qsearch

Release 2.0.0

Marc Grau Davis, Ethan Smith

Sep 12, 2023

CONTENTS:

1 Gatesets in qsearch **3**
- 1.1 Provided Gatesets . . . 3
- 1.2 Custom Gatesets . . . 4

2 Gates in qsearch **5**
- 2.1 Provided Gates . . . 5
- 2.2 Custom Gates . . . 5

3 API Reference **7**
- 3.1 `qsearch` . . . 7

4 Working with nonlinear topologies **129**

5 Working with nonstandard gates or qutrits **131**

6 Customizing your compilation **133**

7 Indices and tables **135**

Python Module Index **137**

Index **139**

Note: The documentation is currently a work in progress and will be expanded upon soon.

CHAPTER

ONE

GATESETS IN QSEARCH

To synthesize with different gates or topologies, you will need to create an instance of an `qsearch.gatesets.Gateset` subclass.

```
# example: synthesizing for the ring topology
import qsearch as qs
ring_gateset = qs.gatesets.QubitCNOTRing()

# use your gateset with a project
myproject = qs.Project("myproject")
myproject["gateset"] = ring_gateset

# or use it with SearchCompiler directly
mycompiler = qs.SearchCompiler(gateset=ring_gateset)
```

1.1 Provided Gatesets

1.1.1 Basic Gatesets

- `QubitCNOTLinear` - a gateset that is useful for synthesizing circuits with CNOTs and single qubit gates with the linear topology. It is similar to `U3CNOTLinear`, but is slightly more efficient without sacrificing generality. It is the default gateset.
- `U3CNOTLinear` - a gateset based on IBM's U3 gate and CNOTs for the linear topology. It is generally better to use `QubitCNOTLinear`, which is more efficient.
- `ZXZXZCNOTLinear` - a gateset based on the RZ-RX90-RZ-RX90-RZ decomposition of single qubit gates for the linear topology. It is generally better to use `U3CNOTLinear`, which is more efficient.

1.1.2 Nonlinear Topologies

- `QubitCNOTRing` - a gateset that is equivalent to `QubitCNOTLinear` except it implements the ring topology. For 3 qubits, this is the triangle topology and is all-to-all.
- `QubitCNOTAdjacencyList` - a gateset that takes a list of CNOT connections, and creates a gateset that is similar to `QubitCNOTLinear` but uses a toplogy based on the adjacency list. If the desired topology can be achieved by using `QubitCNOTLinear` or `QubitCNOTRing`, it is recommended to choose one of those because it will be more efficient.

```
# This would create a gateset for 4 qubits with CNOT connections 0 to 1, 0 to 2, and 1␣
↪to 3
mygateset = qs.gatesets.QubitCNOTAdjacencyList([(0,1), (0,2), (1,3)])
```

1.1.3 Qutrits

- `QutritCPIPhaseLinear` - a gateset designed for qutrits that uses single qutrit gates and the CPI two-qutrit gate with a phase applied.

1.2 Custom Gatesets

If none of these gatesets suite your needs, you can write your own! Make a subclass of `qs.gatesets.Gateset` and implement these two functions:

- `intial_layer(n)` The single input, `n`, is an integer which deqsribes how many qudits will be in the circuit. The function returns a single `qs.gates.Gate` object representing an initial layer for the search. Normally, this is a kronecker product of single-qudit gates, and you can use the provided `fill_row` helper function to produce this.
- `search_layers(n)` The single input, `n`, is an integer which deqsribes how many qudits will be in the circuit. The function returns a list of `qs.gates.Gate` objects, each representing a possible building block in a possible location for expanding the current circuit.

See the existing implementations in `qs.gatesets` for examples of how to write a gateset.

CHAPTER

TWO

GATES IN QSEARCH

The classes representing quantum gates are found in `gates.py`, and are subclasses of `qsearch.gates.Gate`. You will need to work with `Gate` objects to crate custom gatesets, and you will get a `Gate` object as a return value from compilation.

```
# Here are some examples of what you can do with Gate objects
U3 = qsearch.U3Gate()
CNOT = qsearch.CNOTGate()

# get the matrix that a gate represents in a numpy matrix format
U3_unitary = U3.matrix([np.pi/2, np.pi/4, np./pi/6]) # the array of parameters must be␣
↪provided
CNOT_unitary = CNOT.matrix([]) # cnot takes no parameters so an empty array is provided

# combine multiple gates to form a larger circuit
mycircuit = ProductGate(KroneckerGate(U3,U3), CNOT) # note that mycircuit is itself an␣
↪instance of Gate
```

2.1 Provided Gates

For more information, see the API documentation in `qsearch.gates`.

2.2 Custom Gates

There is an existing gate that can be customized to your needs. However it will not show up when you assembly the circuit to OpenQASM or Qiskit.

- `UGate` - represents the gate deqsribed by the unitary `U` passed to `__init__`, and takes up `qudits` qudits.

You can also write your own `Gate` subclasses the required functions are:

- `__init__` - you must customize the initializer to set `self.num_inputs` to the number of parameters for the gate (e.g. 3 for U3 or ZXZXZ, 0 for CNOT), and `self.qudits` to the number of qudits used by the gate (e.g. 1 for U3 or ZXZXZ, 2 for CNOT).
- `matrix(v)` - here you generate and return the matrix represented by your gate when passed the parameters provided in the array `v`.

2.2.1 Assembling with Custom Gates

If you want your code to output your custom gates when assembling, you must implement `assemble` as well.

- `assemble(v, i)` - here you are given `v`, the list of parameters needed for your gate, and `i`, the index of the first qubit in the set of qubits that your gate is assigned. You must return an array of the form `[gate1, gate2]` where `gate1` and `gate2` are tuples that represent gates that the assembler will be able to interpret. For example, `ZXZXZGate` returns an array for 5 tuples, one for each of the Z and X gates that it is based on, but `U3Gate` only returns an array of 1 tuple because the assembler interprets it as a single gate. The tuples take the form `("gate", gatename, parameters, indices)`, where the word "gate" is included to specify that this tuple represents a well defined gate as opposed to a Kronecker product of gates, `gatename` is a string that will be used to look up the relevant format to print this gate when assembling, and `parameters` is a list of the parameters formatted and organized the way they are needed to fill the format specified in the assembler, and `indices` is a list of the indices of the involved qubits.

2.2.2 Faster Solving with Jacobians

If you would like to take advantage of faster solvers that can take advantage of the Jacobian (marked with `Jac` in their name), and your custom gate uses one or more parameters, you will need to implement `mat_jac` as well.

- `mat_jac(v)` - here you generate and return a tuple `(U, [J1, ... ,Jn])` where `U` is the same matrix you would return in `matrix`, and `[J1, ... ,Jn]` is a list of matrix derivatives, where `J1` is the matrix of derivatives with respect to the first parameter in `v`, and so on for all the parameters in v. **If your custom gate is constant (`self.num_inputs == 0`), then you can take advantage of Jacobian solvers without implementing `mat_jac` yourself.**

CHAPTER

THREE

API REFERENCE

This page contains auto-generated API reference documentation[1].

3.1 qsearch

3.1.1 Submodules

qsearch.advanced_unitaries

A collection of constant gates and gate generators that are unusual or more complicated than those found in unitaries.py.

Module Contents

Functions

generate_miro()	Generates a gate that was described as the X gate on the space of multiple qubits.
generate_HHL()	

Attributes

mirogate
HHL

qsearch.advanced_unitaries.**generate_miro**()

Generates a gate that was described as the X gate on the space of multiple qubits.

qsearch.advanced_unitaries.**mirogate**

qsearch.advanced_unitaries.**generate_HHL**()

[1] Created with sphinx-autoapi

qsearch.advanced_unitaries.**HHL**

qsearch.assemblers

This module defines the Assembler class, which is used to convert a Qsearch-style circuit into other formats, such as Qiskit or Qasm.

The DictionaryAssembler subclass is provided as the default implementation of an Assembler. Use it as-is or as an example for writing your own Assembler. Some constants are also defined as DictionaryAssembler instances preloaded with the most common assembly dictionaries.

qsearch.assemblers.**ASSEMBLER_QISKIT**

Outputs Python code that generates a Qiskit circuit object.

qsearch.assemblers.**ASSEMBLER_OPENQASM**

Outputs generic Openqasm code. This may not be compatible with IBM Qiskit.

qsearch.assemblers.**ASSEMBLER_IBMOPENQASM**

Outputs Openqasm code with the IBM imports and gate names. This flavor of Openqasm is compatible with IBM Qiskit.

qsearch.assemblers.**ASSEMBLER_QUTRIT**

Outputs pseudocode for circuits built with single-qutrit gates and CNOTs.

Module Contents

Classes

Assembler	This class is used to translate Qsearch-style circuits to other formats.
DictionaryAssembler	This subclass of Assembler uses a dictionary that specifies mappings from gate names to output code, as well as an output code initial line.

Functions

flatten_intermediate(intermediate)	This is a helper function for working with the intermediate tuple language that is output by the assemble method of QuantumStep objects.

Attributes

assemblydict_qiskit
assemblydict_openqasm
assemblydict_ibmopenqasm
assemblydict_qutrit
ASSEMBLER_QISKIT
ASSEMBLER_OPENQASM
ASSEMBLER_IBMOPENQASM
ASSEMBLER_QUTRIT

class qsearch.assemblers.**Assembler**(*options=Options()*)

This class is used to translate Qsearch-style circuits to other formats.

abstract assemble(*resultdict*, *options=None*)

The assemble function is used to convert the circuit described in resultdict. See DictionaryAssembler for an example implementation.

Parameters
resultdict – The dictionary representing the desired circuit. It is expected to contain the entries "stucture" and "parameters". It may contain other entries.

Returns
A string representing the converted circuit code.

Return type
str

class qsearch.assemblers.**DictionaryAssembler**(*options=Options()*)

Bases: *Assembler*

This subclass of Assembler uses a dictionary that specifies mappings from gate names to output code, as well as an output code initial line.

Options: assemblydict (required) : A dictionary that specifies mappings from gate names to output code.

assemble(*resultdict*, *options=None*)

The assemble function is used to convert the circuit described in resultdict. See DictionaryAssembler for an example implementation.

Parameters
resultdict – The dictionary representing the desired circuit. It is expected to contain the entries "stucture" and "parameters". It may contain other entries.

Returns
A string representing the converted circuit code.

Return type
str

qsearch.assemblers.**flatten_intermediate**(*intermediate*)

This is a helper function for working with the intermediate tuple language that is output by the assemble method of QuantumStep objects.

qsearch.assemblers.**assemblydict_qiskit**

qsearch.assemblers.**assemblydict_openqasm**

qsearch.assemblers.**assemblydict_ibmopenqasm**

qsearch.assemblers.**assemblydict_qutrit**

qsearch.assemblers.**ASSEMBLER_QISKIT**

qsearch.assemblers.**ASSEMBLER_OPENQASM**

qsearch.assemblers.**ASSEMBLER_IBMOPENQASM**

qsearch.assemblers.**ASSEMBLER_QUTRIT**

qsearch.backends

This module describes Backend, a class which is called before the Solver is run in order to replace a Python Qsearch circuit with a Qsearch circuit based on another implementation, such as Rust or GPU.

There are three provided Backend implementations:

qsearch.backends.**PythonBackend**

This simply returns the Python circuit, such that Python and Numpy are used for computation.

qsearch.backends.**NativeBackend**

This returns the converted circuit from native_from_object, which uses the Rust implementation of Qsearch circuits provided in the qsrs module.

qsearch.backends.**SmartDefaultBackend**

This backend tries to use the native Rust backend, but if it fails to convert the circuit (such as if there are unsupported gates), it will fallback to Python rather than throwing an error.

Module Contents

Classes

Backend	This class prepares a circuit for solving, replacing a Python circuit with another implementation.
SmartDefaultBackend	This Backend tries to use the native Rust code, but will gracefully fallback to Python if there is an issue.
PythonBackend	This Backend will simply return the Python Qsearch circuit passed in, therefore using Python and Numpy for matrix computation.
NativeBackend	This Backend will use the native Rust implementation of Qsearch circuits for faster matrix computation.

Attributes

RUST_ENABLED

qsearch.backends.**RUST_ENABLED = True**

class qsearch.backends.**Backend**(*options=Options()*)

This class prepares a circuit for solving, replacing a Python circuit with another implementation.

abstract prepare_circuit(*circ*, *options=None*)

This function accepts a Python Qsearch circuit and returns a Qsearch circuit with a different implementation. :param circ: The Python Qsearch circuit to be converted.

class qsearch.backends.**SmartDefaultBackend**(*options=Options()*)

Bases: *Backend*

This Backend tries to use the native Rust code, but will gracefully fallback to Python if there is an issue.

prepare_circuit(*circuit*, *options=None*)

This function accepts a Python Qsearch circuit and returns a Qsearch circuit with a different implementation. :param circ: The Python Qsearch circuit to be converted.

class qsearch.backends.**PythonBackend**(*options=Options()*)

Bases: *Backend*

This Backend will simply return the Python Qsearch circuit passed in, therefore using Python and Numpy for matrix computation.

prepare_circuit(*circuit*, *options=None*)

This function accepts a Python Qsearch circuit and returns a Qsearch circuit with a different implementation. :param circ: The Python Qsearch circuit to be converted.

class qsearch.backends.**NativeBackend**(*options=Options()*)

Bases: *Backend*

This Backend will use the native Rust implementation of Qsearch circuits for faster matrix computation.

prepare_circuit(*circuit*, *options=None*)

This function accepts a Python Qsearch circuit and returns a Qsearch circuit with a different implementation. :param circ: The Python Qsearch circuit to be converted.

qsearch.checkpoints

This module defines the Checkpoint class, which is used for storing intermediate state while compiling, to allow an interrupted compilation to resume at a later time.

Two default implementations are provided. It is recommended that you look at FileCheckpoint as an example if you are interested in writing your own implementation.

qsearch.checkpoints.**FileCheckpoint**

Saves and recovers the intermediate state from a file, specified as "statefile" in the options.

qsearch.checkpoints.**ChildCheckpoint**

Allows for hierarchial checkpointing, which is useful in cases where there are sub-compilers, such as with LEAP.

Module Contents

Classes

Checkpoint	This class is used for storing intermediate state while compiling, to allow an interrupted compilation to resume at a later time.
FileCheckpoint	This Checkpoint will store the state in the file specified in the options as statefile.
ChildCheckpoint	This Checkpoint is used for hierarchial checkpointing for when there is a sub-compiler, such as in LEAP.

class qsearch.checkpoints.**Checkpoint**(*options=options.Options()*)

This class is used for storing intermediate state while compiling, to allow an interrupted compilation to resume at a later time.

abstract save(*state*)

Save the passed state to be recovered later. :param state: A Python object representing the intermediate state of the compilation. Usually a dictionary, but it could be anything. :type state: object

abstract recover()

Return the state previously stored with save(state).

Returns
A Python object equivalent to the object originally stored via save(state), or None if no state is saved.

Return type
object

abstract delete()

Delete the state that was stored such that None will be returned next time recover() is called.

class qsearch.checkpoints.**FileCheckpoint**(*options=options.Options()*)

Bases: *Checkpoint*

This Checkpoint will store the state in the file specified in the options as statefile.

Options:
statefile : A string with a filepath where the state will be stored, or None, in which case no state will be stored and None will always be returned by recover()

save(*state*)

Save the passed state to be recovered later. :param state: A Python object representing the intermediate state of the compilation. Usually a dictionary, but it could be anything. :type state: object

recover()

Return the state previously stored with save(state).

Returns
A Python object equivalent to the object originally stored via save(state), or None if no state is saved.

Return type
object

delete()

Delete the state that was stored such that None will be returned next time recover() is called.

class qsearch.checkpoints.**ChildCheckpoint**(*options=options.Options()*)

Bases: *Checkpoint*

This Checkpoint is used for hierarchial checkpointing for when there is a sub-compiler, such as in LEAP.

Options:

parent (required) : The Checkpoint class that the creator of the ChildCheckpoint was passed.

Below is an explanation of how ChildCheckpoint works. See leap_compiler for an example.

My compiler class, ParentCompiler, is passed a FileCheckpoint as options.checkpoint. I create a ChildCheckpoint with the FileCheckpoint as the parent: child_checkpoint = ChildCheckpoint(Options(parent=options.checkpoint)) I pass this ChildCheckpoint to the sub-compiler I create: sub_compiler = SubCompiler(Options(checkpoint=child_checkpoint))

To the SubCompiler, the passed ChildCheckpoint will behave as any other Checkpoint would be expected to behavior, saving state with save(state), and recovering it with recover(), and deleting it with delete().

As the ParentCompiler, you save your state with save_parent(parentstate), recover it with recover_parent(), and deleting it with delete_parent(), making these function calls to child_checkpoint instead of interacting directly with the FileCheckpoint that was originally passed via options.

The states of both ParentCompiler and SubCompiler will get saved in a manner specified by the original FileCheckpoint.

ChildCheckpoint fully conforms to Checkpoint, and makes no assumptions about its parent, so it is compatible with any class that makes use ofa Checkpoint and works with any Checkpoint as a parent. This means you can even have multiple layers of nested ChildCheckpoint.

However, the class creating the ChildCheckpoint must be sure to use the parent functions.

Also, note that calling delete_parent() also deletes the state for the child. However, this is rather uncommon because usually it is the creator of the Checkpoint that calls delete, not the class it is passed to. For example, Project will call delete() to delete the checkpoint from a Compiler. ParentCompiler might call delete() to delete the state of SubCompiler once SubCompiler has finished (in fact, this happens in leap_compiler).

save(*state*)

Save the passed state to be recovered later. :param state: A Python object representing the intermediate state of the compilation. Usually a dictionary, but it could be anything. :type state: object

save_parent(*parentstate*)

Saves the parentstate alongside the child state.

recover()

Return the state previously stored with save(state).

Returns

A Python object equivalent to the object originally stored via save(state), or None if no state is saved.

Return type

object

recover_parent()

Recovers the parentstate.

delete()

Delete the state that was stored such that None will be returned next time recover() is called.

delete_parent()

Deletes the state. Note that this delete both the parentstate and the child state.

qsearch.comparison

This module contains functions for comparing matrices, vectors, and other numerical objects. These functions do not all follow a standardized form, but many of these have a standardized version found in evaluation.py.

Module Contents

Functions

matrix_distance_squared(A, B)	This is a distance function used to compare two matrices. It is phase agnostic and fast to calculate.
matrix_distance(A, B)	The square root of matrix_distance_squared is more analgous to "distance", although for most purposes, working with a distance squared is fine, since inequalities hold.
matrix_distance_squared_jac(U, M, J)	The jacobian version of matrix_distance_squared.
matrix_residuals(A, B, I)	
matrix_residuals_jac(U, M, J)	
matrix_residuals_v2(A, B, I)	
matrix_residuals_v2_jac(U, M, J)	
matrix_residuals_slice(slices, A, B, I)	
matrix_residuals_slice_jac(slices, A, B, J)	
matrix_residuals_blacklist(badrows, badcols, A, B, I)	
matrix_residuals_blacklist_jac(badrows, badcols, A, B, J)	
distance_with_initial_state(stateA, stateB, A, B)	
distance_with_initial_state_jac(stateA, stateB, A, B, J)	
residuals_with_initial_state(stateA, stateB, A, B, I)	
residuals_with_initial_state_jac(stateA, stateB, U, M, J)	
eval_func_from_residuals(f, A, B)	

qsearch.comparison.**matrix_distance_squared**(*A*, *B*)

This is a distance function used to compare two matrices. It is phase agnostic and fast to calculate.

Parameters

- **A** – A unitary matrix in the form of a numpy ndarray.
- **B** – Another unitary matrix of the same size as A, as a numpy ndarray.

Returns
A single value between 0 and 1, representing how closely A and B match. A value near 0 indicates that A and B are the same unitary, up to an overall phase difference.

Return type
Float

qsearch.comparison.**matrix_distance**(*A*, *B*)

The square root of matrix_distance_squared is more analgous to "distance", although for most purposes, working with a distance squared is fine, since inequalities hold.

Parameters

- **A** – A unitary matrix in the form of a numpy ndarray.
- **B** – Another unitary matrix of the same size as A, as a numpy ndarray.

Returns
A single value between 0 and 1, representing how closely A and B match. A value near 0 indicates that A and B are the same unitary, up to an overall phase difference.

Return type
Float

qsearch.comparison.**matrix_distance_squared_jac**(*U*, *M*, *J*)

The jacobian version of matrix_distance_squared.

Parameters

- **U** – A constant unitary matrix in the form of a numpy ndarray.
- **M** – A variable unitary matrix of the same size as A, as a numpy ndarray.
- **J** – A list of nump ndarrays representing the jacobians of M with respect to the parameters of interest.

Returns
The matrix distance squared as a float (the same value that would be returned from matrix_distance_squared) jacs : A list of the derivative of dsq with resepect to each of the parameters.

