```

Class for gates that can be represented as a matrix.

Ancestors (in MRO)

- hybridq.base.base.__Base__

Class ParamGate

```
class ParamGate(
    params: iter[any] = None,
    **kwargs
)
```

Class representing a gate with qubits.

Ancestors (in MRO)

- hybridq.base.property.Params
- hybridq.base.base.__Base__

Descendants

- hybridq.gate.property.RotationGate

Instance variables

Variable `Matrix` Type: `numpy.ndarray`

Class PowerGate

```
class PowerGate(
    power: any = 1,
    **kwargs
)
```

Class representing a gate that can be raised to a given power.

Attributes

`power` : any, optional

Ancestors (in MRO)

- hybridq.base.base.__Base__

Descendants

- hybridq.gate.property.PowerMatrixGate
- hybridq.gate.property.RotationGate

Instance variables

Variable `power` Type: `<built-in function any>`

Methods

Method `inv`

```
def inv(
    self,
    *,
    inplace: bool = False
) -> hybridq.gate.property.PowerGate
```

Return inverse of PowerGate. If inplace is True, PowerGate is modified in place.

Parameters

`inplace` : bool, optional If True, PowerGate is modified in place. Otherwise, a new PowerGate is returned.

Returns

PowerGate Inverse of PowerGate. If True, PowerGate is modified in place.

Example

```
>>> g = PowerGate(U=[[1, 0], [0, np.exp(-0.23j)]])
>>> g.matrix()
array([[1.        +0.j        , 0.        +0.j        ],
       [0.        +0.j        , 0.9736664-0.22797752j]])
>>> g.inv().matrix()
array([[1.        -0.j        , 0.        -0.j        ],
       [0.        -0.j        , 0.9736664+0.22797752j]])
>>> g.inv().matrix() @ g.matrix()
array([[1.+0.j, 0.+0.j],
       [0.+0.j, 1.+0.j]])
```

Method `set_power`

```
def set_power(
    self,
    power: any,
    *,
    inplace: bool = False
) -> hybridq.gate.property.PowerGate
```

Return PowerGate to the given power. If inplace is True, PowerGate is modified in place.

Parameters

`power` : any Power to elevate PowerGate.
`inplace` : bool, optional If True, PowerGate is modified in place. Otherwise, a new PowerGate is returned.

Returns

PowerGate New PowerGate to the given power. If inplace is True, PowerGate is modified in place.

Example

```
>>> PowerGate(U=[[1, 0], [0, -1]]).matrix()
array([[ 1,  0],
       [ 0, -1]])
>>> PowerGate(U=[[1, 0], [0, -1]]).set_power(1.2345).matrix()
array([[ 1.         +0.j         ,  0.         +0.j         ],
       [ 0.         +0.j         , -0.74068735-0.67184987j]])
```

Class `PowerMatrixGate`

```
class PowerMatrixGate(
    *args,
    **kwargs
)
```

Class representing a single matrix gate.

Ancestors (in MRO)

- hybridq.gate.property.PowerGate
- hybridq.base.base.__Base__

Descendants

- hybridq.gate.property.UnitaryGate

Methods

Method `T`

```
def T(
    self,
    *,
    inplace: bool = False
) -> hybridq.gate.property.PowerMatrixGate
```

Method `adj`

```
def adj(
    self,
    *,
    inplace: bool = False
) -> hybridq.gate.property.PowerMatrixGate
```

Method `commutes_with`

```
def commutes_with(
    self,
    gate: PowerMatrixGate,
    atol: float = 1e-07
) -> bool
```

Return True if the calling gate commutes with gate.

Parameters

`gate` : **PowerMatrixGate** Gate to check commutation with.
`atol` : **float** Absolute tollerance.

Returns

bool True if the calling gate commutes with gate, otherwise False.

Method `conj`

```
def conj(
    self,
    *,
    inplace: bool = False
) -> hybridq.gate.property.PowerMatrixGate
```

Method `is_conjugated`

```
def is_conjugated(
    self
) -> bool
```

Method `is_transposed`

```
def is_transposed(
    self
) -> bool
```

Method `isclose`

```
def isclose(
    self,
    gate: Gate,
    atol: float = 1e-08
) -> bool
```

Determine if the matrix of gate is close within an absolute tollerance. If the gates are acting on a different set of qubits, isclose will return False.

Parameters

`gate` **: PowerMatrixGate** Gate to compare with.
`atol` **: float, optional** Absolute tollerance.

Returns

bool True if the two gates are close withing the given absolute tollerance, otherwise False.

Example

```
>>> g1 = PowerMatrixGate(U = [[1, 2], [3, 4]])
>>> g2 = PowerMatrixGate(U = [[4, 5], [3, 4]])
>>> g1.isclose(g1)
True
>>> g1.isclose(g2)
False
>>> g1.on([3]).isclose(g1)
False
>>> g1.on([3]).isclose(g1.on([3])
True
```

Method `matrix`