Return type
dsq

qsearch.comparison.**matrix_residuals**(*A*, *B*, *I*)

qsearch.comparison.**matrix_residuals_jac**(*U*, *M*, *J*)

qsearch.comparison.**matrix_residuals_v2**(*A*, *B*, *I*)

qsearch.comparison.**matrix_residuals_v2_jac**(*U*, *M*, *J*)

qsearch.comparison.**matrix_residuals_slice**(*slices*, *A*, *B*, *I*)

qsearch.comparison.**matrix_residuals_slice_jac**(*slices*, *A*, *B*, *J*)

qsearch.comparison.**matrix_residuals_blacklist**(*badrows*, *badcols*, *A*, *B*, *I*)

qsearch.comparison.**matrix_residuals_blacklist_jac**(*badrows*, *badcols*, *A*, *B*, *J*)

qsearch.comparison.**distance_with_initial_state**(*stateA*, *stateB*, *A*, *B*)

qsearch.comparison.**distance_with_initial_state_jac**(*stateA*, *stateB*, *A*, *B*, *J*)

qsearch.comparison.**residuals_with_initial_state**(*stateA*, *stateB*, *A*, *B*, *I*)

qsearch.comparison.**residuals_with_initial_state_jac**(*stateA*, *stateB*, *U*, *M*, *J*)

qsearch.comparison.**eval_func_from_residuals**(*f*, *A*, *B*)

qsearch.compiler

This module defines the Compiler class, which is a framework for classes that take a unitary and return a circuit implementing that unitary.

The default implementation SearchCompiler is also defined here. SearchCompiler compiles the desired unitary using an A* search strategy, as described in the paper Towards Optimal Topology Aware Quantum Circuit Synthesis.

Module Contents

Classes

Compiler	This class defines the pattern for compilers that convert a unitary matrix to a circuit that implements that matrix.
SearchCompiler	This Compiler uses an A* search strategy to synthesize a unitary, as described in the paper Towards Optimal Topology Aware Quantum Circuit Synthesis.

class qsearch.compiler.**Compiler**(*options=Options()*)

This class defines the pattern for compilers that convert a unitary matrix to a circuit that implements that matrix.

abstract compile(*options*)

class qsearch.compiler.**SearchCompiler**(*options=Options()*)

Bases: *Compiler*

This Compiler uses an A* search strategy to synthesize a unitary, as described in the paper Towards Optimal Topology Aware Quantum Circuit Synthesis.

Options:

target (required) : The unitary matrix to be synthesized, in the form of a numpy ndarray with dtype="complex128". gateset : The Gateset used for synthesis. weight_limit : A limit on the maximum weight for circuits to be expanded for further searching. See gatesets.py for more information. The default is None for unlimited. heuristic : A heuristic used to order the search tree. See heuristics.py for more information. solver : A Solver used for optimizing the parameters in parameterized circuits generated by the search tree. parallelizer : A Parallelizer used for solving multiple parameterized circuits in parallel. beams : The number of nodes to pop from the search tree at a time. The default value of -1 will create enough branches to maximize utilization of your CPU. objective : An Objective used for scoring the quality of a parameterization for both synthesis and search. timeout : An uper limit on the amount of time the compiler will spend trying to synthesize a circuit. The default is float('inf'), for unlimited. checkpoint : The compiler will use this Checkpoint to save intermediate state, and will resume from this Checkpoint if there was an existing state. logger : A qsearch.logging.Logger that will be used for logging the synthesis process.

Parameters
options – See class level documentation for the options SearchCompiler uses

compile(*options=Options()*)

Parameters
options – See class level documentation for the options SearchCompiler uses

qsearch.defaults

This module provides defaults for Options objects. This includes definitions of smart_default functions, and dictionaries to be used with set_defaults and set_smart_defaults.

Three default dictionaries are provided.

qsearch.defaults.**standard_defaults**
A dictionary containing defaults for standard gate synthesis.

qsearch.defaults.**standard_smart_defaults**
A dictionary containing smart_defaults functions for standard gate synthesis.

qsearch.defaults.**stateprep_defaults**
A dictionary containing defaults for stateprep synthesis.

Module Contents

Functions

default_heuristic(options)
default_logger(options)
default_checkpoint(options)
default_eval_func(options)
default_error_func(options)
default_error_residuals(options)
default_error_jac(options)
default_error_residuals_jac(options)
default_objective(options)
stateprep_initial_state(options)
stateprep_target(options)
default_compiler(options)
identity(U)

Attributes

standard_defaults
standard_smart_defaults
stateprep_defaults
stateprep_smart_defaults

qsearch.defaults.**default_heuristic**(*options*)

qsearch.defaults.**default_logger**(*options*)

qsearch.defaults.**default_checkpoint**(*options*)

qsearch.defaults.**default_eval_func**(*options*)

qsearch.defaults.**default_error_func**(*options*)

qsearch.defaults.**default_error_residuals**(*options*)

qsearch.defaults.**default_error_jac**(*options*)

qsearch.defaults.**default_error_residuals_jac**(*options*)

qsearch.defaults.**default_objective**(*options*)

qsearch.defaults.**stateprep_initial_state**(*options*)

qsearch.defaults.**stateprep_target**(*options*)

qsearch.defaults.**default_compiler**(*options*)

qsearch.defaults.**identity**(*U*)

qsearch.defaults.**standard_defaults**

qsearch.defaults.**standard_smart_defaults**

qsearch.defaults.**stateprep_defaults**

qsearch.defaults.**stateprep_smart_defaults**

qsearch.evaluation

This module contains functions for comparing and otherwise evaluating matrices, including distance functions, cost functions, and constraint functions.

The standarized format for these types of functions is as follows:

def my_func(circuit, parameters, target, options):
    return <one or more real-valued numbers>

def my_func_jac(circuit, parameters, target, jacs, options):
    return <one or more real-valued numbers>

Module Contents

Functions

error_distsq(circuit, parameters, target, options)	
error_distsq_jac(circuit, parameters, target, jacs, ...)	
error_stateprep_distsq(circuit, parameters, target, ...)	
error_stateprep_distsq_jac(circuit, parameters, ...)	
residuals_product(circuit, parameters, target, options)	
residuals_product_jac(circuit, parameters, target, options)	
residuals_difference(circuit, parameters, target, options)	
residuals_difference_jac(circuit, parameters, target, ...)	
residuals_slice(circuit, parameters, target, options)	
residuals_slice_jac(circuit, parameters, target, options)	
residuals_blacklist(circuit, parameters, target, options)	
residuals_blacklist_jac(circuit, parameters, target, ...)	
cost_linear(circuit, parameters, target, options)	
cost_linear_jac(circuit, parameters, target, options)	
constraint_distsq(circuit, parameters, target, options)	
constraint_distsq_jac(circuit, parameters, target, ...)	
cost_combo_linear(circuit, parameters, target, options)	
cost_combo_linear_jac(circuit, parameters, target, ...)	

qsearch.evaluation.**error_distsq**(*circuit*, *parameters*, *target*, *options*)

qsearch.evaluation.**error_distsq_jac**(*circuit*, *parameters*, *target*, *jacs*, *options*)

qsearch.evaluation.**error_stateprep_distsq**(*circuit*, *parameters*, *target*, *options*)

qsearch.evaluation.**error_stateprep_distsq_jac**(*circuit*, *parameters*, *target*, *options*)

qsearch.evaluation.**residuals_product**(*circuit*, *parameters*, *target*, *options*)

qsearch.evaluation.**residuals_product_jac**(*circuit*, *parameters*, *target*, *options*)

qsearch.evaluation.**residuals_difference**(*circuit*, *parameters*, *target*, *options*)

qsearch.evaluation.**residuals_difference_jac**(*circuit*, *parameters*, *target*, *options*)

qsearch.evaluation.**residuals_slice**(*circuit*, *parameters*, *target*, *options*)

qsearch.evaluation.**residuals_slice_jac**(*circuit*, *parameters*, *target*, *options*)

qsearch.evaluation.**residuals_blacklist**(*circuit*, *parameters*, *target*, *options*)

qsearch.evaluation.**residuals_blacklist_jac**(*circuit*, *parameters*, *target*, *options*)

qsearch.evaluation.**cost_linear**(*circuit*, *parameters*, *target*, *options*)

qsearch.evaluation.**cost_linear_jac**(*circuit*, *parameters*, *target*, *options*)

qsearch.evaluation.**constraint_distsq**(*circuit*, *parameters*, *target*, *options*)

qsearch.evaluation.**constraint_distsq_jac**(*circuit*, *parameters*, *target*, *jacs*, *options*)

qsearch.evaluation.**cost_combo_linear**(*circuit*, *parameters*, *target*, *options*)

qsearch.evaluation.**cost_combo_linear_jac**(*circuit*, *parameters*, *target*, *jacs*, *options*)

qsearch.gates

This module defines the Gate class, which represents a quantum gate, as well as implementations of many common Gates.

Through the use of KroneckerGate and ProductGate, Gates can be formed for complex circuit structures. The matrix and mat_jac functions are used for numerical optimization of parameterized gates. The assemble function is used to generate an intermediate language of tuples that can be used by Assemblers to output descriptions of quantum circuits in other formats.

Module Contents

Classes

Gate	This class shows the framework for working with quantum gates in Qsearch.
IdentityGate	Represents an identity gate of any number of qudits of any size.
XGate	Represents a parameterized X rotation on one qubit.
YGate	Represents a parameterized Y rotation on one qubit.
ZGate	Represents a parameterized Z rotation on one qubit.
SXGate	Represents a sqrt(X) rotation on one qubit, which is equivalent to XGate() with a paramter of pi/2, up to an overall phase.
ZXZXZGate	Represents an arbitrary parameterized single-qubit gate, decomposed into 3 parameterized Z gates separated by X(PI/2) gates.
XZXZGate	Represents a partially parameterized single qubit gate, equivalent to ZXZXZ but without the first Z gate. This is useful because that first Z gate can commute through the control of a CNOT, thereby reducing the number of parameters we need to solve for.
U3Gate	Represents an arbitrary parameterized single qubit gate, parameterized in the same way as IBM's U3 gate.
U2Gate	Represents a parameterized single qubit gate, parameterized in the same way as IBM's U2 gate.
U1Gate	Represents an parameterized single qubit gate, parameterized in the same way as IBM's U1 gate.
SingleQutritGate	This gate represents an arbitrary parameterized single-qutrit gate.
CSUMGate	Represents the constant two-qutrit gate CSUM
CPIGate	Represents the constant two-qutrit gate CPI.
CPIPhaseGate	Represents the constant two-qutrit gate CPI with phase differences.
CNOTGate	Represents the constant two-qubit gate CNOT.
CZGate	Represents the constant two-qubit gate Controlled-Z.
ISwapGate	Represents the constant two-qubit gate ISwap.
XXGate	Represents the constant two-qubit gate XX(pi/2).
NonadjacentCNOTGate	Represents the two-qubit gate CNOT, but between two qubits that are not necessarily next to each other.
UGate	Represents an arbitrary constant gate, defined by the unitary passed to the initializer.
UpgradedConstantGate	Represents a constant gate, based on the Gate passed to its initializer, but upgraded to act on qudits of a larger size.
CUGate	Represents an arbitrary controlled gate, defined by the unitary passed to the initializer.
CNOTRootGate	Represents the sqrt(CNOT) gate. Two sqrt(CNOT) gates in a row will form a CNOT gate.
KroneckerGate	Represents the Kronecker product of a list of gates. This is equivalent to performing those gate in parallel in a quantum circuit.
ProductGate	Represents a matrix product of Gates. This is equivalent to performing those gates sequentially in a quantum circuit.

Attributes

native_from_object

qsearch.gates.**native_from_object**

class qsearch.gates.**Gate**

This class shows the framework for working with quantum gates in Qsearch.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

abstract matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

mat_jac(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

abstract assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__eq__(*other*)
Return self==value.

__hash__()
Return hash(self).

copy()

_parts()

__copy__()

__deepcopy__(*memo*)

__repr__()
Return repr(self).

validate_structure()

class qsearch.gates.**IdentityGate**(*qudits=1*, *d=2*)
Bases: *Gate*

Represents an identity gate of any number of qudits of any size.

Parameters

- **qudits** – The number of qudits represented by this identity.
- **d** – The size of qudits represented by this identity (2 for qubits, 3 for qutrits, etc.)

matrix(*v*)
Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.gates.**XGate**

Bases: *Gate*

Represents a parameterized X rotation on one qubit.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

mat_jac(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.gates.**YGate**

Bases: *Gate*

Represents a parameterized Y rotation on one qubit.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`mat_jac`(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

`assemble`(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

`__repr__`()

Return repr(self).

class qsearch.gates.**ZGate**

Bases: *Gate*

Represents a parameterized Z rotation on one qubit.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

`matrix`(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`mat_jac`(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

`assemble`(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.gates.**SXGate**

Bases: *Gate*

Represents a sqrt(X) rotation on one qubit, which is equivalent to XGate() with a paramter of pi/2, up to an overall phase.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.gates.**ZXZXZGate**

Bases: *Gate*

Represents an arbitrary parameterized single-qubit gate, decomposed into 3 parameterized Z gates separated by X(PI/2) gates.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

mat_jac(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.gates.**XZXZGate**

Bases: *Gate*

Represents a partially parameterized single qubit gate, equivalent to ZXZXZ but without the first Z gate. This is useful because that first Z gate can commute through the control of a CNOT, thereby reducing the number of parameters we need to solve for.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

mat_jac(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.gates.**U3Gate**

Bases: *Gate*

Represents an arbitrary parameterized single qubit gate, parameterized in the same way as IBM's U3 gate.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`mat_jac`(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

`assemble`(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

`__eq__`(*other*)

Return self==value.

`__repr__`()

Return repr(self).

class qsearch.gates.**U2Gate**

Bases: *Gate*

Represents a parameterized single qubit gate, parameterized in the same way as IBM's U2 gate.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

`matrix`(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`mat_jac`(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

`assemble`(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__eq__(*other*)

Return self==value.

__repr__()

Return repr(self).

class qsearch.gates.**U1Gate**

Bases: *Gate*

Represents an parameterized single qubit gate, parameterized in the same way as IBM's U1 gate.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

mat_jac(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).

- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__eq__(*other*)
Return self==value.

__repr__()
Return repr(self).

class qsearch.gates.**SingleQutritGate**

Bases: *Gate*

This gate represents an arbitrary parameterized single-qutrit gate.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

matrix(*v*)
Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

mat_jac(*v*)
Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.gates.**CSUMGate**

Bases: *Gate*

Represents the constant two-qutrit gate CSUM

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

_csum

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`assemble`(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

`__repr__`()

Return repr(self).

`class qsearch.gates.CPIGate`

Bases: *Gate*

Represents the constant two-qutrit gate CPI.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

`_cpi`

`matrix`(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`assemble`(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

`__repr__`()

Return repr(self).

class qsearch.gates.**CPIPhaseGate**

Bases: *Gate*

Represents the constant two-qutrit gate CPI with phase differences.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

`matrix`(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`assemble`(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()
Return repr(self).

class qsearch.gates.**CNOTGate**

Bases: *Gate*

Represents the constant two-qubit gate CNOT.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

_cnot

__eq__(*other*)
Return self==value.

matrix(*v*)
Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)
Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.gates.**CZGate**

Bases: *Gate*

Represents the constant two-qubit gate Controlled-Z.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

_gate

__eq__(*other*)

Return self==value.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).

- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()
Return repr(self).

class qsearch.gates.**ISwapGate**

Bases: *Gate*

Represents the constant two-qubit gate ISwap.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

_gate

__eq__(*other*)
Return self==value.

matrix(*v*)
Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)
Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()
Return repr(self).

class qsearch.gates.**XXGate**

Bases: *Gate*

Represents the constant two-qubit gate XX(pi/2).

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

_gate

__eq__(*other*)
Return self==value.

matrix(*v*)
Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)
Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

`__repr__()`
Return repr(self).

`class qsearch.gates.NonadjacentCNOTGate(qudits, control, target)`
Bases: `Gate`

Represents the two-qubit gate CNOT, but between two qubits that are not necessarily next to each other.

Parameters

- **qudits** – The total number of qubits that a unitary of the size returned by this gate would represent. For this gate, usually this is the total number of qubits in the larger circuit.
- **control** – The index of the control qubit, relative to the 0th qubit that would be affected by the unitary returned by this gate.
- **target** – The index of the target qubit, relative to the 0th qubit that would be affected by the unitary returned by this gate.

`matrix(v)`
Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`assemble(v, i=0)`
Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

validate_structure()

class qsearch.gates.**UGate**(*U*, *d=2*, *gatename='CUSTOM'*, *gateparams=()*, *gateindices=None*)

Bases: Gate

Represents an arbitrary constant gate, defined by the unitary passed to the initializer.

Parameters

- **U** – The unitary for the operation that this gate represents, as a numpy ndarray with datatype="complex128".
- **d** – The size of qudits for the operation that this gate represents. The default is 2, for qubits.
- **gatename** – A name for this gate, which will get passed to the Assembler at assembly time.
- **gateparams** – A tuple of parameters that will get passed to the Assembler at assembly time.
- **gateindices** – A tuple of indices for the qubits that this gate acts on, which will get passed to the Assembler at assembly time. This overrides the default behavior, which is to return a tuple of all the indices starting with the one passed in assemble(v, i), and ending at i+self.qudits

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters

v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns

A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type

np.ndarray

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns

A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

`__repr__()`

Return repr(self).

`class qsearch.gates.UpgradedConstantGate(other, df=3)`

Bases: *Gate*

Represents a constant gate, based on the Gate passed to its initializer, but upgraded to act on qudits of a larger size.

Parameters

- **other** – A Gate of a lower qudit size.
- **df** – The final, upgraded qudit size. The default is 3, for upgrading gates from qubits to qutrits.

`matrix(v)`

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`assemble(v, i=0)`

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.gates.**CUGate**(*U*, *gatename='Name'*, *gateparams=()*, *flipped=False*)

Bases: Gate

Represents an arbitrary controlled gate, defined by the unitary passed to the initializer.

Parameters

- **U** – The unitary to form the controlled-unitary gate, in the form of a numpy ndarray with dtype="complex128"
- **gatename** – A name for this controlled gate which will get passed to the Assembler at assembly time.
- **gateparams** – A tuple of parameters that will get passed to the Assembler at assembly time.
- **flipped** – A boolean flag, which if set to true, will flip the direction of the gate. The default direction is for the control qubit to be the lower indexed qubit.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.gates.**CNOTRootGate**

Bases: *Gate*

Represents the sqrt(CNOT) gate. Two sqrt(CNOT) gates in a row will form a CNOT gate.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

_cnr

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.gates.**KroneckerGate**(**subgates*)

Bases: *Gate*

Represents the Kronecker product of a list of gates. This is equivalent to performing those gate in parallel in a quantum circuit.

Parameters

***subgates** – An sequence of Gates. KroneckerGate will return the kronecker product of the unitaries returned by those Gates.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters

v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns

A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type

np.ndarray

mat_jac(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters

v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns

A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type

tuple

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

appending(*gate*)

Returns a new KroneckerGate with the new gate added to the list.

Parameters
gate – A Gate to be added to the end of the list of gates in the new KroneckerGate.

_parts()

__deepcopy__(*memo*)

__repr__()

Return repr(self).

validate_structure()

class qsearch.gates.**ProductGate**(**subgates*)

Bases: Gate

Represents a matrix product of Gates. This is equivalent to performing those gates sequentially in a quantum circuit.

Parameters
subgates – A list of Gates to be multiplied together. ProductGate returns the matrix product of the unitaries returned by those Gates.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`mat_jac`(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

`assemble`(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

`appending`(**gates*)

Returns a new ProductGate with the new gates appended to the end.

Parameters
gates – A list of Gates to be appended.

`inserting`(**gates*, *depth=-1*)

Returns a new ProductGate with new *gates* inserted at some index *depth*.

Parameters

- **gates** – A list of Gates to be inserted.
- **depth** – An index in the subgates of the ProductGate after which the new gates will be inserted. The default value of -1 will insert these gates at the begining of the ProductGate.

__deepcopy__(*memo*)

__repr__()

Return repr(self).

validate_structure()

qsearch.gatesets

This module defines the Gateset class, which represents the allowed gates and topology for a specific quantum computer.

Several Implementations of Gateset are also defined here. Several aliases are also defined, for the most common use cases.

qsearch.gatesets.**ZXZXZCNOTLinear**

A Gateset that uses CNOT and the ZXZXZ single qubit parameterization with the linear topology.

qsearch.gatesets.**U3CNOTLinear**

A Gateset that uses CNOT and the U3 single qubit parameterization with the linear topology.

qsearch.gatesets.**QubitCNOTLinear**

A Gateset that uses CNOT and the U3 single qubit parameterization with the linear topology, except it uses an XZXZ instead of a U3 after the control qubit of each CNOT. This results in a gateset that covers the same search space as U3CNOTLinear, but with fewer redundant parameters, and therefore faster runtime.

qsearch.gatesets.**QubitCNOTRing**

Uses U3 and XZXZ like QubitCNOTLinear, but includes a NonadjacentCNOTGate to add a link from the last qubit to the 0th.

qsearch.gatesets.**QubitCNOTAdjacencyList**

Similar to QubitCNOTLinear and QubitCNOTRing, but takes in an adjacency list which uses NonadjacentCNOTGate to define work with a custom topology.

qsearch.gatesets.**QutritCPIPhaseLinear**

A qutrit gateset that uses the CPIPhase gate as its two-qutrit gate, with a linear topology.

qsearch.gatesets.**QutritCNOTLinear**

A qutrit gateset that uses an upgraded version of the CNOT gate as its two-qutrit gate, with a linear topology.

qsearch.gatesets.**DefaultQubit**

The default Gateset for working with qubits. Currently is equivalent to QubitCNOTLinear.

qsearch.gatesets.**DefaultQutrit**

The default Gateset for working with qutrits. Currently is equivalent to QutritCPIPhaseLinear.

qsearch.gatesets.**Default**

The overall default Gateset, which is equivalent to DefaultQubit.

Module Contents

Classes

Gateset	This class defines the supported gates and topology for a specific quantum hardware.
ZXZXZCNOTLinear	A Gateset for working with CNOT and single-qubit gates parameterized with ZXZXZGate on the linear topology.
U3CNOTLinear	A Gateset for working with CNOT and single-qubit gates parameterized with U3Gate on the linear topology.
QubitCNOTLinear	A Gateset for working with CNOT and single-qubit gates parameterized with U3Gate and XZXZGate on the linear topology. This Gateset covers the same search space but uses fewer parameters than ZXZXZCNOTLinear and U3CNOTLinear.
QubitCNOTRing	A Gateset for working with CNOT and single-qubit gates parameterized with U3Gate and XZXZGate on the ring topology.
QubitCZLinear	A Gateset for working with CZ and single-qubit gates parameterized with U3Gate and XZXZGate on the linear topology.
QubitISwapLinear	A Gateset for working with ISwap and single-qubit gates parameterized with U3Gate and XZXZGate on the linear topology.
QubitXXLinear	A Gateset for working with ISwap and single-qubit gates parameterized with U3Gate and XZXZGate on the linear topology.
QubitCNOTAdjacencyList	A Gateset for working with CNOT and single-qubit gates parameterized with U3Gate and XZXZGate on a custom topology, specified in the initializer.
QutritCPIPhaseLinear	A Gateset for working with CPIPhase and single-qutrit gates on the linear topology.
QutritCNOTLinear	A hybrid Gateset for working with CNOT and single-qutrit gates on the linear topology.

Functions

linear_topology(double_gate, single_gate, n, d[, ...])	
fill_row(gate, n)	
find_last_3_cnots_linear(circuit)	

Attributes

DefaultQubit
DefaultQutrit
Default

class qsearch.gatesets.**Gateset**

This class defines the supported gates and topology for a specific quantum hardware.

Gatesets must set the value of d in their initializer, which represents the size of qudits that are supported (e.g. 2 for qubits or 3 for qutrits).

initial_layer(*qudits*)

The initial layer in the compilation. Usually a layer of parameterized single-qudit gates.

Parameters
qudits – The number of qudits in this circuit.

Returns
A single Gate representing an initial layer for the circuit

Return type
qsearch.gates.Gate

search_layers(*qudits*)

A set of possible multi-qubit gates for searching. Usually this is a two-qudit gate followed by two single-qudit gates, for every allowed placement of the two-qudit gate. This defines the branching factor of the search tree.

Parameters
qudits – The number of qudits in this circuit

Returns
A list of tuples of (gate,weight) where Gate is the Gate representing that possible placement of the two-qudit gate, and weight is the weight or cost of adding that gate in that placement to the final circuit.

Return type
list

branching_factor(*qudits*)

Returns an integer indicating the expected branching factor. Usually this is automatically determined from search_layers, but it may need to be overridden if successors is overridden.

Parameters
qudits – The number of qudits in this circuit

Returns
An integer indicating the expecte branching factor

Return type
int

successors(*circ*, *qudits=None*)

Returns a list of Gates that are successors in the search tree to the input Gate, circ, representing a current ansatz circuit.

Parameters

- **circ** – The curret ansatz Gate.
- **qudits** – The number of qudits in this circuit.

Returns
A list of tuples of (gate, weight) where gate is a Gate that is a successor to circ, and weight is the cost or weight of moving to gate from circ.

Return type
list

__eq__(*other*)

Return self==value.

class qsearch.gatesets.**ZXZXZCNOTLinear**

Bases: *Gateset*

A Gateset for working with CNOT and single-qubit gates parameterized with ZXZXZGate on the linear topology.

Gatesets must set the value of d in their initializer, which represents the size of qudits that are supported (e.g. 2 for qubits or 3 for qutrits).

initial_layer(*n*)

The initial layer in the compilation. Usually a layer of parameterized single-qudit gates.

Parameters
qudits – The number of qudits in this circuit.

Returns
A single Gate representing an initial layer for the circuit

Return type
qsearch.gates.Gate

search_layers(*n*)

A set of possible multi-qubit gates for searching. Usually this is a two-qudit gate followed by two single-qudit gates, for every allowed placement of the two-qudit gate. This defines the branching factor of the search tree.

Parameters
qudits – The number of qudits in this circuit

Returns
A list of tuples of (gate,weight) where Gate is the Gate representing that possible placement of the two-qudit gate, and weight is the weight or cost of adding that gate in that placement to the final circuit.

Return type
list

class qsearch.gatesets.**U3CNOTLinear**

Bases: *Gateset*

A Gateset for working with CNOT and single-qubit gates parameterized with U3Gate on the linear topology.

Gatesets must set the value of d in their initializer, which represents the size of qudits that are supported (e.g. 2 for qubits or 3 for qutrits).

initial_layer(*n*)

The initial layer in the compilation. Usually a layer of parameterized single-qudit gates.

Parameters
qudits – The number of qudits in this circuit.

Returns
A single Gate representing an initial layer for the circuit

Return type
qsearch.gates.Gate

search_layers(*n*)

A set of possible multi-qubit gates for searching. Usually this is a two-qudit gate followed by two single-qudit gates, for every allowed placement of the two-qudit gate. This defines the branching factor of the search tree.

Parameters
qudits – The number of qudits in this circuit

Returns
A list of tuples of (gate,weight) where Gate is the Gate representing that possible placement of the two-qudit gate, and weight is the weight or cost of adding that gate in that placement to the final circuit.

Return type
list

class qsearch.gatesets.**QubitCNOTLinear**(*single_gate=U3Gate()*, *single_alt=XZXZGate()*)

Bases: *Gateset*

A Gateset for working with CNOT and single-qubit gates parameterized with U3Gate and XZXZGate on the linear topology. This Gateset covers the same search space but uses fewer parameters than ZXZXZCNOTLinear and U3CNOTLinear.

Parameters

- **single_gate** – A qsearch.gates.Gate object used as the single-qubit gate placed after the target side of a CNOT.
- **single_alt** – A qsearch.gates.Gate object used as the single-qubit gate placed after the control side of a CNOT.

Gatesets must set the value of d in their initializer, which represents the size of qudits that are supported (e.g. 2 for qubits or 3 for qutrits).

initial_layer(*n*)

The initial layer in the compilation. Usually a layer of parameterized single-qudit gates.

Parameters
qudits – The number of qudits in this circuit.

Returns
A single Gate representing an initial layer for the circuit

Return type
qsearch.gates.Gate

search_layers(*n*)

A set of possible multi-qubit gates for searching. Usually this is a two-qudit gate followed by two single-qudit gates, for every allowed placement of the two-qudit gate. This defines the branching factor of the search tree.

Parameters
qudits – The number of qudits in this circuit

Returns
A list of tuples of (gate,weight) where Gate is the Gate representing that possible placement of the two-qudit gate, and weight is the weight or cost of adding that gate in that placement to the final circuit.

Return type
list

`branching_factor`(*qudits*)

Returns an integer indicating the expected branching factor. Usually this is automatically determined from search_layers, but it may need to be overridden if successors is overridden.

Parameters
qudits – The number of qudits in this circuit

Returns
An integer indicating the expecte branching factor

Return type
int

`successors`(*circ*, *qudits=None*)

Returns a list of Gates that are successors in the search tree to the input Gate, circ, representing a current ansatz circuit.

Parameters

- **circ** – The curret ansatz Gate.
- **qudits** – The number of qudits in this circuit.

Returns
A list of tuples of (gate, weight) where gate is a Gate that is a successor to circ, and weight is the cost or weight of moving to gate from circ.

Return type
list

`class qsearch.gatesets.QubitCNOTRing`(*single_gate=U3Gate()*, *single_alt=XZXZGate()*)

Bases: *Gateset*

A Gateset for working with CNOT and single-qubit gates parameterized with U3Gate and XZXZGate on the ring topology. :param single_gate: A qsearch.gates.Gate object used as the single-qubit gate placed after the target side of a CNOT. :param single_alt: A qsearch.gates.Gate object used as the single-qubit gate placed after the control side of a CNOT.

Gatesets must set the value of d in their initializer, which represents the size of qudits that are supported (e.g. 2 for qubits or 3 for qutrits).

`initial_layer`(*n*)

The initial layer in the compilation. Usually a layer of parameterized single-qudit gates.

Parameters
qudits – The number of qudits in this circuit.

Returns
A single Gate representing an initial layer for the circuit

Return type
qsearch.gates.Gate

search_layers(*n*)

A set of possible multi-qubit gates for searching. Usually this is a two-qudit gate followed by two single-qudit gates, for every allowed placement of the two-qudit gate. This defines the branching factor of the search tree.

Parameters
qudits – The number of qudits in this circuit

Returns
A list of tuples of (gate,weight) where Gate is the Gate representing that possible placement of the two-qudit gate, and weight is the weight or cost of adding that gate in that placement to the final circuit.

Return type
list

class qsearch.gatesets.**QubitCZLinear**

Bases: *Gateset*

A Gateset for working with CZ and single-qubit gates parameterized with U3Gate and XZXZGate on the linear topology.

Gatesets must set the value of d in their initializer, which represents the size of qudits that are supported (e.g. 2 for qubits or 3 for qutrits).

initial_layer(*n*)

The initial layer in the compilation. Usually a layer of parameterized single-qudit gates.

Parameters
qudits – The number of qudits in this circuit.

Returns
A single Gate representing an initial layer for the circuit

Return type
qsearch.gates.Gate

search_layers(*n*)

A set of possible multi-qubit gates for searching. Usually this is a two-qudit gate followed by two single-qudit gates, for every allowed placement of the two-qudit gate. This defines the branching factor of the search tree.

Parameters
qudits – The number of qudits in this circuit

Returns
A list of tuples of (gate,weight) where Gate is the Gate representing that possible placement of the two-qudit gate, and weight is the weight or cost of adding that gate in that placement to the final circuit.

Return type
list

branching_factor(*qudits*)

Returns an integer indicating the expected branching factor. Usually this is automatically determined from search_layers, but it may need to be overridden if successors is overridden.

Parameters
qudits – The number of qudits in this circuit

Returns
An integer indicating the expecte branching factor

Return type
int

`successors`(*circ*, *qudits=None*)

Returns a list of Gates that are successors in the search tree to the input Gate, circ, representing a current ansatz circuit.

Parameters

- **circ** – The curret ansatz Gate.
- **qudits** – The number of qudits in this circuit.

Returns
A list of tuples of (gate, weight) where gate is a Gate that is a successor to circ, and weight is the cost or weight of moving to gate from circ.

Return type
list

`class qsearch.gatesets.QubitISwapLinear`

Bases: *Gateset*

A Gateset for working with ISwap and single-qubit gates parameterized with U3Gate and XZXZGate on the linear topology.

Gatesets must set the value of d in their initializer, which represents the size of qudits that are supported (e.g. 2 for qubits or 3 for qutrits).

`initial_layer`(*n*)

The initial layer in the compilation. Usually a layer of parameterized single-qudit gates.

Parameters
qudits – The number of qudits in this circuit.

Returns
A single Gate representing an initial layer for the circuit

Return type
qsearch.gates.Gate

`search_layers`(*n*)

A set of possible multi-qubit gates for searching. Usually this is a two-qudit gate followed by two single-qudit gates, for every allowed placement of the two-qudit gate. This defines the branching factor of the search tree.

Parameters
qudits – The number of qudits in this circuit

Returns
A list of tuples of (gate,weight) where Gate is the Gate representing that possible placement of the two-qudit gate, and weight is the weight or cost of adding that gate in that placement to the final circuit.

Return type
list

branching_factor(*qudits*)

Returns an integer indicating the expected branching factor. Usually this is automatically determined from search_layers, but it may need to be overridden if successors is overridden.

Parameters
qudits – The number of qudits in this circuit

Returns
An integer indicating the expecte branching factor

Return type
int

successors(*circ*, *qudits=None*)

Returns a list of Gates that are successors in the search tree to the input Gate, circ, representing a current ansatz circuit.

Parameters

- **circ** – The curret ansatz Gate.
- **qudits** – The number of qudits in this circuit.

Returns
A list of tuples of (gate, weight) where gate is a Gate that is a successor to circ, and weight is the cost or weight of moving to gate from circ.

Return type
list

class qsearch.gatesets.**QubitXXLinear**

Bases: *Gateset*

A Gateset for working with ISwap and single-qubit gates parameterized with U3Gate and XZXZGate on the linear topology.

Gatesets must set the value of d in their initializer, which represents the size of qudits that are supported (e.g. 2 for qubits or 3 for qutrits).

initial_layer(*n*)

The initial layer in the compilation. Usually a layer of parameterized single-qudit gates.

Parameters
qudits – The number of qudits in this circuit.

Returns
A single Gate representing an initial layer for the circuit

Return type
qsearch.gates.Gate

search_layers(*n*)

A set of possible multi-qubit gates for searching. Usually this is a two-qudit gate followed by two single-qudit gates, for every allowed placement of the two-qudit gate. This defines the branching factor of the search tree.

Parameters
qudits – The number of qudits in this circuit

Returns
A list of tuples of (gate,weight) where Gate is the Gate representing that possible placement

of the two-qudit gate, and weight is the weight or cost of adding that gate in that placement to the final circuit.

Return type
list

`branching_factor`(*qudits*)

Returns an integer indicating the expected branching factor. Usually this is automatically determined from search_layers, but it may need to be overridden if successors is overridden.

Parameters
qudits – The number of qudits in this circuit

Returns
An integer indicating the expecte branching factor

Return type
int

`successors`(*circ*, *qudits=None*)

Returns a list of Gates that are successors in the search tree to the input Gate, circ, representing a current ansatz circuit.

Parameters

- **circ** – The curret ansatz Gate.
- **qudits** – The number of qudits in this circuit.

Returns
A list of tuples of (gate, weight) where gate is a Gate that is a successor to circ, and weight is the cost or weight of moving to gate from circ.

Return type
list

`class qsearch.gatesets.QubitCNOTAdjacencyList`(*adjacency*, *single_gate=U3Gate()*, *single_alt=XZXZGate()*)

Bases: *Gateset*

A Gateset for working with CNOT and single-qubit gates parameterized with U3Gate and XZXZGate on a custom topology, specified in the initializer.

Allows the specification of a custom topology through an adjacency list.

For example, this is how you would specifiy the ring topology for 3 qubits: *[(0,1), (1,2), (2,1)]*

It is not recommended to add bi-directional links, because with the arbitrary parameterized single qubit gates everywhere, such links would be redundant.

Parameters

- **adjacency** – A list of tuples specifying which CNOT placements are allowed. The tuples must be in the form of (control, target).
- **single_gate** – A qsearch.gates.Gate object used as the single-qubit gate placed after the target side of a CNOT.
- **single_alt** – A qsearch.gates.Gate object used as the single-qubit gate placed after the control side of a CNOT.

`initial_layer`(*n*)

The initial layer in the compilation. Usually a layer of parameterized single-qudit gates.

Parameters
qudits – The number of qudits in this circuit.

Returns
A single Gate representing an initial layer for the circuit

Return type
qsearch.gates.Gate

`search_layers(n)`

A set of possible multi-qubit gates for searching. Usually this is a two-qudit gate followed by two single-qudit gates, for every allowed placement of the two-qudit gate. This defines the branching factor of the search tree.

Parameters
qudits – The number of qudits in this circuit

Returns
A list of tuples of (gate,weight) where Gate is the Gate representing that possible placement of the two-qudit gate, and weight is the weight or cost of adding that gate in that placement to the final circuit.

Return type
list

class qsearch.gatesets.**QutritCPIPhaseLinear**

Bases: *Gateset*

A Gateset for working with CPIPhase and single-qutrit gates on the linear topology.

Gatesets must set the value of d in their initializer, which represents the size of qudits that are supported (e.g. 2 for qubits or 3 for qutrits).

`initial_layer(n)`

The initial layer in the compilation. Usually a layer of parameterized single-qudit gates.

Parameters
qudits – The number of qudits in this circuit.

Returns
A single Gate representing an initial layer for the circuit

Return type
qsearch.gates.Gate

`search_layers(n)`

A set of possible multi-qubit gates for searching. Usually this is a two-qudit gate followed by two single-qudit gates, for every allowed placement of the two-qudit gate. This defines the branching factor of the search tree.

Parameters
qudits – The number of qudits in this circuit

Returns
A list of tuples of (gate,weight) where Gate is the Gate representing that possible placement of the two-qudit gate, and weight is the weight or cost of adding that gate in that placement to the final circuit.

Return type
list

class qsearch.gatesets.**QutritCNOTLinear**

Bases: *Gateset*

A hybrid Gateset for working with CNOT and single-qutrit gates on the linear topology.

Gatesets must set the value of d in their initializer, which represents the size of qudits that are supported (e.g. 2 for qubits or 3 for qutrits).

initial_layer(*n*)

The initial layer in the compilation. Usually a layer of parameterized single-qudit gates.

Parameters
qudits – The number of qudits in this circuit.

Returns
A single Gate representing an initial layer for the circuit

Return type
qsearch.gates.Gate

search_layers(*n*)

A set of possible multi-qubit gates for searching. Usually this is a two-qudit gate followed by two single-qudit gates, for every allowed placement of the two-qudit gate. This defines the branching factor of the search tree.

Parameters
qudits – The number of qudits in this circuit

Returns
A list of tuples of (gate,weight) where Gate is the Gate representing that possible placement of the two-qudit gate, and weight is the weight or cost of adding that gate in that placement to the final circuit.

Return type
list

qsearch.gatesets.**linear_topology**(*double_gate*, *single_gate*, *n*, *d*, *identity_gate=None*, *single_alt=None*, *double_weight=1*, *single_weight=0*, *skip_index=None*)

qsearch.gatesets.**fill_row**(*gate*, *n*)

qsearch.gatesets.**find_last_3_cnots_linear**(*circuit*)

qsearch.gatesets.**DefaultQubit**

qsearch.gatesets.**DefaultQutrit**

qsearch.gatesets.**Default**

qsearch.heuristics

The functions in this module are used as heuristics to guide the search in SearchCompiler.

The required format for a heuristic is to take in a circuit, a vector of parameters for that circuit, a weight for that circuit, and an Options object, and to return a single real valued number that will be used to order the search tree.

Module Contents

Functions

greedy(circ, v, weight, options)	Defines a heuristic that results in greedy search, which focuses soley on minimizing the eval_func, and behaves somewhat similarly to depth first sarch.
astar(circ, v, weight, options)	Defines a heuristic that combines the weight of the circuit with the value from eval_func. It generally gives similar quality results to djikstra, but with a drastic reduction in the number of node evaluations.
djikstra(circ, v, weight, options)	Defines a heuristic that relies only on the weight, which gurantees a minimal-weight final solution, at the expense of a long runtime. It behaves somewhat similarly to breadth first search.

qsearch.heuristics.**greedy**(*circ*, *v*, *weight*, *options*)

Defines a heuristic that results in greedy search, which focuses soley on minimizing the eval_func, and behaves somewhat similarly to depth first sarch.

qsearch.heuristics.**astar**(*circ*, *v*, *weight*, *options*)

Defines a heuristic that combines the weight of the circuit with the value from eval_func. It generally gives similar quality results to djikstra, but with a drastic reduction in the number of node evaluations.

qsearch.heuristics.**djikstra**(*circ*, *v*, *weight*, *options*)

Defines a heuristic that relies only on the weight, which gurantees a minimal-weight final solution, at the expense of a long runtime. It behaves somewhat similarly to breadth first search.

qsearch.integrations

Module Contents

Classes

QiskitGateConverter

Functions

qiskit_to_qsearch(circ[, converter])	Convert qiskit code to qsearch *structure* and parameters

Attributes

qiskit	

qsearch.integrations.**qiskit**

exception qsearch.integrations.**QiskitImportError**

Bases: Exception

A class to represent issues importing code from qiskit

class qsearch.integrations.**QiskitGateConverter**(*num_qubits*)

convert(*gate*, *qubits*, *cbits*)

Abstraction to convert an arbitrary qiskit gate to a layer in a qsearch circuit

convert_cx(*gate*, *qubits*, *cbits*)

convert_u3(*gate*, *qubits*, *cbits*)

convert_u2(*gate*, *qubits*, *cbits*)

convert_rx(*gate*, *qubits*, *cbits*)

convert_ry(*gate*, *qubits*, *cbits*)

convert_rz(*gate*, *qubits*, *cbits*)

qsearch.integrations.**qiskit_to_qsearch**(*circ*, *converter=None*)

Convert qiskit code to qsearch *structure* and parameters

qsearch.leap_compiler

This module provides LeapCompiler, which is a more scalable variant of SearchCompiler, at the expense of producing somewhat longer circuits. LeapReoptimizing_PostProcessor can be used to reduce circuit length back to levels that SearchCompiler might generate.

Module Contents

Classes

LeapCompiler	LeapCompiler is a more scalable search based circuit compiler
SubCompiler	A modified SearchCompiler for the LeapCompiler to use.

Functions

cut_end(circ, depth)

qsearch.leap_compiler.**cut_end**(*circ*, *depth*)

class qsearch.leap_compiler.**LeapCompiler**(*options=Options()*)

Bases: *qsearch.compiler.Compiler*

LeapCompiler is a more scalable search based circuit compiler

LeapCompiler uses fixed structure prefixes to greatly reduce the search space and speed up synthesis at the cost of optimiality. Thus it is recommended to use in conjunction with reoptimizing_compiler.LeapReoptimizing_PostProcessor() to obtain the best results.