```
def matrix(
    self,
    order: iter[any] = None
) -> np.ndarray
```

Return matrix representing MatrixPowerGate. If order is provided, the given order of qubits is used to output its matrix.

Parameters

`order` **: iter[any]** Order of qubits used to output the matrix.

Returns

array_like Matrix representing MatrixPowerGate.

Example

```
>>> g = PowerMatrixGate(qubits=[0, 1], U=[[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 1], [0, 0, 1, 0]])
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 0, 1],
       [0, 0, 1, 0]])
```

The order of qubits is [1, 0]. On the contrary:

```
>>> g.on().matrix(order=[1, 0])
array([[1, 0, 0, 0],
       [0, 0, 0, 1],
       [0, 0, 1, 0],
       [0, 1, 0, 0]])
```

outputs a matrix with the qubits order being if [0, 1].

Class `QubitGate`

```
class QubitGate(
    qubits: iter[any] = None,
    **kwargs
)
```

Class representing a gate with qubits.

Attributes

`qubits` **: iter[any], optional**

Ancestors (in MRO)

- hybridq.base.base.__Base__

Descendants

- hybridq.gate.property.FunctionalGate

Instance variables

Variable `qubits` Type: `tuple[any, ...]`

Methods

Method `on`

```
def on(
    self,
    qubits: iter[any] = None,
    *,
    inplace: bool = False
) -> QubitGate
```

Return QubitGate applied to qubits. If inplace is True, QubitGate is modified in place.

Parameters

`qubits` : iter[any] Qubits the new Gate will act on.
`inplace` : bool, optional If True, QubitGate is modified in place. Otherwise, a new QubitGate is returned.

Returns

QubitGate New QubitGate acting on qubits. If inplace is True, QubitGate is modified in place.

Example

```
>>> QubitGate([1, 2]).qubits
[1, 2]
>>> QubitGate().on([42]).qubits
[42]
```

Class `RotationGate`

```
class RotationGate(
    params: iter[any] = None,
    **kwargs
)
```

Gate with form U = exp(-1j * r / 2 * O), with O an arbitrary matrix.

Ancestors (in MRO)

- hybridq.gate.property.ParamGate
- hybridq.base.property.Params
- hybridq.gate.property.PowerGate
- hybridq.base.base.__Base__

Methods

Method `Matrix_gen`

```
def Matrix_gen(
    self,
    r
)
```

Class `SchmidtGate`

```
class SchmidtGate
```

Basic features.

Ancestors (in MRO)

- hybridq.base.base.__Base__

Instance variables

Variable `Matrix`

Construct Matrix representing the Map. Order of qubits for Matrix will be `SchmidtGate.gates[0].qubits + SchmidtGate.gates[1].qubits`.

Class `SelfAdjointUnitaryGate`

```
class SelfAdjointUnitaryGate(
    *args,
    **kwargs
)
```

Class representing a single matrix gate.

Ancestors (in MRO)

- hybridq.gate.property.UnitaryGate
- hybridq.gate.property.PowerMatrixGate
- hybridq.gate.property.PowerGate
- hybridq.base.base.__Base__

Methods

Method `T`

```
def T(
    self,
    *,
    inplace: bool = False
) -> hybridq.gate.property.SelfAdjointUnitaryGate
```

Apply transposition to self.matrix().

Method `adj`

```
def adj(
    self,
    *,
    inplace: bool = False
) -> hybridq.gate.property.SelfAdjointUnitaryGate
```

Apply adjunction to self.matrix().

Method conj

```
def conj(
    self,
    *,
    inplace: bool = False
) -> hybridq.gate.property.SelfAdjointUnitaryGate
```

Apply conjugation to self.matrix().

Class StochasticGate

```
class StochasticGate
```

Basic features.

Ancestors (in MRO)

- hybridq.base.base.__Base__

Descendants

- hybridq.gate.gate._StochasticGate

Class UnitaryGate

```
class UnitaryGate(
    *args,
    **kwargs
)
```

Class representing a single matrix gate.

Ancestors (in MRO)

- hybridq.gate.property.PowerMatrixGate
- hybridq.gate.property.PowerGate
- hybridq.base.base.__Base__

Descendants

- hybridq.gate.property.SelfAdjointUnitaryGate

Methods

Method unitary

```
def unitary(
    self,
    *args,
    **kwargs
) -> numpy.ndarray
```

Alias for self.matrix.

Module hybridq.gate.utils

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function `decompose`

```
def decompose(
    gate: Gate,
    qubits: iter[any],
    return_matrices: bool = False,
    atol: float = 1e-08
) -> SchmidtGate
```

Decompose gate using the Schmidt decomposition.

Parameters

`gate` **: Gate** Gate to decompose.
`qubits` **: iter[any]** Subset of qubits used to decompose gate.
`return_matrices` **: bool, optional** If True, return matrices instead of gates (default: False)
`atol` **: float** Tollerance.

Returns

`d` **: tuple(list[float], tuple[Gate, ...], tuple[Gate, ...])** Decomposition of gate.

See Also

hybridq.utils.svd

Function `get_available_gates`

```
def get_available_gates() -> tuple[str, ...]
```

Return available gates.

Function `get_clifford_gates`

```
def get_clifford_gates() -> tuple[str, ...]
```

Return available Clifford gates.

Function `merge`

```
def merge(
    a: Gate,
    *bs
) -> <function Gate at 0x7fbf6dae0e60>
```

Merge two gates a and b. The merged Gate will be equivalent to apply

```
new_psi = bs.matrix() @ ... @ b.matrix() @ a.matrix() @ psi
```

with psi a quantum state.

Parameters

a, ...: Gate Gates to merge. **`qubits_order`** : iter[any], optional : If provided, qubits in new Gate will be sorted using qubits_order.

Returns

`Gate('MATRIX')` The merged Gate

Module `hybridq.utils`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Sub-modules

- hybridq.utils.aligned
- hybridq.utils.dot
- hybridq.utils.transpose
- hybridq.utils.utils

Module `hybridq.utils.aligned`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Sub-modules

- hybridq.utils.aligned.aligned_array

Module `hybridq.utils.aligned.aligned_array`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function `array`

```
def array(
    a: any,
    dtype: any = None,
    order: "{'C', 'F', 'A', 'K'}" = 'K',
    *,
    alignment: int = 16,
    copy: bool = True,
    **kwargs
) -> np.ndarray
```

Return a copy of a which is aligned to the given alignment.

Parameters

`a` : any Array to align.
`dtype` : any, optional The type of the new array.
`order` : `{'C', 'F', 'A', 'K'}`, optional Memory layout. 'A' and 'K' depend on the order of input array a. 'C' row-major (C-style), 'F' column-major (Fortran-style) memory representation. 'A' (any) means 'F' if a is Fortran contiguous, 'C' otherwise 'K' (keep) preserve input order. Defaults to 'C'.
`alignment` : int, optional The required alignment.
`copy` : bool, optional It copies a to a new array even if a is already aligned. (default: True)

Returns

np.ndarray The aligned array.

See Also

numpy, hybridq.utils.aligned.empty

Function `asarray`

```
def asarray(
    a: any,
    dtype: any = None,
    order: "{'C', 'F', 'A', 'K'}" = 'K',
    *,
    alignment: int = 16,
    **kwargs
) -> np.ndarray
```

Convert a to an aligned array with the given alignment.

Parameters

`a` **: any, optional** Array to align.
`shape` **: any, optional** The shape of the new array.
`dtype` **: any, optional** The type of the new array.
`order` **: `{'C', 'F', 'A', 'K'}`, optional** Memory layout. 'A' and 'K' depend on the order of input array a. 'C' row-major (C-style), 'F' column-major (Fortran-style) memory representation. 'A' (any) means 'F' if a is Fortran contiguous, 'C' otherwise 'K' (keep) preserve input order. Defaults to 'C'.
`alignment` **: int, optional** The required alignment.

Returns

np.ndarray The aligned array.

See Also

numpy, hybridq.utils.aligned.empty

Function `empty`

```
def empty(
    shape: any,
    dtype: any = builtins.float,
    order: "{'C', 'F'}" = 'C',
    *,
    alignment: int = 16,
    **kwargs
)
```

Return an np.ndarray which is aligned to the given alignment.

Parameters

`shape` **: any** The shape of the new array.
`dtype` **: any, optional** The type of the new array.
`order` **: `{'C', 'F'}`, optional** Memory layout. 'C' row-major (C-style), 'F' column-major (Fortran-style) memory representation. Defaults to 'C'.
`alignment` **: int, optional** The required alignment.

Returns

np.ndarray The aligned array.

See Also

numpy

Function `empty_like`

```
def empty_like(
    a: np.array
) -> <built-in function array>
```

Function `get_alignment`

```
def get_alignment(
    a: np.ndarray,
    max_alignment: int = 128
) -> int
```

Get the largest alignment for a, up to max_alignment.

Parameters

a : np.ndarray Array to get the alignment.
`max_alignment` : int, optional Maximum alignment to check.

Returns

int The maximum alignment of a, up to max_alignment.

Function `isaligned`