Options:

target (required) : The unitary matrix to be synthesized, in the form of a numpy ndarray with dtype="complex128". gateset : The Gateset used for synthesis. weight_limit : A limit on the maximum weight for circuits to be expanded for further searching. See gatesets.py for more information. The default is None for unlimited. heuristic : A heuristic used to order the search tree. See heuristics.py for more information. solver : A Solver used for optimizing the parameters in parameterized circuits generated by the search tree. parallelizer : A Parallelizer used for solving multiple parameterized circuits in parallel. beams : The number of nodes to pop from the search tree at a time. The default value of -1 will create enough branches to maximize utilization of your CPU. error_func : The function that the Solver will attempt to minimize. eval_func : The function used by the heuristic in order to guide the search tree. By default this is equal to error_func. error_jac : A function that returns a tuple of the value that error_func would generate and the jacobian of error_func error_residuals : A function that returns an array of real-valued residuals to be used by a least-squares-based Solver. error_residuals_jac : A function that returns the jacobian of error_residuals (note that it does NOT return a tuple of the residuals and the jacobian). timeout : An uper limit on the amount of time the compiler will spend trying to synthesize a circuit. The default is float('inf'), for unlimited. checkpoint : The compiler will use this Checkpoint to save intermediate state, and will resume from this Checkpoint if there was an existing state. logger : A qsearch.logging.Logger that will be used for logging the synthesis process. min_depth : the minimum amount of searching

Run LEAP on the compilation specified in options.

Parameters

options – options for the compilations, see the class level documentation for details.

compile(*options=Options()*)

Run LEAP on the compilation specified in options.

Parameters

options – options for the compilations, see the class level documentation for details.

class qsearch.leap_compiler.**SubCompiler**(*options=Options()*)

Bases: *qsearch.compiler.Compiler*

A modified SearchCompiler for the LeapCompiler to use.

compile(*options=Options()*)

qsearch.logging

This module defines the Logger class, which is used to control and automate the logging of messages to stdout and to files.

Module Contents

Classes

Logger	This class is used to control what level of mesages get printed, and to where.

class qsearch.logging.**Logger**(*stdout_enabled=False*, *output_file=None*, *verbosity=1*)

This class is used to control what level of mesages get printed, and to where.

logprint(*string*, *verbosity=1*)

This function logs the specified string according to the specified options.

qsearch.multistart_solvers

This module defines solvers that use multiple starting points in order to have a higher chance at finding the global minimum.

Module Contents

Classes

MultiStart_Solver	A higher accuracy solver based on APOSMM https://www.mcs.anl.gov/~jlarson/APOSMM/
NaiveMultiStart_Solver	A naive but effective multi-start solver which tries to cover as much of the optimization space at once

Functions

distance_for_x(x, options, circuit)	Calculate the distance between circuit and the target for input x based on the distance metric
optimize_worker(circuit, options, q, x0, error_func)	Worker function used to run the inner solver in parallel

qsearch.multistart_solvers.**distance_for_x**(*x*, *options*, *circuit*)

Calculate the distance between circuit and the target for input x based on the distance metric

qsearch.multistart_solvers.**optimize_worker**(*circuit*, *options*, *q*, *x0*, *error_func*)

Worker function used to run the inner solver in parallel

class qsearch.multistart_solvers.**MultiStart_Solver**(*num_threads*)

Bases: *qsearch.solvers.Solver*

A higher accuracy solver based on APOSMM https://www.mcs.anl.gov/~jlarson/APOSMM/

MultiStart_Solver generally gets better results than other optimizers due to the advanced algorithm to start multiple local optimizers ("inner solvers") and find the global optimum more often.

Create a MultiStart_Solver instance. Pass num_threads to set how many threads to use in parallel optimization runs

solve_for_unitary(*circuit*, *options*, *x0=None*)

Optimize the given circuit based on the provided options with initial point x0 (optional).

Parameters

- **circuit** – A qsearch.gates.Gate describing the circuit to optimize
- **options** – This uses the following options: - inner_solver : which optimizer to use for local optimization runs - target : the target unitary of synthesis - logger : A qsearch.logging.Logger that will be used for logging the synthesis process. - error_func : The function that the Solver will attempt to minimize. - error_residuals : A function that returns an array of real-valued residuals to be used by a least-squares-based Solver.
- **x0** – the starting point for the optimzier

class qsearch.multistart_solvers.**NaiveMultiStart_Solver**(*num_threads*)

Bases: *qsearch.solvers.Solver*

A naive but effective multi-start solver which tries to cover as much of the optimization space at once

Create a NaiveMultiStart_Solver instance. Pass num_threads to set how many threads to use in parallel optimization runs

solve_for_unitary(*circuit*, *options*, *x0=None*)

Finds the best parameters that minimize error_func or error_residuals between the unitary from the circuit and options.target.

qsearch.objectives

Module Contents

Classes

Objective
MatrixDistanceObjective
StateprepObjective
BackwardsCompatibleObjective

class qsearch.objectives.**Objective**

gen_eval_func(*circuit*, *options*)

gen_error_func(*circuit*, *options*)

gen_error_jac(*circuit*, *options*)

gen_error_residuals(*circuit*, *options*)

gen_error_residuals_jac(*circuit*, *options*)

class qsearch.objectives.**MatrixDistanceObjective**

Bases: *Objective*

gen_error_func(*circuit*, *options*)

gen_error_jac(*circuit*, *options*)

gen_error_residuals(*circuit*, *options*)

gen_error_residuals_jac(*circuit*, *options*)

class qsearch.objectives.**StateprepObjective**

Bases: *Objective*

gen_error_func(*circuit*, *options*)

gen_error_jac(*circuit*, *options*)

gen_error_residuals(*circuit*, *options*)

gen_error_residuals_jac(*circuit*, *options*)

class qsearch.objectives.**BackwardsCompatibleObjective**

Bases: *Objective*

gen_error_func(*circuit*, *options*)

gen_error_func(*circuit*, *options*)

gen_error_jac(*circuit*, *options*)

gen_error_residuals(*circuit*, *options*)

gen_error_residuals_jac(*circuit*, *options*)

qsearch.options

A class for holding and managing options that are passed to various other classes in the Qsearch suite.

Options objects work like dictionaries, but if an Options object is queried for an item and it does not have it, it first checks its defaults and smart_defaults properties for the item before throwing an error. This allows the setting of default values which are easily overridden by user-provided values. The smart_defaults dictionary contains functions that return an object, allowing for default behavior that is dependent on other settings within the Options object.

Options objects are also designed to be easily combinable through functions such as update and updated.

Options objects are used ubiquitously throughout Qsearch

Module Contents

Classes

Options	This class manages options that are passed between various Qsearch objects.

Attributes

_options_actual_parameters

qsearch.options.**_options_actual_parameters = ['defaults', 'smart_defaults', 'required', 'cache', 'load_error']**

class qsearch.options.**Options**(*defaults={}*, *smart_defaults={}*, ***xtraargs*)

This class manages options that are passed between various Qsearch objects.

filtered(**names*)

Returns an Options object with only parameters in the specified list names.

__getitem__(*name*)

__delitem__(*name*)

__getattr__(*name*)

__setattr__(*name*, *value*)

Implement setattr(self, name, value).

__contains__(*name*)

manually_entered(**names*, *location='dict'*, *operator='all'*)

empty_copy()

Create an Options object with the same defaults but without any specific values.

copy()

Create a full copy of an Options object.

__copy__()

updated(*other=None*, ***xtraargs*)

Return a new Options object that is a copy of this object, updated with the contents of other and xtraargs.

update(*other=None*, ***xtraargs*)

Mutate the current Options object with the contents of other and xtraargs.

_update_dict(*otherdict*)

set_defaults(***args*)

Set default values for this Options object.

If an Options object is queried for a value, and it does not contain it, it will check its defaults list before throwing an error.

set_smart_defaults(***args*)

Set smart_defaults values for this Options object.

If an Options object is queried for a value, and it does not contain it, it will check its smart_defaults list before throwing an error. If it does find a function in smart_defaults, it calls that function, passing itself as the argument, and returns the return value of that function, caching it for next time.

make_required(**names*)

Marking names as required will cause the Options object to throw an error if it does not contain it, even if it has defaults or smart_defaults defined.

remove_defaults(**names*)

Removes the defaults for the specified names.

remove_smart_defaults(**names*)

Removes the smart_defaults for the specified names.

generate_cache()

Caches valuesa for all functions in smart_defaults.

save(*filepath=None*)

Saves the Options object to a file, or to a returned tuple.

load(*filepath_or_tuple*, *strict=False*)

Loads the Options object from a file or tuple.

If strict is left as False, the Options object will attempt to gracefully handle errors when loading its contents, relying on its ability to fallback to defaults or smart_defaults if those are able to load successfully.

If strict is set to True, the Options object will throw an error upon any error while loading.

__getstate__()

__setstate__(*state*)

qsearch.parallelizers

This module defines Parallelizer, which is a class that defines how to perform multiple circuits in parallel.

Several implementations are provided.

qsearch.parallelizers.**LokyParallelizer**

A Parallelizer based on Loky, a "deadlock-free" ProcessPoolExecutor

qsearch.parallelizers.**MultiprocessingParallelizer**

A Parallelizer based on multiprocessing

qsearch.parallelizers.**ProcessPoolParallelizer**

A Parallelizer based on concurrent.futures.ProcessPoolExecutor

qsearch.parallelizers.**MPIParallelizer**

A distributed MPI based Parallelizer

qsearch.parallelizers.**SequentialParallelizer**

Mostly for debugging purposes, a Parallelizer that runs tasks one at a time.

Module Contents

Classes

Parallelizer	Base class for all Parallelizers. Parallelizers calculate the value of multiple search nodes in parallel.
LokyParallelizer	A parallelizer based on Loky, a "deadlock-free" ProcessPoolExecutor.
MultiprocessingParallelizer	A Parallelizer based on muliprocessing. Note this cannot be used with the MultiStart_Solvers!
ProcessPoolParallelizer	A Parallelizer based on concurrent.futures.ProcessPoolExecutor.
MPIParallelizer	A distributed MPI based Parallelizer.
SequentialParallelizer	A Paralleizer that isn't, it runs tasks one at a time (mostly for debugging).

Functions

default_num_tasks(options)	
evaluate_step(tup, options)	
single_task(opts)	
process_initializer()	

Attributes

MPI	
get_reusable_executor	

qsearch.parallelizers.**MPI**

qsearch.parallelizers.**get_reusable_executor**

qsearch.parallelizers.**default_num_tasks**(*options*)

qsearch.parallelizers.**evaluate_step**(*tup*, *options*)

qsearch.parallelizers.**single_task**(*opts*)

qsearch.parallelizers.**process_initializer**()

class qsearch.parallelizers.**Parallelizer**

Base class for all Parallelizers. Parallelizers calculate the value of multiple search nodes in parallel.

solve_circuits_parallel(*tuples*)

Calculate the value of search tree nodes in parallel.

done()

Finalize/Clean up any state needed to run the Parallelizer.

class qsearch.parallelizers.**LokyParallelizer**(*options*)

Bases: *Parallelizer*

A parallelizer based on Loky, a "deadlock-free" ProcessPoolExecutor.

For more information on Loky see https://loky.readthedocs.io/en/stable/.

solve_circuits_parallel(*tuples*)

Calculate the value of search tree nodes in parallel.

class qsearch.parallelizers.**MultiprocessingParallelizer**(*options*)

Bases: *Parallelizer*

A Parallelizer based on muliprocessing. Note this cannot be used with the MultiStart_Solvers!

solve_circuits_parallel(*tuples*)

Calculate the value of search tree nodes in parallel.

done()

Finalize/Clean up any state needed to run the Parallelizer.

class qsearch.parallelizers.**ProcessPoolParallelizer**(*options*)

Bases: *Parallelizer*

A Parallelizer based on concurrent.futures.ProcessPoolExecutor.

solve_circuits_parallel(*tuples*)

Calculate the value of search tree nodes in parallel.

done()

Finalize/Clean up any state needed to run the Parallelizer.

class qsearch.parallelizers.**MPIParallelizer**(*options*)

Bases: *Parallelizer*

A distributed MPI based Parallelizer.

This implementation unfortunately requires some work on the part of the Project API or the user.

solve_circuits_parallel(*tuples*)

Calculate the value of search tree nodes in parallel.

map_steps(*new_steps*)

done()

Finalize/Clean up any state needed to run the Parallelizer.

class qsearch.parallelizers.**SequentialParallelizer**(*options*)

Bases: *Parallelizer*

A Paralleizer that isn't, it runs tasks one at a time (mostly for debugging).

solve_circuits_parallel(*tuples*)

Calculate the value of search tree nodes in parallel.

qsearch.persistent_aposmm

This module contains methods used our implementation of the Asynchronously Parallel Optimization Solver for finding Multiple Minima (APOSMM) method. https://doi.org/10.1007/s12532-017-0131-4

This implementation of APOSMM was developed by Kaushik Kulkarni and Jeffrey Larson in the summer of 2019.

Module Contents

Functions

aposmm(H, persis_info, gen_specs, libE_info)	APOSMM coordinates multiple local optimization runs, starting from points
update_history_dist(H, n)	Updates distances/indices after new points that have been evaluated.
decide_where_to_start_localopt(H, n, n_s, rk_const[, ...])	APOSMM starts a local optimization runs from a point that:
initialize_APOSMM(H, user_specs, libE_info)	Computes common values every time that APOSMM is reinvoked

qsearch.persistent_aposmm.**aposmm**(*H*, *persis_info*, *gen_specs*, *libE_info*)

APOSMM coordinates multiple local optimization runs, starting from points which do not have a better point nearby (within a distance `r_k`). This generation function uses a `local_H` (serving a similar purpose as `H` in libEnsemble) containing the fields:

- `'x' [n floats]`: Parameters being optimized over
- `'x_on_cube' [n floats]`: Parameters scaled to the unit cube
- `'f' [float]`: Objective function being minimized
- `'local_pt' [bool]`: True if point from a local optimization run
- `'dist_to_unit_bounds' [float]`: Distance to domain boundary
- `'dist_to_better_l' [float]`: Dist to closest better local opt point
- `'dist_to_better_s' [float]`: Dist to closest better sample point
- `'ind_of_better_l' [int]`: Index of point `'dist_to_better_l'` away
- `'ind_of_better_s' [int]`: Index of point `'dist_to_better_s'` away
- `'started_run' [bool]`: True if point has started a local opt run
- `'num_active_runs' [int]`: Number of active local runs point is in
- `'local_min' [float]`: True if point has been ruled a local minima
- `'sim_id' [int]`: Row number of entry in history

and optionally

- `'fvec' [m floats]`: All objective components (if performing a least-squares calculation)
- `'grad' [n floats]`: The gradient (if available) of the objective with respect to x.

Note: - If any of the above fields are desired after a libEnsemble run, name

them in `gen_specs['out']`.

- If intitializing APOSMM with past function values, make sure to include `'x'`, `'x_on_cube'`, `'f'`, `'local_pt'`, etc. in `gen_specs['in']` (and, of course, include them in the H0 array given to libensemble).

Necessary quantities in `gen_specs['user']` are:

- `'lb' [n floats]`: Lower bound on search domain
- `'ub' [n floats]`: Upper bound on search domain
- `'localopt_method' [str]`: Name of an NLopt, PETSc/TAO, or SciPy method (see 'advance_local_run' below for supported methods)
- `'initial_sample_size' [int]`: Number of uniformly sampled points must be returned (non-nan value) before a local opt run is started. Can be zero if no additional sampling is desired, but if zero there must be past sim_f values given to libEnsemble in H0.

Optional `gen_specs['user']` entries are:

- `'sample_points' [numpy array]`: Points to be sampled (original domain). If more sample points are needed by APOSMM during the course of the optimization, points will be drawn uniformly over the domain
- `'components' [int]`: Number of objective components
- `'dist_to_bound_multiple' [float in (0,1]]`: What fraction of the distance to the nearest boundary should the initial step size be in localopt runs
- `'lhs_divisions' [int]`: Number of Latin hypercube sampling partitions (0 or 1 results in uniform sampling)
- `'mu' [float]`: Distance from the boundary that all localopt starting points must satisfy
- `'nu' [float]`: Distance from identified minima that all starting points must satisfy
- `'rk_const' [float]`: Multiplier in front of the r_k value
- `'max_active_runs' [int]`: Bound on number of runs APOSMM is advancing

If the rules in `decide_where_to_start_localopt` produces more than `'max_active_runs'` in some iteration, then existing runs are prioritized.

And `gen_specs['user']` must also contain fields for the given localopt_method's convergence tolerances (e.g., gatol/grtol for PETSC/TAO or ftol_rel for NLopt)

See also:

test_persistent_aposmm_scipy for basic APOSMM usage.

See also:

test_persistent_aposmm_with_grad for an example where past function values are given to libEnsemble/APOSMM.

qsearch.persistent_aposmm.**update_history_dist**(*H*, *n*)

Updates distances/indices after new points that have been evaluated.

See also:

start_persistent_local_opt_gens.py

qsearch.persistent_aposmm.**decide_where_to_start_localopt**(*H*, *n*, *n_s*, *rk_const*, *ld=0*, *mu=0*, *nu=0*)

APOSMM starts a local optimization runs from a point that:

- is not in an active local optimization run,
- is more than `mu` from the boundary (in the unit-cube domain),
- is more than `nu` from identified minima (in the unit-cube domain),
- does not have a better point within a distance `r_k` of it.

For further details, see the conditions (S1-S5 and L1-L8) in Table 1 of the APOSMM paper This method first identifies sample points satisfying S2-S5, and then identifies all localopt points that satisfy L1-L7. We then start from any sample point also satisfying S1. We do not check condition L8 currently.

We don't consider points in the history that have not returned from computation, or that have a `nan` value. As APOSMM works on the unit cube, note that `mu` and `nu` implicitly depend on the scaling of the original domain: adjusting the initial domain can make a run start (or not start) at a point that didn't (or did) previously.

Parameters

- **H** (*numpy structured array*) – History array storing rows for each point.
- **n** (*int*) – Problem dimension
- **n_s** (*int*) – Number of sample points in H
- **r_k_const** (*float*) – Radius for deciding when to start runs
- **ld** (*integer*) – Number of Latin hypercube sampling divisions (0 or 1 means uniform random sampling over the domain)
- **mu** (*nonnegative float*) – Distance from the boundary that all starting points must satisfy
- **nu** (*nonnegative float*) – Distance from identified minima that all starting points must satisfy

Returns

start_inds – Indices where a local opt run should be started, sorted by increasing function value.

Return type

list

See also:

start_persistent_local_opt_gens.py

qsearch.persistent_aposmm.**initialize_APOSMM**(*H*, *user_specs*, *libE_info*)

Computes common values every time that APOSMM is reinvoked

See also:

start_persistent_local_opt_gens.py

qsearch.post_processing

This module defines PostProcessor, a class used to modify circuits after they have been synthesized.

Several implementations are provided.

qsearch.post_processing.**BasicSingleQubitReduction_PostProcessor**

Attempts to remove single-qubit gates without sacrificing the quality of the solution in terms of eval_func

qsearch.post_processing.**ParameterTuning_PostProcessor**

Attempts to reduce eval_func simply by re-running the solver with stronger parameters.

qsearch.post_processing.**LEAPReoptimizing_PostProcessor**

Reduces the length of circuits produced using LEAP by re-running segments of the circuit.

Module Contents

Classes

PostProcessor	This class is used to modify circuits that have already been synthesized.
BasicSingleQubitReduction_PostProcessor	Attempts to reduce the number of single-qubit gates in a circuit by sequentially removing a gate, attempting to use a Solver on it, and keeping that gate removed if successful.
ParameterTuning_PostProcessor	Attempts to reduce the eval_func value of the circuit simply by tuning the parameters better using stronger Solver parameters.
LEAPReoptimizing_PostProcessor	A PostProcessor that re-optimizes LeapCompiler-compiled circuits via search.

class qsearch.post_processing.**PostProcessor**(*options=opt.Options()*)

This class is used to modify circuits that have already been synthesized.

post_process_circuit(*result*, *options=None*)

Processes the circuit dictionary and returns a new one.

Parameters

result – A dictionary containing a synthesized circuit. Expect it to contain "structure" and "parameters", but it may contain more, depending on what previous PostProcessors were run and on the compiler.

Returns

A dictionary containing any updates that should be made to the circuit dictionary, such as new values for "structure" or "parameters" or arbitrary other data.

Return type

dict

class qsearch.post_processing.**BasicSingleQubitReduction_PostProcessor**(*options=opt.Options()*)

Bases: *PostProcessor*

Attempts to reduce the number of single-qubit gates in a circuit by sequentially removing a gate, attempting to use a Solver on it, and keeping that gate removed if successful.

post_process_circuit(*result*, *options=None*)

Processes the circuit dictionary and returns a new one.

Parameters
result – A dictionary containing a synthesized circuit. Expect it to contain "structure" and "parameters", but it may contain more, depending on what previous PostProcessors were run and on the compiler.

Returns
A dictionary containing any updates that should be made to the circuit dictionary, such as new values for "structure" or "parameters" or arbitrary other data.

Return type
dict

class qsearch.post_processing.**ParameterTuning_PostProcessor**(*options=opt.Options()*)

Bases: *PostProcessor*

Attempts to reduce the eval_func value of the circuit simply by tuning the parameters better using stronger Solver parameters.

post_process_circuit(*result*, *options=None*)

Processes the circuit dictionary and returns a new one.

Parameters
result – A dictionary containing a synthesized circuit. Expect it to contain "structure" and "parameters", but it may contain more, depending on what previous PostProcessors were run and on the compiler.

Returns
A dictionary containing any updates that should be made to the circuit dictionary, such as new values for "structure" or "parameters" or arbitrary other data.