```
def isaligned(
    a: np.ndarray,
    alignment: int
) -> bool
```

Return True if a is aligned with alignment.

Parameters

a : np.ndarray Array to check the alignment.
`alignment` : int The desired alignment.

Returns

bool True is a is aligned to alignment, and False otherwise.

Function `ones`

```
def ones(
    shape: any,
    dtype: any = builtins.float,
    order: "{'C', 'F'}" = 'C',
    *,
    alignment: int = 16,
    **kwargs
)
```

Return an np.ndarray of ones which is aligned to the given alignment.

Parameters

shape : any The shape of the new array.
dtype : any, optional The type of the new array.
order : {'C', 'F'}, optional Memory layout. 'C' row-major (C-style), 'F' column-major (Fortran-style) memory representation. Defaults to 'C'.
alignment : int, optional The required alignment.

Returns

np.ndarray The aligned array.

See Also

hybridq.utils.aligned.empty

Function `ones_like`

```
def ones_like(
    a: np.array
) -> <built-in function array>
```

Function `zeros`

```
def zeros(
    shape: any,
    dtype: any = builtins.float,
    order: "{'C', 'F'}" = 'C',
    *,
    alignment: int = 16,
    **kwargs
)
```

Return an np.ndarray of zeros which is aligned to the given alignment.

Parameters

shape : any The shape of the new array.
dtype : any, optional The type of the new array.
order : {'C', 'F'}, optional Memory layout. 'C' row-major (C-style), 'F' column-major (Fortran-style) memory representation. Defaults to 'C'.
alignment : int, optional The required alignment.

Returns

np.ndarray The aligned array.

See Also

hybridq.utils.aligned.empty

Function `zeros_like`

```
def zeros_like(
    a: np.array
) -> <built-in function array>
```

Module `hybridq.utils.dot`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function `dot`

```
def dot(
    a: np.ndarray,
    b: np.ndarray,
    axes_b: iter[int] = None,
    b_as_complex_array: bool = False,
    inplace: bool = False,
    backend: any = 'numpy',
    **kwargs
)
```

Function `to_complex`

```
def to_complex(
    a: array_like,
    b: array_like
)
```

Function `to_complex_array`

```
def to_complex_array(
    a: array_like
)
```

Module `hybridq.utils.transpose`

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function transpose

```
def transpose(
    a: np.ndarray,
    axes: iter[int] = None,
    inplace: bool = False,
    backend: any = 'numpy',
    **kwargs
) -> np.ndarray
```

Module hybridq.utils.utils

Author: Salvatore Mandra (salvatore.mandra@nasa.gov)

Functions

Function argsort

```
def argsort(
    iterable,
    *,
    key=None,
    reverse=False
)
```

Argsort heterogeneous list.

Function isintegral

```
def isintegral(
    x: any
)
```

Return True if x is integral. The test is done by converting the x to int.

Function isnumber

```
def isnumber(
    x: any
)
```

Return True if x is integral. The test is done by converting the x to int.

Function `kron`

```
def kron(
    a: np.ndarray,
    *cs: tuple[np.ndarray, ...],
    **kwargs
)
```

Compute the Kronecker product among multiple arrays.

Parameters

a, cs...: numpy.ndarray Arrays used to compute the Kronecker product

Returns

numpy.ndarray The Kronecker product.

See Also

numpy.kron

Function `load_library`

```
def load_library(
    libname: str,
   prefix: list[str, ...] = (None, 'lib', 'local/lib', 'usr/lib', 'usr/local/lib')
)
```

Function `sort`

```
def sort(
    iterable,
    *,
    key=None,
    reverse=False
)
```

Sort heterogeneous list.

Function `svd`

```
def svd(
    a,
    axes: iter[int],
    sort: bool = False,
    atol: float = 1e-08,
    **kwargs
)
```

Return the SVD of a by splitting it accordingly to axes.

Parameters

`a` **: numpy.ndarray** Array to decompose.
`axes` **: iter[int]** Axes used to split a.
`sort` **: bool, optional** If True, sort Schmidt decomposition.

`atol` **: float, optoinal** Remove all Schmidt decomposition with weight smaller than atol.

Returns

`s, uh, vh:` Decomposition of a in uh and vh, with uh containing axes. s are the weights of the decomposition.

See Also

scipy.linalg.svd

Classes

Class `DeprecationWarning`

```
class DeprecationWarning(
    *args,
    **kwargs
)
```

Base class for warning categories.

Ancestors (in MRO)

- builtins.Warning
- builtins.Exception
- builtins.BaseException

Class `globalize`

```
class globalize(
    f: callable,
    *,
    name: str = None,
    check_if_safe: bool = False
)
```

Globalize any function.

Instance variables

Variable `f`

Variable `name`

Variable `namespace`

Generated by *pdoc* 0.9.2 (https://pdoc3.github.io).