Return type
dict

class qsearch.post_processing.**LEAPReoptimizing_PostProcessor**(*options=Options()*)

Bases: *qsearch.compiler.Compiler*, *PostProcessor*

A PostProcessor that re-optimizes LeapCompiler-compiled circuits via search.

This PostProcessor puts "holes" in the circuit where LEAP fixed prefixes and runs qsearch on those holes to reduce the total number of gates.

post_process_circuit(*result*, *options=None*)

Re-optimize a LEAP circuit. Pass "depth" to indicate the size to re-synthesize. It is recommended to call like: *project.post_process(post_processing.LEAPReoptimizing_PostProcessor(), solver=multistart_solvers.MultiStart_Solver(8), parallelizer=parallelizers.ProcessPoolParallelizer, depth=7)*

compile(*options=Options()*)

Backwards compatible interface since this is technically a Compiler.

You should use LEAPReoptimizing_PostProcessor.post_process_circuit with the Project post_processing API.

qsearch.project

This module provides a wrapper that makes it easier to interface with the rest of Qsearch.

Module Contents

Classes

Project_Status	Generic enumeration.
Project	The project class wraps most of the functionality of Qsearch as intended to help manage working with Qsearch.

Attributes

MPI

qsearch.project.**MPI**

class qsearch.project.**Project_Status**

Bases: enum.Enum

Generic enumeration.

Derive from this class to define new enumerations.

PROGRESS = 1

COMPLETE = 2

NOTBEGUN = 3

class qsearch.project.**Project**(*path*, *use_mpi=False*)

The project class wraps most of the functionality of Qsearch as intended to help manage working with Qsearch.

property compilations

The list of names corresponding to compilations on this Project.

_save()

_checkpoint_path(*name*)

add_compilation(*name*, *U*, *options=None*, *handle_existing=None*, ***extraargs*)

Adds a unitary to be compiled.

Parameters

- **name** – A name for this unitary. Must be unique in this Project.
- **U** – The unitary to be compiled, in the form of a numpy ndarray with dtype="complex128"

- **handle_existing** – A variable which defines how to behave if a compilation with the given name already exists. If it is set to "ignore", it will simply return without doing anything. If it is set to "overwrite", it will overwrite the previous entry. If it is set to the default of None, it will offer a warning asking the user to remove and re-add the compilation.
- **options** – The options passed to this function will be used only when this compilation is run.
- **extraargs** – The extraargs passed to this function will be used only when this compilation is run.

__setitem__(*keyword*, *value*)

configure_compiler_override(*keyword*, *value*)

An unsafe method that allows the user to set global Project Options even if there is existing work.

__getitem__(*keyword*)

__delitem__(*keyword*)

configure(***dictionary*)

Adds multiple options to the global Project Options at once.

reset(*name=None*)

Resets a Project, removing any work done but not the initial configurations.

Parameters
name – Optionally specify a particular compilation by name to reset

remove_compilation(*name*)

Removes a compilation from a Project.

Parameters
name – The name of the compilation to remove

clear(*name=None*)

Clears a Project, reverting it to a state similar to a newly created Project.

Parameters
name – Optionally specify a particular compilation by name to clear

__enter__()

__exit__(*exc_typ*, *exc_val*, *exc_tb*)

set_defaults(*defaults=standard_defaults*)

Updates the Project Options with the standard defaults from defaults.py, or a provided dictionary.

set_smart_defaults(*smart_defaults=standard_smart_defaults*)

Updates the Project Options with the standard smart_defaults from defaults.py, or a provided dictionary

run()

Runs all of the compilations in the Project.

post_process(*postprocessor*, *name=None*, *options=None*, ***xtraargs*)

Post-processes the specified compilation, or all compilations if name is None, using the specified postprocessor.

Parameters

- **postprocessor** – The qsearch.post_processing.PostProcessor to run on the compilation or project
- **name** – Optionally specify a particular compilation by name to reset
- **options** – Options to pass to the qsearch.post_processing.PostProcessor passed in *postprocessor*
- **extraargs** – Extra arguments passed as options to the qsearch.post_processing.PostProcessor passed in *postprocessor*

`complete()`

Returns a True if all compilations in the Project have finished and False otherwise.

`finish()`

Called when done running compilations in order to end MPI tasks.

`status(name=None, logger=None)`

Prints a status update on how much of a Project has finished running.

Parameters
name – Optionally specify which compilation to check the status of

`_compilation_status(name)`

`_overall_status()`

`get_result(name)`

Get the result of a compilation.

Parameters
name – The name of the compilation to get the result dictionary from

Returns
The result dictionary for a finished compilation. Usually this contains the entries "structure", a Gate, and "parameters", an array of real number parameters.

Return type
dict

`get_target(name)`

Get the target unitary of a compilation.

Parameters
name – The name of the compilation to get the target from

Returns
The target unitary of the compilation

Return type
np.ndarray

`get_time(name)`

Get the runtime that it took to run a compilation.

Parameters
name – The name of the compilation to get the runtime of

Returns
The number of seconds the compilation took

Return type
float

get_options(*name=None*)

Get the qsearch.options.Options object from a compilation of project

Parameters
name – Optionally pass the name of the compilation to get the qsearch.options.Options object from

Returns
the requested options object

Return type
qsearch.options.Options

assemble(*name*, *options=None*, ***xtraargs*)

Assembles a compilation using the Assembler specified as assembler in the Options. :param name: The compilation to assemble :param options: Contains the qsearch.assemblers.Assembler to use in assembly

Returns
The resulting assembled code

Return type
str

qsearch.solvers

Defines Solver, a class used to wrap various numerical optimizers for finding parameters such that an ansatz circuit is a solution to a target unitary.

Module Contents

Classes

Solver	This class is used to wrap numerical optimizers for circuit solving.
CMA_Solver	Uses cmaes gradient-free optimization from the cma package.
COBYLA_Solver	Uses cobyla gradient-free optimization from scipy.
DIY_Solver	An easier way to wrap a numerical optimizer than writing your own Solver class.
BFGS_Jac_Solver	A solver based on the BFGS implementation in scipy. It requires gradients.
LeastSquares_Jac_Solver	Uses the Leavenberg-Marquardt least-squares optimizer in scipy.

Functions

default_solver(options[, x0])	Runs a complex list of tests to determine the best Solver for a specific situation.

qsearch.solvers.**default_solver**(*options*, *x0=None*)

Runs a complex list of tests to determine the best Solver for a specific situation.

class qsearch.solvers.**Solver**

This class is used to wrap numerical optimizers for circuit solving.

property distance_metric

abstract solve_for_unitary(*circuit*, *options*, *x0=None*)

Finds the best parameters that minimize error_func or error_residuals between the unitary from the circuit and options.target.

__eq__(*other*)

Return self==value.

class qsearch.solvers.**CMA_Solver**

Bases: *Solver*

Uses cmaes gradient-free optimization from the cma package.

solve_for_unitary(*circuit*, *options*, *x0=None*)

Finds the best parameters that minimize error_func or error_residuals between the unitary from the circuit and options.target.

class qsearch.solvers.**COBYLA_Solver**

Bases: *Solver*

Uses cobyla gradient-free optimization from scipy.

solve_for_unitary(*circuit*, *options*, *x0=None*)

Finds the best parameters that minimize error_func or error_residuals between the unitary from the circuit and options.target.

class qsearch.solvers.**DIY_Solver**(*f*)

Bases: *Solver*

An easier way to wrap a numerical optimizer than writing your own Solver class.

Uses the function f that takes in eval_func and initial_guess and returns the parameters that minimizes eval_func.

solve_for_unitary(*circuit*, *options*, *x0=None*)

Finds the best parameters that minimize error_func or error_residuals between the unitary from the circuit and options.target.

class qsearch.solvers.**BFGS_Jac_Solver**

Bases: *Solver*

A solver based on the BFGS implementation in scipy. It requires gradients.

solve_for_unitary(*circuit*, *options*, *x0=None*)

Finds the best parameters that minimize error_func or error_residuals between the unitary from the circuit and options.target.

class qsearch.solvers.**LeastSquares_Jac_Solver**

Bases: *Solver*

Uses the Leavenberg-Marquardt least-squares optimizer in scipy.

property distance_metric

solve_for_unitary(*circuit*, *options*, *x0=None*)

Finds the best parameters that minimize error_func or error_residuals between the unitary from the circuit and options.target.

qsearch.unitaries

This module contains a list of predefined commonly used constant unitaries, and functions for generating commonly used unitaries.

qsearch.unitaries.**cnot**

Constant *CNOT* unitary

qsearch.unitaries.**sqrt_cnot**

Constant square root of *CNOT* unitary

qsearch.unitaries.**swap**

Constant 2 qubit swap unitary

qsearch.unitaries.**toffoli**

Constant toffoli unitary

qsearch.unitaries.**fredkin**

Constant fredkin unitary

qsearch.unitaries.**peres**

Constant peres unitary

qsearch.unitaries.**logical_or**

Constant logical or unitary

qsearch.unitaries.**full_adder**

Constant adder unitary

qsearch.unitaries.**rot_x**

Function to generate an *X* rotation by *theta*

Parameters
theta (*float*) –

qsearch.unitaries.**rot_x_jac**

Function that returns the jacobian of rot_x()

Parameters
theta (*float*) –

qsearch.unitaries.**rot_y**

Function to generate an *Y* rotation by *theta*

Parameters
theta (*float*) –

qsearch.unitaries.**rot_y_jac**

Function that returns the jacobian of rot_y()

Parameters
theta (*float*) –

qsearch.unitaries.**rot_z**

Function to generate an *Z* rotation by *theta*

Parameters
theta (*float*) –

qsearch.unitaries.**rot_z_jac**

Function that returns the jacobian of rot_z()

Parameters
theta (*float*) –

qsearch.unitaries.**qft**

Returns a *n `x `n* qft matrix.

Parameters
n (*int*) –

qsearch.unitaries.**identity**

Returns a *n `x `n* identity matrix

Parameters
n (*int*) –

qsearch.unitaries.**general_swap**

Returns the swap matrix for qudits of the specified size.

Parameters
d (*int*) –

qsearch.unitaries.**arbitrary_cnot**

Returns a CNOT between any two qubits within the specified number of qubits

Parameters

- **qudits** (*int*) –
- **control** (*int*) –
- **target** (*int*) –

Module Contents

Functions

rot_z(theta)	
rot_z_jac(theta)	
rot_x(theta)	
rot_x_jac(theta)	
rot_y(theta)	
rot_y_jac(theta)	
qft(n)	
identity(n)	
general_swap([d])	
arbitrary_cnot(qudits, control, target)	

Attributes

cnot
sqrt_cnot
swap
toffoli
fredkin
peres
logical_or
full_adder
pauli_x
pauli_y
pauli_z
sqrt_x

qsearch.unitaries.**cnot**

qsearch.unitaries.**sqrt_cnot**

qsearch.unitaries.**swap**

qsearch.unitaries.**toffoli**

qsearch.unitaries.**fredkin**

qsearch.unitaries.**peres**

qsearch.unitaries.**logical_or**

qsearch.unitaries.**full_adder**

qsearch.unitaries.**pauli_x**

qsearch.unitaries.**pauli_y**

qsearch.unitaries.**pauli_z**

qsearch.unitaries.**sqrt_x**

qsearch.unitaries.**rot_z**(*theta*)

Parameters

theta (*float*) –

qsearch.unitaries.**rot_z_jac**(*theta*)

Parameters
theta (*float*) –

qsearch.unitaries.**rot_x**(*theta*)

Parameters
theta (*float*) –

qsearch.unitaries.**rot_x_jac**(*theta*)

Parameters
theta (*float*) –

qsearch.unitaries.**rot_y**(*theta*)

Parameters
theta (*float*) –

qsearch.unitaries.**rot_y_jac**(*theta*)

Parameters
theta (*float*) –

qsearch.unitaries.**qft**(*n*)

Parameters
n (*int*) –

qsearch.unitaries.**identity**(*n*)

Parameters
n (*int*) –

qsearch.unitaries.**general_swap**(*d=2*)

Parameters
d (*int*) –

qsearch.unitaries.**arbitrary_cnot**(*qudits*, *control*, *target*)

Parameters

- **qudits** (*int*) –
- **control** (*int*) –
- **target** (*int*) –

qsearch.utils

This module contains miscellaneous helper functions and tools.

The functions you may want to be aware of:

qsearch.utils.**endian_reverse**
: Reverses the endianness of the specified unitary. Necessary for working with unitaries from Qiskit.

qsearch.utils.**matrix_distance_squared**
: The default error_func. Returns the Hilbert-Schmidt norm between two matrices.

qsearch.utils.**matrix_distance_squared_jac**

Returns the value that matrix_distance_squared would return, as well as the jacobian.

qsearch.utils.**matrix_residuals**

The default error_residuals. Returns residuals based on difference between the poduct of the implemented matrix and the hermitian conjugate of the target and the identitiy.

qsearch.utils.**matrix_residuals_jac**

Returns the jacobian of matrix_residuals. Does not return the value of matrix_residuals as well.

qsearch.utils.**remap**

Remaps a unitary for acting on qudits in a different order.

qsearch.utils.**upgrade_qudits**

Upgrades a unitary from a lower qudit size to a larger qudit size.

Module Contents

Functions

matrix_product(*LU)	Performs matrix multiplication of a list of matrices.
matrix_kron(*LU)	Performs the kronecker product on a list of matrices.
op_norm(A)	An implementation of the l1-l1 operator norm.
nearest_unitary(A)	Calculate the closest unitary to a given matrix.
index_test(i, di, df)	
downgrade_qudits_residuals(di, df, A, B, I)	
downgrade_qudits_residuals_jac(di, df, A, B, J)	
generate_stateprep_target_matrix(state)	
re_rot_z(theta, old_z)	
re_rot_z_jac(theta, old_z[, multiplier])	
q1_unitary(x)	
qt_arb_rot(Theta_1, Theta_2, Theta_3, Phi_1, Phi_2, ...)	Using the parameterization found in https://journals.aps.org/prd/pdf/10.1103/PhysRevD.38.1994,
random_near_identity(n, alpha)	
remap(U, order[, d])	
upgrade_qudits(U[, di, df])	
endian_reverse(U[, d])	
mpi_rank()	
mpi_do_work(comm)	Do the work of a single compilation.
mpi_worker()	Create a worker that will keep running compilation requests until told to stop

Attributes

MPI

qsearch.utils.**MPI**

qsearch.utils.**matrix_product**(**LU*)

Performs matrix multiplication of a list of matrices.

qsearch.utils.**matrix_kron**(**LU*)

Performs the kronecker product on a list of matrices.

qsearch.utils.**op_norm**(*A*)

An implementation of the l1-l1 operator norm.

qsearch.utils.**nearest_unitary**(*A*)

Calculate the closest unitary to a given matrix.

Calculate the unitary matrix U that is closest with respect to the operator norm distance to the general matrix A.

D.M.Reich. "Characterisation and Identification of Unitary Dynamics Maps in Terms of Their Action on Density Matrices"

Parameters
A (*np.ndarray*) – The matrix input.

Returns
The unitary matrix closest to A. Return U as a numpy matrix.

Return type
(np.ndarray)

Thank you to Ed Younis, this is based on code from qfast

qsearch.utils.**index_test**(*i*, *di*, *df*)

qsearch.utils.**downgrade_qudits_residuals**(*di*, *df*, *A*, *B*, *I*)

qsearch.utils.**downgrade_qudits_residuals_jac**(*di*, *df*, *A*, *B*, *J*)

qsearch.utils.**generate_stateprep_target_matrix**(*state*)

qsearch.utils.**re_rot_z**(*theta*, *old_z*)

qsearch.utils.**re_rot_z_jac**(*theta*, *old_z*, *multiplier=1*)

qsearch.utils.**q1_unitary**(*x*)

qsearch.utils.**qt_arb_rot**(*Theta_1*, *Theta_2*, *Theta_3*, *Phi_1*, *Phi_2*, *Phi_3*, *Phi_4*, *Phi_5*)

Using the parameterization found in https://journals.aps.org/prd/pdf/10.1103/PhysRevD.38.1994, this method constructs an arbitrary single_qutrit unitary operation.

Parameters
qutrit_params – a list of eight parameters, in the following order Theta_1, Theta_2, Theta_3, Phi_1, Phi_2, Phi_3, Phi_4, Phi_5 The formula for the matrix is:

u11 = cos[Theta_1]*cos[Theta_2]*exp[i*Phi_1] u12 = sin[Theta_1]*exp[i*Phi_3] u13 = cos[Theta_1]*sin[Theta_2]*exp[i*Phi_4] u21 = sin[Theta_2]*sin[Theta_3]*exp[-i*Phi_4 - i*Phi_5] -

sin[Theta_1]*cos[Theta_2]*cos[Theta_3]*exp[i*Phi_1+i*Phi_2-i*Phi_3]

u22 = cos[Theta_1]*cos[Theta_3]*exp[i*Phi_2] u23 = -cos[Theta_2]*sin[Theta_3]*exp[-i*Phi_1 - i*Phi_5] -

sin[Theta_1]*sin[Theta_2]*cos[Theta_3]*exp[i*Phi_2 - i*Phi_3 + i*Phi_4]

u31 = -sin[Theta_1]*cos[Theta_2]*sin[Theta_3]*exp[i*Phi_1 - i*Phi_3 + i*Phi_5]

- sin[Theta_2]*cos[Theta_3]*exp[-i*Phi_2-i*Phi_4]

u32 = cos[Theta_1]*sin[Theta_3]*exp[i*Phi_5] u33 = cos[Theta_2]*cos[Theta_3]*exp[-i*Phi_1 - i*Phi_2] -

sin[Theta_1]*sin[Theta_2]*sin[Theta_3]*exp[-i*Phi_3 + i*Phi_4 + i*Phi_5]

qsearch.utils.**random_near_identity**(*n*, *alpha*)

qsearch.utils.**remap**(*U*, *order*, *d=2*)

qsearch.utils.**upgrade_qudits**(*U*, *di=2*, *df=3*)

qsearch.utils.**endian_reverse**(*U*, *d=2*)

qsearch.utils.**mpi_rank**()

qsearch.utils.**mpi_do_work**(*comm*)

Do the work of a single compilation.

Parameters

comm – An MPI communication object

qsearch.utils.**mpi_worker**()

Create a worker that will keep running compilation requests until told to stop

3.1.2 Package Contents

Classes

Options	This class manages options that are passed between various Qsearch objects.
Compiler	This class defines the pattern for compilers that convert a unitary matrix to a circuit that implements that matrix.
SearchCompiler	This Compiler uses an A* search strategy to synthesize a unitary, as described in the paper Towards Optimal Topology Aware Quantum Circuit Synthesis.
Gate	This class shows the framework for working with quantum gates in Qsearch.
IdentityGate	Represents an identity gate of any number of qudits of any size.
XGate	Represents a parameterized X rotation on one qubit.
YGate	Represents a parameterized Y rotation on one qubit.
ZGate	Represents a parameterized Z rotation on one qubit.
SXGate	Represents a sqrt(X) rotation on one qubit, which is equivalent to XGate() with a paramter of pi/2, up to an overall phase.
ZXZXZGate	Represents an arbitrary parameterized single-qubit gate, decomposed into 3 parameterized Z gates separated by X(PI/2) gates.
XZXZGate	Represents a partially parameterized single qubit gate, equivalent to ZXZXZ but without the first Z gate. This is useful because that first Z gate can commute through the control of a CNOT, thereby reducing the number of parameters we need to solve for.
U3Gate	Represents an arbitrary parameterized single qubit gate, parameterized in the same way as IBM's U3 gate.
U2Gate	Represents a parameterized single qubit gate, parameterized in the same way as IBM's U2 gate.
U1Gate	Represents an parameterized single qubit gate, parameterized in the same way as IBM's U1 gate.
SingleQutritGate	This gate represents an arbitrary parameterized single-qutrit gate.
CSUMGate	Represents the constant two-qutrit gate CSUM
CPIGate	Represents the constant two-qutrit gate CPI.
CPIPhaseGate	Represents the constant two-qutrit gate CPI with phase differences.
CNOTGate	Represents the constant two-qubit gate CNOT.
CZGate	Represents the constant two-qubit gate Controlled-Z.
ISwapGate	Represents the constant two-qubit gate ISwap.
XXGate	Represents the constant two-qubit gate XX(pi/2).
NonadjacentCNOTGate	Represents the two-qubit gate CNOT, but between two qubits that are not necessarily next to each other.
UGate	Represents an arbitrary constant gate, defined by the unitary passed to the initializer.
UpgradedConstantGate	Represents a constant gate, based on the Gate passed to its initializer, but upgraded to act on qudits of a larger size.
CUGate	Represents an arbitrary controlled gate, defined by the unitary passed to the initializer.
CNOTRootGate	Represents the sqrt(CNOT) gate. Two sqrt(CNOT) gates in a row will form a CNOT gate.
KroneckerGate	Represents the Kronecker product of a list of gates. This is equivalent to performing those gate in parallel in a quantum circuit.
ProductGate	Represents a matrix product of Gates. This is equivalent to performing those gates sequentially in a quantum

Attributes

standard_defaults
standard_smart_defaults
native_from_object

class qsearch.**Options**(*defaults={}*, *smart_defaults={}*, ***xtraargs*)

This class manages options that are passed between various Qsearch objects.

filtered(**names*)

Returns an Options object with only parameters in the specified list names.

__getitem__(*name*)

__delitem__(*name*)

__getattr__(*name*)

__setattr__(*name*, *value*)

Implement setattr(self, name, value).

__contains__(*name*)

manually_entered(**names*, *location='dict'*, *operator='all'*)

empty_copy()

Create an Options object with the same defaults but without any specific values.

copy()

Create a full copy of an Options object.

__copy__()

updated(*other=None*, ***xtraargs*)

Return a new Options object that is a copy of this object, updated with the contents of other and xtraargs.

update(*other=None*, ***xtraargs*)

Mutate the current Options object with the contents of other and xtraargs.

_update_dict(*otherdict*)

set_defaults(***args*)

Set default values for this Options object.

If an Options object is queried for a value, and it does not contain it, it will check its defaults list before throwing an error.

set_smart_defaults(***args*)

Set smart_defaults values for this Options object.

If an Options object is queried for a value, and it does not contain it, it will check its smart_defaults list before throwing an error. If it does find a function in smart_defaults, it calls that function, passing itself as the argument, and returns the return value of that function, caching it for next time.

make_required(**names*)

Marking names as required will cause the Options object to throw an error if it does not contain it, even if it has defaults or smart_defaults defined.

remove_defaults(**names*)

Removes the defaults for the specified names.

remove_smart_defaults(**names*)

Removes the smart_defaults for the specified names.

generate_cache()

Caches valuesa for all functions in smart_defaults.

save(*filepath=None*)

Saves the Options object to a file, or to a returned tuple.

load(*filepath_or_tuple*, *strict=False*)

Loads the Options object from a file or tuple.

If strict is left as False, the Options object will attempt to gracefully handle errors when loading its contents, relying on its ability to fallback to defaults or smart_defaults if those are able to load successfully.

If strict is set to True, the Options object will throw an error upon any error while loading.

__getstate__()

__setstate__(*state*)

qsearch.**standard_defaults**

qsearch.**standard_smart_defaults**

class qsearch.**Compiler**(*options=Options()*)

This class defines the pattern for compilers that convert a unitary matrix to a circuit that implements that matrix.

abstract compile(*options*)

class qsearch.**SearchCompiler**(*options=Options()*)

Bases: *Compiler*

This Compiler uses an A* search strategy to synthesize a unitary, as described in the paper Towards Optimal Topology Aware Quantum Circuit Synthesis.

Options:

target (required) : The unitary matrix to be synthesized, in the form of a numpy ndarray with dtype="complex128". gateset : The Gateset used for synthesis. weight_limit : A limit on the maximum weight for circuits to be expanded for further searching. See gatesets.py for more information. The default is None for unlimited. heuristic : A heuristic used to order the search tree. See heuristics.py for more information. solver : A Solver used for optimizing the parameters in parameterized circuits generated by the search tree. parallelizer : A Parallelizer used for solving multiple parameterized circuits in parallel. beams : The number of nodes to pop from the search tree at a time. The default value of -1 will create enough branches to maximize utilization of your CPU. objective : An Objective used for scoring the quality of a parameterization for both synthesis and search. timeout : An uper limit on the amount of time the compiler will spend trying to synthesize a circuit. The default is float('inf'), for unlimited. checkpoint : The compiler will use this Checkpoint to save intermediate state, and will resume from this Checkpoint if there was an existing state. logger : A qsearch.logging.Logger that will be used for logging the synthesis process.

Parameters
options – See class level documentation for the options SearchCompiler uses

compile(*options=Options()*)

Parameters
options – See class level documentation for the options SearchCompiler uses

qsearch.**native_from_object**

class qsearch.**Gate**

This class shows the framework for working with quantum gates in Qsearch.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

abstract matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

mat_jac(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

abstract assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__eq__(*other*)
Return self==value.

__hash__()
Return hash(self).

copy()

_parts()

__copy__()

__deepcopy__(*memo*)

__repr__()
Return repr(self).

validate_structure()

class qsearch.**IdentityGate**(*qudits=1*, *d=2*)
Bases: *Gate*

Represents an identity gate of any number of qudits of any size.

Parameters

- **qudits** – The number of qudits represented by this identity.
- **d** – The size of qudits represented by this identity (2 for qubits, 3 for qutrits, etc.)

matrix(*v*)
Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.**XGate**

Bases: *Gate*

Represents a parameterized X rotation on one qubit.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

mat_jac(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.**YGate**

Bases: *Gate*

Represents a parameterized Y rotation on one qubit.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`mat_jac`(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

`assemble`(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

`__repr__`()

Return repr(self).

class qsearch.**ZGate**

Bases: *Gate*

Represents a parameterized Z rotation on one qubit.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

`matrix`(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`mat_jac`(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

`assemble`(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.**SXGate**

Bases: *Gate*

Represents a sqrt(X) rotation on one qubit, which is equivalent to XGate() with a paramter of pi/2, up to an overall phase.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.ZXZXZGate

Bases: Gate

Represents an arbitrary parameterized single-qubit gate, decomposed into 3 parameterized Z gates separated by X(PI/2) gates.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters

v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns

A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type

np.ndarray

mat_jac(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters

v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns

A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type

tuple

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns

A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()
Return repr(self).

class qsearch.**XZXZGate**

Bases: *Gate*

Represents a partially parameterized single qubit gate, equivalent to ZXZXZ but without the first Z gate. This is useful because that first Z gate can commute through the control of a CNOT, thereby reducing the number of parameters we need to solve for.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

matrix(*v*)
Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

mat_jac(*v*)
Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

`assemble`(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

`__repr__`()

Return repr(self).

`class qsearch.U3Gate`

Bases: *Gate*

Represents an arbitrary parameterized single qubit gate, parameterized in the same way as IBM's U3 gate.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

`matrix`(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`mat_jac`(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

`assemble`(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

`__eq__`(*other*)

Return self==value.

`__repr__`()

Return repr(self).

class qsearch.**U2Gate**

Bases: *Gate*

Represents a parameterized single qubit gate, parameterized in the same way as IBM's U2 gate.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

`matrix`(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`mat_jac`(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

`assemble`(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__eq__(*other*)

Return self==value.

__repr__()

Return repr(self).

class qsearch.**U1Gate**

Bases: *Gate*

Represents an parameterized single qubit gate, parameterized in the same way as IBM's U1 gate.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

mat_jac(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).

- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__eq__(*other*)
Return self==value.

__repr__()
Return repr(self).

class qsearch.**SingleQutritGate**
Bases: *Gate*

This gate represents an arbitrary parameterized single-qutrit gate.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

matrix(*v*)
Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

mat_jac(*v*)
Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.**CSUMGate**

Bases: *Gate*

Represents the constant two-qutrit gate CSUM

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

_csum

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`assemble`(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

`__repr__`()

Return repr(self).

`class qsearch.CPIGate`

Bases: *Gate*

Represents the constant two-qutrit gate CPI.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

`_cpi`

`matrix`(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.**CPIPhaseGate**

Bases: *Gate*

Represents the constant two-qutrit gate CPI with phase differences.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()
Return repr(self).

class qsearch.**CNOTGate**

Bases: *Gate*

Represents the constant two-qubit gate CNOT.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

_cnot

__eq__(*other*)
Return self==value.

matrix(*v*)
Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)
Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.**CZGate**

Bases: *Gate*

Represents the constant two-qubit gate Controlled-Z.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

_gate

__eq__(*other*)

Return self==value.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).

- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()
Return repr(self).

class qsearch.**ISwapGate**

Bases: *Gate*

Represents the constant two-qubit gate ISwap.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

_gate

__eq__(*other*)
Return self==value.

matrix(*v*)
Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)
Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()
Return repr(self).

class qsearch.**XXGate**
Bases: *Gate*

Represents the constant two-qubit gate XX(pi/2).

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

_gate

__eq__(*other*)
Return self==value.

matrix(*v*)
Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)
Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

`__repr__()`

Return repr(self).

`class qsearch.NonadjacentCNOTGate(qudits, control, target)`

Bases: `Gate`

Represents the two-qubit gate CNOT, but between two qubits that are not necessarily next to each other.

Parameters

- **qudits** – The total number of qubits that a unitary of the size returned by this gate would represent. For this gate, usually this is the total number of qubits in the larger circuit.
- **control** – The index of the control qubit, relative to the 0th qubit that would be affected by the unitary returned by this gate.
- **target** – The index of the target qubit, relative to the 0th qubit that would be affected by the unitary returned by this gate.

`matrix(v)`

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`assemble(v, i=0)`

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

validate_structure()

class qsearch.**UGate**(*U*, *d=2*, *gatename='CUSTOM'*, *gateparams=()*, *gateindices=None*)

Bases: Gate

Represents an arbitrary constant gate, defined by the unitary passed to the initializer.

Parameters

- **U** – The unitary for the operation that this gate represents, as a numpy ndarray with datatype="complex128".
- **d** – The size of qudits for the operation that this gate represents. The default is 2, for qubits.
- **gatename** – A name for this gate, which will get passed to the Assembler at assembly time.
- **gateparams** – A tuple of parameters that will get passed to the Assembler at assembly time.
- **gateindices** – A tuple of indices for the qubits that this gate acts on, which will get passed to the Assembler at assembly time. This overrides the default behavior, which is to return a tuple of all the indices starting with the one passed in assemble(v, i), and ending at i+self.qudits

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters

v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns

A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type

np.ndarray

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns

A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

`__repr__()`

Return repr(self).

`class qsearch.UpgradedConstantGate(other, df=3)`

Bases: `Gate`

Represents a constant gate, based on the Gate passed to its initializer, but upgraded to act on qudits of a larger size.

Parameters

- **other** – A Gate of a lower qudit size.
- **df** – The final, upgraded qudit size. The default is 3, for upgrading gates from qubits to qutrits.

`matrix(v)`

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`assemble(v, i=0)`

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.**CUGate**(*U*, *gatename='Name'*, *gateparams=()*, *flipped=False*)

Bases: Gate

Represents an arbitrary controlled gate, defined by the unitary passed to the initializer.

Parameters

- **U** – The unitary to form the controlled-unitary gate, in the form of a numpy ndarray with dtype="complex128"
- **gatename** – A name for this controlled gate which will get passed to the Assembler at assembly time.
- **gateparams** – A tuple of parameters that will get passed to the Assembler at assembly time.
- **flipped** – A boolean flag, which if set to true, will flip the direction of the gate. The default direction is for the control qubit to be the lower indexed qubit.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.**CNOTRootGate**

Bases: *Gate*

Represents the sqrt(CNOT) gate. Two sqrt(CNOT) gates in a row will form a CNOT gate.

Gates must set the following variables in __init__

self.num_inputs : The number of parameters needed to generate a unitary. This can be 0. self.qudits : The number of qudits acted on by a unitary of the size generated by the gate. For example, this would be 1 for U3, 2 for CNOT.

_cnr

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

__repr__()

Return repr(self).

class qsearch.**KroneckerGate**(**subgates*)

Bases: *Gate*

Represents the Kronecker product of a list of gates. This is equivalent to performing those gate in parallel in a quantum circuit.

Parameters

***subgates** – An sequence of Gates. KroneckerGate will return the kronecker product of the unitaries returned by those Gates.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters

v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns

A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type

np.ndarray

mat_jac(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters

v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns

A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type

tuple

assemble(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

appending(*gate*)

Returns a new KroneckerGate with the new gate added to the list.

Parameters
gate – A Gate to be added to the end of the list of gates in the new KroneckerGate.

_parts()

__deepcopy__(*memo*)

__repr__()

Return repr(self).

validate_structure()

class qsearch.**ProductGate**(**subgates*)

Bases: *Gate*

Represents a matrix product of Gates. This is equivalent to performing those gates sequentially in a quantum circuit.

Parameters
subgates – A list of Gates to be multiplied together. ProductGate returns the matrix product of the unitaries returned by those Gates.

matrix(*v*)

Generates a matrix using the given vector of input parameters. For a constant gate, v will be empty.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A unitary matrix with dtype="complex128", equal in size to d**self.qudits, where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

Return type
np.ndarray

`mat_jac`(*v*)

Generates a matrix and the jacobian(s) using the given vector of input parameters.

It is not required to implement mat_jac for constant gates, nor is it required when using gradient-free Solvers.

The jacobian matrices will be complex valued, and should be the elementwise partial derivative with respect to each of the parameters. There should be self.num_inputs matrices in the array, with the ith entry being the partial derivative with respect to v[i]. See U3Gate for an example implementation.

Parameters
v – A numpy array of real floating point numbers, ranging from 0 to 2*PI. Its size is equal to self.num_inputs

Returns
A tuple of the same unitary that would be returned by matrix(v), and an array of Jacobian matrices.

Return type
tuple

`assemble`(*v*, *i=0*)

Generates an array of tuples as an intermediate format before being processed by an Assembler for conversion to other circuit formats.

Parameters

- **v** – The same numpy array of real floating point numbers that might be passed to matrix(v).
- **i** – The index of the lowest-indexed qubit that the unitary generated by the gate acts on.

Returns
A list of tuples following the format described above.

Return type
list

The format of the tuples returned looks like:

*("gate", gatename, (*gateparameters), (*gateindices))*

Where *gatename* corresponds to a gate that an Assembler will recognize, *gateparameters* corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gateindices* corresponds to the qubit indices that the gate acts on (usually but not always calculated from i).

You can also have tuples of the form ("block", *tuples) Where tuples is an array of tuples in this same format.

For some helpful examples, look at U3Gate, XZXZGate, CNOTGate, and NonadjacentCNOTGate.

`appending`(**gates*)

Returns a new ProductGate with the new gates appended to the end.

Parameters
gates – A list of Gates to be appended.

`inserting`(**gates*, *depth=-1*)

Returns a new ProductGate with new *gates* inserted at some index *depth*.

Parameters

- **gates** – A list of Gates to be inserted.
- **depth** – An index in the subgates of the ProductGate after which the new gates will be inserted. The default value of -1 will insert these gates at the begining of the ProductGate.

__deepcopy__(*memo*)

__repr__()

Return repr(self).

validate_structure()

class qsearch.Project(*path*, *use_mpi=False*)

The project class wraps most of the functionality of Qsearch as intended to help manage working with Qsearch.

property compilations

The list of names corresponding to compilations on this Project.

_save()

_checkpoint_path(*name*)

add_compilation(*name*, *U*, *options=None*, *handle_existing=None*, ***extraargs*)

Adds a unitary to be compiled.

Parameters

- **name** – A name for this unitary. Must be unique in this Project.
- **U** – The unitary to be compiled, in the form of a numpy ndarray with dtype="complex128"
- **handle_existing** – A variable which defines how to behave if a compilation with the given name already exists. If it is set to "ignore", it will simply return without doing anything. If it is set to "overwrite", it will overwrite the previous entry. If it is set to the default of None, it will offer a warning asking the user to remove and re-add the compilation.
- **options** – The options passed to this function will be used only when this compilation is run.
- **extraargs** – The extraargs passed to this function will be used only when this compilation is run.

__setitem__(*keyword*, *value*)

configure_compiler_override(*keyword*, *value*)

An unsafe method that allows the user to set global Project Options even if there is existing work.

__getitem__(*keyword*)

__delitem__(*keyword*)

configure(***dictionary*)

Adds multiple options to the global Project Options at once.

reset(*name=None*)

Resets a Project, removing any work done but not the initial configurations.

Parameters

name – Optionally specify a particular compilation by name to reset

remove_compilation(*name*)

Removes a compilation from a Project.

Parameters

name – The name of the compilation to remove

clear(*name=None*)

Clears a Project, reverting it to a state similar to a newly created Project.

Parameters
name – Optionally specify a particular compilation by name to clear

__enter__()

__exit__(*exc_typ*, *exc_val*, *exc_tb*)

set_defaults(*defaults=standard_defaults*)

Updates the Project Options with the standard defaults from defaults.py, or a provided dictionary.

set_smart_defaults(*smart_defaults=standard_smart_defaults*)

Updates the Project Options with the standard smart_defaults from defaults.py, or a provided dictionary

run()

Runs all of the compilations in the Project.

post_process(*postprocessor*, *name=None*, *options=None*, ***xtraargs*)

Post-processes the specified compilation, or all compilations if name is None, using the specified postprocessor.

Parameters

- **postprocessor** – The qsearch.post_processing.PostProcessor to run on the compilation or project
- **name** – Optionally specify a particular compilation by name to reset
- **options** – Options to pass to the qsearch.post_processing.PostProcessor passed in *postprocessor*
- **extraargs** – Extra arguments passed as options to the qsearch.post_processing.PostProcessor passed in *postprocessor*

complete()

Returns a True if all compilations in the Project have finished and False otherwise.

finish()

Called when done running compilations in order to end MPI tasks.

status(*name=None*, *logger=None*)

Prints a status update on how much of a Project has finished running.

Parameters
name – Optionally specify which compilation to check the status of

_compilation_status(*name*)

_overall_status()

get_result(*name*)

Get the result of a compilation.

Parameters
name – The name of the compilation to get the result dictionary from

Returns
The result dictionary for a finished compilation. Usually this contains the entries "structure", a Gate, and "parameters", an array of real number parameters.

Return type
dict

`get_target`(*name*)

Get the target unitary of a compilation.

Parameters
name – The name of the compilation to get the target from

Returns
The target unitary of the compilation

Return type
np.ndarray

`get_time`(*name*)

Get the runtime that it took to run a compilation.

Parameters
name – The name of the compilation to get the runtime of

Returns
The number of seconds the compilation took

Return type
float

`get_options`(*name=None*)

Get the qsearch.options.Options object from a compilation of project

Parameters
name – Optionally pass the name of the compilation to get the qsearch.options.Options object from

Returns
the requested options object

Return type
qsearch.options.Options

`assemble`(*name*, *options=None*, ***xtraargs*)

Assembles a compilation using the Assembler specified as assembler in the Options. :param name: The compilation to assemble :param options: Contains the qsearch.assemblers.Assembler to use in assembly

Returns
The resulting assembled code

Return type
str

CHAPTER

FOUR

WORKING WITH NONLINEAR TOPOLOGIES

The default topology is linear. To synthesize for another topology, you will need to choose a gateset for your desired topology, usually either *QubitCNOTRing* or *QubitCNOTAdjacencyList*, but custom gatesets are also supported. See *Gatesets in qsearch* for more information.

CHAPTER

FIVE

WORKING WITH NONSTANDARD GATES OR QUTRITS

You will to choose a gateset that supports your desired gates. See *Gatesets in qsearch* for a list of implemented gatesets, and instructions on how to make your own. See *Gates in qsearch* for a list of supported gates and instructions on how to make your own.

CHAPTER

SIX

CUSTOMIZING YOUR COMPILATION

Once you have your desired gateset object, you can pass it either to a *Project* or a *SearchCompiler*. In addition, both *Project* and *SearchCompiler* have many other options used for customizing things like the search type or distance function. See the *Options* API documentation for more information.

Make sure to check out the example scripts as well!

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

q

qsearch, 7
qsearch.advanced_unitaries, 7
qsearch.assemblers, 8
qsearch.backends, 10
qsearch.checkpoints, 11
qsearch.comparison, 14
qsearch.compiler, 16
qsearch.defaults, 17
qsearch.evaluation, 19
qsearch.gates, 21
qsearch.gatesets, 53
qsearch.heuristics, 64
qsearch.integrations, 65
qsearch.leap_compiler, 66
qsearch.logging, 68
qsearch.multistart_solvers, 68
qsearch.objectives, 69
qsearch.options, 70
qsearch.parallelizers, 72
qsearch.persistent_aposmm, 75
qsearch.post_processing, 78
qsearch.project, 80
qsearch.solvers, 83
qsearch.unitaries, 85
qsearch.utils, 89

INDEX

Symbols

__contains__() (*qsearch.Options method*), 95
__contains__() (*qsearch.options.Options method*), 71
__copy__() (*qsearch.Gate method*), 98
__copy__() (*qsearch.Options method*), 95
__copy__() (*qsearch.gates.Gate method*), 25
__copy__() (*qsearch.options.Options method*), 71
__deepcopy__() (*qsearch.Gate method*), 98
__deepcopy__() (*qsearch.KroneckerGate method*), 124
__deepcopy__() (*qsearch.ProductGate method*), 125
__deepcopy__() (*qsearch.gates.Gate method*), 25
__deepcopy__() (*qsearch.gates.KroneckerGate method*), 51
__deepcopy__() (*qsearch.gates.ProductGate method*), 52
__delitem__() (*qsearch.Options method*), 95
__delitem__() (*qsearch.Project method*), 126
__delitem__() (*qsearch.options.Options method*), 71
__delitem__() (*qsearch.project.Project method*), 81
__enter__() (*qsearch.Project method*), 127
__enter__() (*qsearch.project.Project method*), 81
__eq__() (*qsearch.CNOTGate method*), 114
__eq__() (*qsearch.CZGate method*), 115
__eq__() (*qsearch.Gate method*), 98
__eq__() (*qsearch.ISwapGate method*), 116
__eq__() (*qsearch.U1Gate method*), 110
__eq__() (*qsearch.U2Gate method*), 109
__eq__() (*qsearch.U3Gate method*), 107
__eq__() (*qsearch.XXGate method*), 117
__eq__() (*qsearch.gates.CNOTGate method*), 41
__eq__() (*qsearch.gates.CZGate method*), 42
__eq__() (*qsearch.gates.Gate method*), 25
__eq__() (*qsearch.gates.ISwapGate method*), 43
__eq__() (*qsearch.gates.U1Gate method*), 37
__eq__() (*qsearch.gates.U2Gate method*), 36
__eq__() (*qsearch.gates.U3Gate method*), 34
__eq__() (*qsearch.gates.XXGate method*), 44
__eq__() (*qsearch.gatesets.Gateset method*), 56
__eq__() (*qsearch.solvers.Solver method*), 84
__exit__() (*qsearch.Project method*), 127
__exit__() (*qsearch.project.Project method*), 81
__getattr__() (*qsearch.Options method*), 95
__getattr__() (*qsearch.options.Options method*), 71
__getitem__() (*qsearch.Options method*), 95
__getitem__() (*qsearch.Project method*), 126
__getitem__() (*qsearch.options.Options method*), 71
__getitem__() (*qsearch.project.Project method*), 81
__getstate__() (*qsearch.Options method*), 96
__getstate__() (*qsearch.options.Options method*), 72
__hash__() (*qsearch.Gate method*), 98
__hash__() (*qsearch.gates.Gate method*), 25
__repr__() (*qsearch.CNOTGate method*), 115
__repr__() (*qsearch.CNOTRootGate method*), 123
__repr__() (*qsearch.CPIGate method*), 113
__repr__() (*qsearch.CPIPhaseGate method*), 114
__repr__() (*qsearch.CSUMGate method*), 112
__repr__() (*qsearch.CUGate method*), 122
__repr__() (*qsearch.CZGate method*), 116
__repr__() (*qsearch.Gate method*), 98
__repr__() (*qsearch.ISwapGate method*), 117
__repr__() (*qsearch.IdentityGate method*), 99
__repr__() (*qsearch.KroneckerGate method*), 124
__repr__() (*qsearch.NonadjacentCNOTGate method*), 119
__repr__() (*qsearch.ProductGate method*), 126
__repr__() (*qsearch.SXGate method*), 103
__repr__() (*qsearch.SingleQutritGate method*), 111
__repr__() (*qsearch.U1Gate method*), 110
__repr__() (*qsearch.U2Gate method*), 109
__repr__() (*qsearch.U3Gate method*), 107
__repr__() (*qsearch.UGate method*), 120
__repr__() (*qsearch.UpgradedConstantGate method*), 121
__repr__() (*qsearch.XGate method*), 100
__repr__() (*qsearch.XXGate method*), 118
__repr__() (*qsearch.XZXZGate method*), 106
__repr__() (*qsearch.YGate method*), 101
__repr__() (*qsearch.ZGate method*), 103
__repr__() (*qsearch.ZXZXZGate method*), 105
__repr__() (*qsearch.gates.CNOTGate method*), 42
__repr__() (*qsearch.gates.CNOTRootGate method*), 50
__repr__() (*qsearch.gates.CPIGate method*), 40
__repr__() (*qsearch.gates.CPIPhaseGate method*), 41
__repr__() (*qsearch.gates.CSUMGate method*), 39

__repr__() (*qsearch.gates.CUGate method*), 49
__repr__() (*qsearch.gates.CZGate method*), 43
__repr__() (*qsearch.gates.Gate method*), 25
__repr__() (*qsearch.gates.ISwapGate method*), 44
__repr__() (*qsearch.gates.IdentityGate method*), 26
__repr__() (*qsearch.gates.KroneckerGate method*), 51
__repr__() (*qsearch.gates.NonadjacentCNOTGate method*), 46
__repr__() (*qsearch.gates.ProductGate method*), 53
__repr__() (*qsearch.gates.SXGate method*), 30
__repr__() (*qsearch.gates.SingleQutritGate method*), 38
__repr__() (*qsearch.gates.U1Gate method*), 37
__repr__() (*qsearch.gates.U2Gate method*), 36
__repr__() (*qsearch.gates.U3Gate method*), 34
__repr__() (*qsearch.gates.UGate method*), 47
__repr__() (*qsearch.gates.UpgradedConstantGate method*), 48
__repr__() (*qsearch.gates.XGate method*), 27
__repr__() (*qsearch.gates.XXGate method*), 45
__repr__() (*qsearch.gates.XZXZGate method*), 33
__repr__() (*qsearch.gates.YGate method*), 28
__repr__() (*qsearch.gates.ZGate method*), 30
__repr__() (*qsearch.gates.ZXZXZGate method*), 32
__setattr__() (*qsearch.Options method*), 95
__setattr__() (*qsearch.options.Options method*), 71
__setitem__() (*qsearch.Project method*), 126
__setitem__() (*qsearch.project.Project method*), 81
__setstate__() (*qsearch.Options method*), 96
__setstate__() (*qsearch.options.Options method*), 72
_checkpoint_path() (*qsearch.Project method*), 126
_checkpoint_path() (*qsearch.project.Project method*), 80
_cnot (*qsearch.CNOTGate attribute*), 114
_cnot (*qsearch.gates.CNOTGate attribute*), 41
_cnr (*qsearch.CNOTRootGate attribute*), 122
_cnr (*qsearch.gates.CNOTRootGate attribute*), 49
_compilation_status() (*qsearch.Project method*), 127
_compilation_status() (*qsearch.project.Project method*), 82
_cpi (*qsearch.CPIGate attribute*), 112
_cpi (*qsearch.gates.CPIGate attribute*), 39
_csum (*qsearch.CSUMGate attribute*), 111
_csum (*qsearch.gates.CSUMGate attribute*), 38
_gate (*qsearch.CZGate attribute*), 115
_gate (*qsearch.ISwapGate attribute*), 116
_gate (*qsearch.XXGate attribute*), 117
_gate (*qsearch.gates.CZGate attribute*), 42
_gate (*qsearch.gates.ISwapGate attribute*), 43
_gate (*qsearch.gates.XXGate attribute*), 44
_options_actual_parameters (*in module qsearch.options*), 71
_overall_status() (*qsearch.Project method*), 127
_overall_status() (*qsearch.project.Project method*), 82
_parts() (*qsearch.Gate method*), 98
_parts() (*qsearch.KroneckerGate method*), 124
_parts() (*qsearch.gates.Gate method*), 25
_parts() (*qsearch.gates.KroneckerGate method*), 51
_save() (*qsearch.Project method*), 126
_save() (*qsearch.project.Project method*), 80
_update_dict() (*qsearch.Options method*), 95
_update_dict() (*qsearch.options.Options method*), 71

A

add_compilation() (*qsearch.Project method*), 126
add_compilation() (*qsearch.project.Project method*), 80
aposmm() (*in module qsearch.persistent_aposmm*), 75
appending() (*qsearch.gates.KroneckerGate method*), 51
appending() (*qsearch.gates.ProductGate method*), 52
appending() (*qsearch.KroneckerGate method*), 124
appending() (*qsearch.ProductGate method*), 125
arbitrary_cnot (*in module qsearch.unitaries*), 86
arbitrary_cnot() (*in module qsearch.unitaries*), 89
assemble() (*qsearch.assemblers.Assembler method*), 9
assemble() (*qsearch.assemblers.DictionaryAssembler method*), 9
assemble() (*qsearch.CNOTGate method*), 114
assemble() (*qsearch.CNOTRootGate method*), 122
assemble() (*qsearch.CPIGate method*), 113
assemble() (*qsearch.CPIPhaseGate method*), 113
assemble() (*qsearch.CSUMGate method*), 112
assemble() (*qsearch.CUGate method*), 121
assemble() (*qsearch.CZGate method*), 115
assemble() (*qsearch.Gate method*), 97
assemble() (*qsearch.gates.CNOTGate method*), 41
assemble() (*qsearch.gates.CNOTRootGate method*), 49
assemble() (*qsearch.gates.CPIGate method*), 40
assemble() (*qsearch.gates.CPIPhaseGate method*), 40
assemble() (*qsearch.gates.CSUMGate method*), 39
assemble() (*qsearch.gates.CUGate method*), 48
assemble() (*qsearch.gates.CZGate method*), 42
assemble() (*qsearch.gates.Gate method*), 24
assemble() (*qsearch.gates.IdentityGate method*), 25
assemble() (*qsearch.gates.ISwapGate method*), 43
assemble() (*qsearch.gates.KroneckerGate method*), 50
assemble() (*qsearch.gates.NonadjacentCNOTGate method*), 45
assemble() (*qsearch.gates.ProductGate method*), 52
assemble() (*qsearch.gates.SingleQutritGate method*), 38
assemble() (*qsearch.gates.SXGate method*), 30
assemble() (*qsearch.gates.U1Gate method*), 36
assemble() (*qsearch.gates.U2Gate method*), 35
assemble() (*qsearch.gates.U3Gate method*), 34
assemble() (*qsearch.gates.UGate method*), 46

assemble() (*qsearch.gates.UpgradedConstantGate method*), 47
assemble() (*qsearch.gates.XGate method*), 27
assemble() (*qsearch.gates.XXGate method*), 44
assemble() (*qsearch.gates.XZXZGate method*), 33
assemble() (*qsearch.gates.YGate method*), 28
assemble() (*qsearch.gates.ZGate method*), 29
assemble() (*qsearch.gates.ZXZXZGate method*), 31
assemble() (*qsearch.IdentityGate method*), 98
assemble() (*qsearch.ISwapGate method*), 116
assemble() (*qsearch.KroneckerGate method*), 123
assemble() (*qsearch.NonadjacentCNOTGate method*), 118
assemble() (*qsearch.ProductGate method*), 125
assemble() (*qsearch.Project method*), 128
assemble() (*qsearch.project.Project method*), 83
assemble() (*qsearch.SingleQutritGate method*), 111
assemble() (*qsearch.SXGate method*), 103
assemble() (*qsearch.U1Gate method*), 109
assemble() (*qsearch.U2Gate method*), 108
assemble() (*qsearch.U3Gate method*), 107
assemble() (*qsearch.UGate method*), 119
assemble() (*qsearch.UpgradedConstantGate method*), 120
assemble() (*qsearch.XGate method*), 100
assemble() (*qsearch.XXGate method*), 117
assemble() (*qsearch.XZXZGate method*), 106
assemble() (*qsearch.YGate method*), 101
assemble() (*qsearch.ZGate method*), 102
assemble() (*qsearch.ZXZXZGate method*), 104
Assembler (*class in qsearch.assemblers*), 9
ASSEMBLER_IBMOPENQASM (*in module qsearch.assemblers*), 8, 10
ASSEMBLER_OPENQASM (*in module qsearch.assemblers*), 8, 10
ASSEMBLER_QISKIT (*in module qsearch.assemblers*), 8, 10
ASSEMBLER_QUTRIT (*in module qsearch.assemblers*), 8, 10
assemblydict_ibmopenqasm (*in module qsearch.assemblers*), 10
assemblydict_openqasm (*in module qsearch.assemblers*), 10
assemblydict_qiskit (*in module qsearch.assemblers*), 10
assemblydict_qutrit (*in module qsearch.assemblers*), 10
astar() (*in module qsearch.heuristics*), 65

B

Backend (*class in qsearch.backends*), 11
BackwardsCompatibleObjective (*class in qsearch.objectives*), 70
BasicSingleQubitReduction_PostProcessor (*class in qsearch.post_processing*), 78
BasicSingleQubitReduction_PostProcessor (*in module qsearch.post_processing*), 78
BFGS_Jac_Solver (*class in qsearch.solvers*), 84
branching_factor() (*qsearch.gatesets.Gateset method*), 55
branching_factor() (*qsearch.gatesets.QubitCNOTLinear method*), 58
branching_factor() (*qsearch.gatesets.QubitCZLinear method*), 59
branching_factor() (*qsearch.gatesets.QubitISwapLinear method*), 60
branching_factor() (*qsearch.gatesets.QubitXXLinear method*), 62

C

Checkpoint (*class in qsearch.checkpoints*), 12
ChildCheckpoint (*class in qsearch.checkpoints*), 13
ChildCheckpoint (*in module qsearch.checkpoints*), 11
clear() (*qsearch.Project method*), 126
clear() (*qsearch.project.Project method*), 81
CMA_Solver (*class in qsearch.solvers*), 84
cnot (*in module qsearch.unitaries*), 85, 88
CNOTGate (*class in qsearch*), 114
CNOTGate (*class in qsearch.gates*), 41
CNOTRootGate (*class in qsearch*), 122
CNOTRootGate (*class in qsearch.gates*), 49
COBYLA_Solver (*class in qsearch.solvers*), 84
compilations (*qsearch.Project property*), 126
compilations (*qsearch.project.Project property*), 80
compile() (*qsearch.Compiler method*), 96
compile() (*qsearch.compiler.Compiler method*), 16
compile() (*qsearch.compiler.SearchCompiler method*), 17
compile() (*qsearch.leap_compiler.LeapCompiler method*), 67
compile() (*qsearch.leap_compiler.SubCompiler method*), 67
compile() (*qsearch.post_processing.LEAPReoptimizing_PostProcessor method*), 79
compile() (*qsearch.SearchCompiler method*), 97
Compiler (*class in qsearch*), 96
Compiler (*class in qsearch.compiler*), 16
COMPLETE (*qsearch.project.Project_Status attribute*), 80
complete() (*qsearch.Project method*), 127
complete() (*qsearch.project.Project method*), 82
configure() (*qsearch.Project method*), 126
configure() (*qsearch.project.Project method*), 81
configure_compiler_override() (*qsearch.Project method*), 126
configure_compiler_override() (*qsearch.project.Project method*), 81

constraint_distsq() (*in module qsearch.evaluation*), 21
constraint_distsq_jac() (*in module qsearch.evaluation*), 21
convert() (*qsearch.integrations.QiskitGateConverter method*), 66
convert_cx() (*qsearch.integrations.QiskitGateConverter method*), 66
convert_rx() (*qsearch.integrations.QiskitGateConverter method*), 66
convert_ry() (*qsearch.integrations.QiskitGateConverter method*), 66
convert_rz() (*qsearch.integrations.QiskitGateConverter method*), 66
convert_u2() (*qsearch.integrations.QiskitGateConverter method*), 66
convert_u3() (*qsearch.integrations.QiskitGateConverter method*), 66
copy() (*qsearch.Gate method*), 98
copy() (*qsearch.gates.Gate method*), 25
copy() (*qsearch.Options method*), 95
copy() (*qsearch.options.Options method*), 71
cost_combo_linear() (*in module qsearch.evaluation*), 21
cost_combo_linear_jac() (*in module qsearch.evaluation*), 21
cost_linear() (*in module qsearch.evaluation*), 21
cost_linear_jac() (*in module qsearch.evaluation*), 21
CPIGate (*class in qsearch*), 112
CPIGate (*class in qsearch.gates*), 39
CPIPhaseGate (*class in qsearch*), 113
CPIPhaseGate (*class in qsearch.gates*), 40
CSUMGate (*class in qsearch*), 111
CSUMGate (*class in qsearch.gates*), 38
CUGate (*class in qsearch*), 121
CUGate (*class in qsearch.gates*), 48
cut_end() (*in module qsearch.leap_compiler*), 67
CZGate (*class in qsearch*), 115
CZGate (*class in qsearch.gates*), 42

D

decide_where_to_start_localopt() (*in module qsearch.persistent_aposmm*), 77
Default (*in module qsearch.gatesets*), 53, 64
default_checkpoint() (*in module qsearch.defaults*), 18
default_compiler() (*in module qsearch.defaults*), 19
default_error_func() (*in module qsearch.defaults*), 18
default_error_jac() (*in module qsearch.defaults*), 19
default_error_residuals() (*in module qsearch.defaults*), 18
default_error_residuals_jac() (*in module qsearch.defaults*), 19
default_eval_func() (*in module qsearch.defaults*), 18
default_heuristic() (*in module qsearch.defaults*), 18
default_logger() (*in module qsearch.defaults*), 18
default_num_tasks() (*in module qsearch.parallelizers*), 73
default_objective() (*in module qsearch.defaults*), 19
default_solver() (*in module qsearch.solvers*), 84
DefaultQubit (*in module qsearch.gatesets*), 53, 64
DefaultQutrit (*in module qsearch.gatesets*), 53, 64
delete() (*qsearch.checkpoints.Checkpoint method*), 12
delete() (*qsearch.checkpoints.ChildCheckpoint method*), 13
delete() (*qsearch.checkpoints.FileCheckpoint method*), 12
delete_parent() (*qsearch.checkpoints.ChildCheckpoint method*), 13
DictionaryAssembler (*class in qsearch.assemblers*), 9
distance_for_x() (*in module qsearch.multistart_solvers*), 68
distance_metric (*qsearch.solvers.LeastSquares_Jac_Solver property*), 85
distance_metric (*qsearch.solvers.Solver property*), 84
distance_with_initial_state() (*in module qsearch.comparison*), 15
distance_with_initial_state_jac() (*in module qsearch.comparison*), 16
DIY_Solver (*class in qsearch.solvers*), 84
djikstra() (*in module qsearch.heuristics*), 65
done() (*qsearch.parallelizers.MPIParallelizer method*), 74
done() (*qsearch.parallelizers.MultiprocessingParallelizer method*), 74
done() (*qsearch.parallelizers.Parallelizer method*), 74
done() (*qsearch.parallelizers.ProcessPoolParallelizer method*), 74
downgrade_qudits_residuals() (*in module qsearch.utils*), 91
downgrade_qudits_residuals_jac() (*in module qsearch.utils*), 91

E

empty_copy() (*qsearch.Options method*), 95
empty_copy() (*qsearch.options.Options method*), 71
endian_reverse (*in module qsearch.utils*), 89
endian_reverse() (*in module qsearch.utils*), 92
error_distsq() (*in module qsearch.evaluation*), 20
error_distsq_jac() (*in module qsearch.evaluation*), 20
error_stateprep_distsq() (*in module qsearch.evaluation*), 20
error_stateprep_distsq_jac() (*in module qsearch.evaluation*), 20
eval_func_from_residuals() (*in module qsearch.comparison*), 16

evaluate_step() (*in module qsearch.parallelizers*), 73

F

FileCheckpoint (*class in qsearch.checkpoints*), 12
FileCheckpoint (*in module qsearch.checkpoints*), 11
fill_row() (*in module qsearch.gatesets*), 64
filtered() (*qsearch.Options method*), 95
filtered() (*qsearch.options.Options method*), 71
find_last_3_cnots_linear() (*in module qsearch.gatesets*), 64
finish() (*qsearch.Project method*), 127
finish() (*qsearch.project.Project method*), 82
flatten_intermediate() (*in module qsearch.assemblers*), 9
fredkin (*in module qsearch.unitaries*), 85, 88
full_adder (*in module qsearch.unitaries*), 85, 88

G

Gate (*class in qsearch*), 97
Gate (*class in qsearch.gates*), 24
Gateset (*class in qsearch.gatesets*), 55
gen_error_func() (*qsearch.objectives.BackwardsCompatibleObjective method*), 70
gen_error_func() (*qsearch.objectives.MatrixDistanceObjective method*), 70
gen_error_func() (*qsearch.objectives.Objective method*), 70
gen_error_func() (*qsearch.objectives.StateprepObjective method*), 70
gen_error_jac() (*qsearch.objectives.BackwardsCompatibleObjective method*), 70
gen_error_jac() (*qsearch.objectives.MatrixDistanceObjective method*), 70
gen_error_jac() (*qsearch.objectives.Objective method*), 70
gen_error_jac() (*qsearch.objectives.StateprepObjective method*), 70
gen_error_residuals() (*qsearch.objectives.BackwardsCompatibleObjective method*), 70
gen_error_residuals() (*qsearch.objectives.MatrixDistanceObjective method*), 70
gen_error_residuals() (*qsearch.objectives.Objective method*), 70
gen_error_residuals() (*qsearch.objectives.StateprepObjective method*), 70
gen_error_residuals_jac() (*qsearch.objectives.BackwardsCompatibleObjective method*), 70
gen_error_residuals_jac() (*qsearch.objectives.MatrixDistanceObjective method*), 70
gen_error_residuals_jac() (*qsearch.objectives.Objective method*), 70
gen_error_residuals_jac() (*qsearch.objectives.StateprepObjective method*), 70
gen_eval_func() (*qsearch.objectives.Objective method*), 69
general_swap (*in module qsearch.unitaries*), 86
general_swap() (*in module qsearch.unitaries*), 89
generate_cache() (*qsearch.Options method*), 96
generate_cache() (*qsearch.options.Options method*), 72
generate_HHL() (*in module qsearch.advanced_unitaries*), 7
generate_miro() (*in module qsearch.advanced_unitaries*), 7
generate_stateprep_target_matrix() (*in module qsearch.utils*), 91
get_options() (*qsearch.Project method*), 128
get_options() (*qsearch.project.Project method*), 82
get_result() (*qsearch.Project method*), 127
get_result() (*qsearch.project.Project method*), 82
get_reusable_executor() (*in module qsearch.parallelizers*), 73
get_target() (*qsearch.Project method*), 128
get_target() (*qsearch.project.Project method*), 82
get_time() (*qsearch.Project method*), 128
get_time() (*qsearch.project.Project method*), 82
greedy() (*in module qsearch.heuristics*), 65

H

HHL (*in module qsearch.advanced_unitaries*), 7

I

identity (*in module qsearch.unitaries*), 86
identity() (*in module qsearch.defaults*), 19
identity() (*in module qsearch.unitaries*), 89
IdentityGate (*class in qsearch*), 98
IdentityGate (*class in qsearch.gates*), 25
index_test() (*in module qsearch.utils*), 91
initial_layer() (*qsearch.gatesets.Gateset method*), 55
initial_layer() (*qsearch.gatesets.QubitCNOTAdjacencyList method*), 62
initial_layer() (*qsearch.gatesets.QubitCNOTLinear method*), 57
initial_layer() (*qsearch.gatesets.QubitCNOTRing method*), 58
initial_layer() (*qsearch.gatesets.QubitCZLinear method*), 59
initial_layer() (*qsearch.gatesets.QubitISwapLinear method*), 60
initial_layer() (*qsearch.gatesets.QubitXXLinear method*), 61

initial_layer() (*qsearch.gatesets.QutritCNOTLinear method*), 64
initial_layer() (*qsearch.gatesets.QutritCPIPhaseLinear method*), 63
initial_layer() (*qsearch.gatesets.U3CNOTLinear method*), 56
initial_layer() (*qsearch.gatesets.ZXZXZCNOTLinear method*), 56
initialize_APOSMM() (*in module qsearch.persistent_aposmm*), 77
inserting() (*qsearch.gates.ProductGate method*), 52
inserting() (*qsearch.ProductGate method*), 125
ISwapGate (*class in qsearch*), 116
ISwapGate (*class in qsearch.gates*), 43

K

KroneckerGate (*class in qsearch*), 123
KroneckerGate (*class in qsearch.gates*), 50

L

LeapCompiler (*class in qsearch.leap_compiler*), 67
LEAPReoptimizing_PostProcessor (*class in qsearch.post_processing*), 79
LEAPReoptimizing_PostProcessor (*in module qsearch.post_processing*), 78
LeastSquares_Jac_Solver (*class in qsearch.solvers*), 85
linear_topology() (*in module qsearch.gatesets*), 64
load() (*qsearch.Options method*), 96
load() (*qsearch.options.Options method*), 72
Logger (*class in qsearch.logging*), 68
logical_or (*in module qsearch.unitaries*), 85, 88
logprint() (*qsearch.logging.Logger method*), 68
LokyParallelizer (*class in qsearch.parallelizers*), 74
LokyParallelizer (*in module qsearch.parallelizers*), 72

M

make_required() (*qsearch.Options method*), 95
make_required() (*qsearch.options.Options method*), 72
manually_entered() (*qsearch.Options method*), 95
manually_entered() (*qsearch.options.Options method*), 71
map_steps() (*qsearch.parallelizers.MPIParallelizer method*), 74
mat_jac() (*qsearch.Gate method*), 97
mat_jac() (*qsearch.gates.Gate method*), 24
mat_jac() (*qsearch.gates.KroneckerGate method*), 50
mat_jac() (*qsearch.gates.ProductGate method*), 51
mat_jac() (*qsearch.gates.SingleQutritGate method*), 37
mat_jac() (*qsearch.gates.U1Gate method*), 36
mat_jac() (*qsearch.gates.U2Gate method*), 35
mat_jac() (*qsearch.gates.U3Gate method*), 33
mat_jac() (*qsearch.gates.XGate method*), 26
mat_jac() (*qsearch.gates.XZXZGate method*), 32
mat_jac() (*qsearch.gates.YGate method*), 28
mat_jac() (*qsearch.gates.ZGate method*), 29
mat_jac() (*qsearch.gates.ZXZXZGate method*), 31
mat_jac() (*qsearch.KroneckerGate method*), 123
mat_jac() (*qsearch.ProductGate method*), 124
mat_jac() (*qsearch.SingleQutritGate method*), 110
mat_jac() (*qsearch.U1Gate method*), 109
mat_jac() (*qsearch.U2Gate method*), 108
mat_jac() (*qsearch.U3Gate method*), 106
mat_jac() (*qsearch.XGate method*), 99
mat_jac() (*qsearch.XZXZGate method*), 105
mat_jac() (*qsearch.YGate method*), 101
mat_jac() (*qsearch.ZGate method*), 102
mat_jac() (*qsearch.ZXZXZGate method*), 104
matrix() (*qsearch.CNOTGate method*), 114
matrix() (*qsearch.CNOTRootGate method*), 122
matrix() (*qsearch.CPIGate method*), 112
matrix() (*qsearch.CPIPhaseGate method*), 113
matrix() (*qsearch.CSUMGate method*), 111
matrix() (*qsearch.CUGate method*), 121
matrix() (*qsearch.CZGate method*), 115
matrix() (*qsearch.Gate method*), 97
matrix() (*qsearch.gates.CNOTGate method*), 41
matrix() (*qsearch.gates.CNOTRootGate method*), 49
matrix() (*qsearch.gates.CPIGate method*), 39
matrix() (*qsearch.gates.CPIPhaseGate method*), 40
matrix() (*qsearch.gates.CSUMGate method*), 38
matrix() (*qsearch.gates.CUGate method*), 48
matrix() (*qsearch.gates.CZGate method*), 42
matrix() (*qsearch.gates.Gate method*), 24
matrix() (*qsearch.gates.IdentityGate method*), 25
matrix() (*qsearch.gates.ISwapGate method*), 43
matrix() (*qsearch.gates.KroneckerGate method*), 50
matrix() (*qsearch.gates.NonadjacentCNOTGate method*), 45
matrix() (*qsearch.gates.ProductGate method*), 51
matrix() (*qsearch.gates.SingleQutritGate method*), 37
matrix() (*qsearch.gates.SXGate method*), 30
matrix() (*qsearch.gates.U1Gate method*), 36
matrix() (*qsearch.gates.U2Gate method*), 35
matrix() (*qsearch.gates.U3Gate method*), 33
matrix() (*qsearch.gates.UGate method*), 46
matrix() (*qsearch.gates.UpgradedConstantGate method*), 47
matrix() (*qsearch.gates.XGate method*), 26
matrix() (*qsearch.gates.XXGate method*), 44
matrix() (*qsearch.gates.XZXZGate method*), 32
matrix() (*qsearch.gates.YGate method*), 27
matrix() (*qsearch.gates.ZGate method*), 29
matrix() (*qsearch.gates.ZXZXZGate method*), 31
matrix() (*qsearch.IdentityGate method*), 98
matrix() (*qsearch.ISwapGate method*), 116
matrix() (*qsearch.KroneckerGate method*), 123

matrix() (*qsearch.NonadjacentCNOTGate method*), 118
matrix() (*qsearch.ProductGate method*), 124
matrix() (*qsearch.SingleQutritGate method*), 110
matrix() (*qsearch.SXGate method*), 103
matrix() (*qsearch.U1Gate method*), 109
matrix() (*qsearch.U2Gate method*), 108
matrix() (*qsearch.U3Gate method*), 106
matrix() (*qsearch.UGate method*), 119
matrix() (*qsearch.UpgradedConstantGate method*), 120
matrix() (*qsearch.XGate method*), 99
matrix() (*qsearch.XXGate method*), 117
matrix() (*qsearch.XZXZGate method*), 105
matrix() (*qsearch.YGate method*), 100
matrix() (*qsearch.ZGate method*), 102
matrix() (*qsearch.ZXZXZGate method*), 104
matrix_distance() (*in module qsearch.comparison*), 15
matrix_distance_squared (*in module qsearch.utils*), 89
matrix_distance_squared() (*in module qsearch.comparison*), 14
matrix_distance_squared_jac (*in module qsearch.utils*), 89
matrix_distance_squared_jac() (*in module qsearch.comparison*), 15
matrix_kron() (*in module qsearch.utils*), 91
matrix_product() (*in module qsearch.utils*), 91
matrix_residuals (*in module qsearch.utils*), 90
matrix_residuals() (*in module qsearch.comparison*), 15
matrix_residuals_blacklist() (*in module qsearch.comparison*), 15
matrix_residuals_blacklist_jac() (*in module qsearch.comparison*), 15
matrix_residuals_jac (*in module qsearch.utils*), 90
matrix_residuals_jac() (*in module qsearch.comparison*), 15
matrix_residuals_slice() (*in module qsearch.comparison*), 15
matrix_residuals_slice_jac() (*in module qsearch.comparison*), 15
matrix_residuals_v2() (*in module qsearch.comparison*), 15
matrix_residuals_v2_jac() (*in module qsearch.comparison*), 15
MatrixDistanceObjective (*class in qsearch.objectives*), 70
mirogate (*in module qsearch.advanced_unitaries*), 7
module
    qsearch, 7
    qsearch.advanced_unitaries, 7
    qsearch.assemblers, 8
    qsearch.backends, 10
    qsearch.checkpoints, 11
    qsearch.comparison, 14
    qsearch.compiler, 16
    qsearch.defaults, 17
    qsearch.evaluation, 19
    qsearch.gates, 21
    qsearch.gatesets, 53
    qsearch.heuristics, 64
    qsearch.integrations, 65
    qsearch.leap_compiler, 66
    qsearch.logging, 68
    qsearch.multistart_solvers, 68
    qsearch.objectives, 69
    qsearch.options, 70
    qsearch.parallelizers, 72
    qsearch.persistent_aposmm, 75
    qsearch.post_processing, 78
    qsearch.project, 80
    qsearch.solvers, 83
    qsearch.unitaries, 85
    qsearch.utils, 89
MPI (*in module qsearch.parallelizers*), 73
MPI (*in module qsearch.project*), 80
MPI (*in module qsearch.utils*), 91
mpi_do_work() (*in module qsearch.utils*), 92
mpi_rank() (*in module qsearch.utils*), 92
mpi_worker() (*in module qsearch.utils*), 92
MPIParallelizer (*class in qsearch.parallelizers*), 74
MPIParallelizer (*in module qsearch.parallelizers*), 72
MultiprocessingParallelizer (*class in qsearch.parallelizers*), 74
MultiprocessingParallelizer (*in module qsearch.parallelizers*), 72
MultiStart_Solver (*class in qsearch.multistart_solvers*), 68

N

NaiveMultiStart_Solver (*class in qsearch.multistart_solvers*), 69
native_from_object (*in module qsearch*), 97
native_from_object (*in module qsearch.gates*), 24
NativeBackend (*class in qsearch.backends*), 11
NativeBackend (*in module qsearch.backends*), 10
nearest_unitary() (*in module qsearch.utils*), 91
NonadjacentCNOTGate (*class in qsearch*), 118
NonadjacentCNOTGate (*class in qsearch.gates*), 45
NOTBEGUN (*qsearch.project.Project_Status attribute*), 80

O

Objective (*class in qsearch.objectives*), 69
op_norm() (*in module qsearch.utils*), 91
optimize_worker() (*in module qsearch.multistart_solvers*), 68
Options (*class in qsearch*), 95

Options (*class in qsearch.options*), 71

P

Parallelizer (*class in qsearch.parallelizers*), 74
ParameterTuning_PostProcessor (*class in qsearch.post_processing*), 79
ParameterTuning_PostProcessor (*in module qsearch.post_processing*), 78
pauli_x (*in module qsearch.unitaries*), 88
pauli_y (*in module qsearch.unitaries*), 88
pauli_z (*in module qsearch.unitaries*), 88
peres (*in module qsearch.unitaries*), 85, 88
post_process() (*qsearch.Project method*), 127
post_process() (*qsearch.project.Project method*), 81
post_process_circuit() (*qsearch.post_processing.BasicSingleQubitReduction_PostProcessor method*), 78
post_process_circuit() (*qsearch.post_processing.LEAPReoptimizing_PostProcessor method*), 79
post_process_circuit() (*qsearch.post_processing.ParameterTuning_PostProcessor method*), 79
post_process_circuit() (*qsearch.post_processing.PostProcessor method*), 78
PostProcessor (*class in qsearch.post_processing*), 78
prepare_circuit() (*qsearch.backends.Backend method*), 11
prepare_circuit() (*qsearch.backends.NativeBackend method*), 11
prepare_circuit() (*qsearch.backends.PythonBackend method*), 11
prepare_circuit() (*qsearch.backends.SmartDefaultBackend method*), 11
process_initializer() (*in module qsearch.parallelizers*), 73
ProcessPoolParallelizer (*class in qsearch.parallelizers*), 74
ProcessPoolParallelizer (*in module qsearch.parallelizers*), 72
ProductGate (*class in qsearch*), 124
ProductGate (*class in qsearch.gates*), 51
PROGRESS (*qsearch.project.Project_Status attribute*), 80
Project (*class in qsearch*), 126
Project (*class in qsearch.project*), 80
Project_Status (*class in qsearch.project*), 80
PythonBackend (*class in qsearch.backends*), 11
PythonBackend (*in module qsearch.backends*), 10

Q

q1_unitary() (*in module qsearch.utils*), 91
qft (*in module qsearch.unitaries*), 86
qft() (*in module qsearch.unitaries*), 89
qiskit (*in module qsearch.integrations*), 66
qiskit_to_qsearch() (*in module qsearch.integrations*), 66
QiskitGateConverter (*class in qsearch.integrations*), 66
QiskitImportError, 66
qsearch
    module, 7
qsearch.advanced_unitaries
    module, 7
qsearch.assemblers
    module, 8
qsearch.backends
    module, 10
qsearch.checkpoints
    module, 11
qsearch.comparison
    module, 14
qsearch.compiler
    module, 16
qsearch.defaults
    module, 17
qsearch.evaluation
    module, 19
qsearch.gates
    module, 21
qsearch.gatesets
    module, 53
qsearch.heuristics
    module, 64
qsearch.integrations
    module, 65
qsearch.leap_compiler
    module, 66
qsearch.logging
    module, 68
qsearch.multistart_solvers
    module, 68
qsearch.objectives
    module, 69
qsearch.options
    module, 70
qsearch.parallelizers
    module, 72
qsearch.persistent_aposmm
    module, 75
qsearch.post_processing
    module, 78
qsearch.project
    module, 80
qsearch.solvers
    module, 83
qsearch.unitaries
    module, 85

qsearch.utils
    module, 89
qt_arb_rot() (*in module qsearch.utils*), 91
QubitCNOTAdjacencyList (*class in qsearch.gatesets*), 62
QubitCNOTAdjacencyList (*in module qsearch.gatesets*), 53
QubitCNOTLinear (*class in qsearch.gatesets*), 57
QubitCNOTLinear (*in module qsearch.gatesets*), 53
QubitCNOTRing (*class in qsearch.gatesets*), 58
QubitCNOTRing (*in module qsearch.gatesets*), 53
QubitCZLinear (*class in qsearch.gatesets*), 59
QubitISwapLinear (*class in qsearch.gatesets*), 60
QubitXXLinear (*class in qsearch.gatesets*), 61
QutritCNOTLinear (*class in qsearch.gatesets*), 63
QutritCNOTLinear (*in module qsearch.gatesets*), 53
QutritCPIPhaseLinear (*class in qsearch.gatesets*), 63
QutritCPIPhaseLinear (*in module qsearch.gatesets*), 53

R

random_near_identity() (*in module qsearch.utils*), 92
re_rot_z() (*in module qsearch.utils*), 91
re_rot_z_jac() (*in module qsearch.utils*), 91
recover() (*qsearch.checkpoints.Checkpoint method*), 12
recover() (*qsearch.checkpoints.ChildCheckpoint method*), 13
recover() (*qsearch.checkpoints.FileCheckpoint method*), 12
recover_parent() (*qsearch.checkpoints.ChildCheckpoint method*), 13
remap (*in module qsearch.utils*), 90
remap() (*in module qsearch.utils*), 92
remove_compilation() (*qsearch.Project method*), 126
remove_compilation() (*qsearch.project.Project method*), 81
remove_defaults() (*qsearch.Options method*), 96
remove_defaults() (*qsearch.options.Options method*), 72
remove_smart_defaults() (*qsearch.Options method*), 96
remove_smart_defaults() (*qsearch.options.Options method*), 72
reset() (*qsearch.Project method*), 126
reset() (*qsearch.project.Project method*), 81
residuals_blacklist() (*in module qsearch.evaluation*), 21
residuals_blacklist_jac() (*in module qsearch.evaluation*), 21
residuals_difference() (*in module qsearch.evaluation*), 20
residuals_difference_jac() (*in module qsearch.evaluation*), 20
residuals_product() (*in module qsearch.evaluation*), 20
residuals_product_jac() (*in module qsearch.evaluation*), 20
residuals_slice() (*in module qsearch.evaluation*), 21
residuals_slice_jac() (*in module qsearch.evaluation*), 21
residuals_with_initial_state() (*in module qsearch.comparison*), 16
residuals_with_initial_state_jac() (*in module qsearch.comparison*), 16
rot_x (*in module qsearch.unitaries*), 85
rot_x() (*in module qsearch.unitaries*), 89
rot_x_jac (*in module qsearch.unitaries*), 85
rot_x_jac() (*in module qsearch.unitaries*), 89
rot_y (*in module qsearch.unitaries*), 85
rot_y() (*in module qsearch.unitaries*), 89
rot_y_jac (*in module qsearch.unitaries*), 86
rot_y_jac() (*in module qsearch.unitaries*), 89
rot_z (*in module qsearch.unitaries*), 86
rot_z() (*in module qsearch.unitaries*), 88
rot_z_jac (*in module qsearch.unitaries*), 86
rot_z_jac() (*in module qsearch.unitaries*), 88
run() (*qsearch.Project method*), 127
run() (*qsearch.project.Project method*), 81
RUST_ENABLED (*in module qsearch.backends*), 11

S

save() (*qsearch.checkpoints.Checkpoint method*), 12
save() (*qsearch.checkpoints.ChildCheckpoint method*), 13
save() (*qsearch.checkpoints.FileCheckpoint method*), 12
save() (*qsearch.Options method*), 96
save() (*qsearch.options.Options method*), 72
save_parent() (*qsearch.checkpoints.ChildCheckpoint method*), 13
search_layers() (*qsearch.gatesets.Gateset method*), 55
search_layers() (*qsearch.gatesets.QubitCNOTAdjacencyList method*), 63
search_layers() (*qsearch.gatesets.QubitCNOTLinear method*), 57
search_layers() (*qsearch.gatesets.QubitCNOTRing method*), 59
search_layers() (*qsearch.gatesets.QubitCZLinear method*), 59
search_layers() (*qsearch.gatesets.QubitISwapLinear method*), 60
search_layers() (*qsearch.gatesets.QubitXXLinear method*), 61
search_layers() (*qsearch.gatesets.QutritCNOTLinear method*), 64
search_layers() (*qsearch.gatesets.QutritCPIPhaseLinear method*), 63

search_layers() (*qsearch.gatesets.U3CNOTLinear method*), 57
search_layers() (*qsearch.gatesets.ZXZXZCNOTLinear method*), 56
SearchCompiler (*class in qsearch*), 96
SearchCompiler (*class in qsearch.compiler*), 16
SequentialParallelizer (*class in qsearch.parallelizers*), 74
SequentialParallelizer (*in module qsearch.parallelizers*), 72
set_defaults() (*qsearch.Options method*), 95
set_defaults() (*qsearch.options.Options method*), 71
set_defaults() (*qsearch.Project method*), 127
set_defaults() (*qsearch.project.Project method*), 81
set_smart_defaults() (*qsearch.Options method*), 95
set_smart_defaults() (*qsearch.options.Options method*), 72
set_smart_defaults() (*qsearch.Project method*), 127
set_smart_defaults() (*qsearch.project.Project method*), 81
single_task() (*in module qsearch.parallelizers*), 73
SingleQutritGate (*class in qsearch*), 110
SingleQutritGate (*class in qsearch.gates*), 37
SmartDefaultBackend (*class in qsearch.backends*), 11
SmartDefaultBackend (*in module qsearch.backends*), 10
solve_circuits_parallel() (*qsearch.parallelizers.LokyParallelizer method*), 74
solve_circuits_parallel() (*qsearch.parallelizers.MPIParallelizer method*), 74
solve_circuits_parallel() (*qsearch.parallelizers.MultiprocessingParallelizer method*), 74
solve_circuits_parallel() (*qsearch.parallelizers.Parallelizer method*), 74
solve_circuits_parallel() (*qsearch.parallelizers.ProcessPoolParallelizer method*), 74
solve_circuits_parallel() (*qsearch.parallelizers.SequentialParallelizer method*), 75
solve_for_unitary() (*qsearch.multistart_solvers.MultiStart_Solver method*), 69
solve_for_unitary() (*qsearch.multistart_solvers.NaiveMultiStart_Solver method*), 69
solve_for_unitary() (*qsearch.solvers.BFGS_Jac_Solver method*), 84
solve_for_unitary() (*qsearch.solvers.CMA_Solver method*), 84
solve_for_unitary() (*qsearch.solvers.COBYLA_Solver method*), 84
solve_for_unitary() (*qsearch.solvers.DIY_Solver method*), 84
solve_for_unitary() (*qsearch.solvers.LeastSquares_Jac_Solver method*), 85
solve_for_unitary() (*qsearch.solvers.Solver method*), 84
Solver (*class in qsearch.solvers*), 84
sqrt_cnot (*in module qsearch.unitaries*), 85, 88
sqrt_x (*in module qsearch.unitaries*), 88
standard_defaults (*in module qsearch*), 96
standard_defaults (*in module qsearch.defaults*), 17, 19
standard_smart_defaults (*in module qsearch*), 96
standard_smart_defaults (*in module qsearch.defaults*), 17, 19
stateprep_defaults (*in module qsearch.defaults*), 17, 19
stateprep_initial_state() (*in module qsearch.defaults*), 19
stateprep_smart_defaults (*in module qsearch.defaults*), 19
stateprep_target() (*in module qsearch.defaults*), 19
StateprepObjective (*class in qsearch.objectives*), 70
status() (*qsearch.Project method*), 127
status() (*qsearch.project.Project method*), 82
SubCompiler (*class in qsearch.leap_compiler*), 67
successors() (*qsearch.gatesets.Gateset method*), 55
successors() (*qsearch.gatesets.QubitCNOTLinear method*), 58
successors() (*qsearch.gatesets.QubitCZLinear method*), 60
successors() (*qsearch.gatesets.QubitISwapLinear method*), 61
successors() (*qsearch.gatesets.QubitXXLinear method*), 62
swap (*in module qsearch.unitaries*), 85, 88
SXGate (*class in qsearch*), 103
SXGate (*class in qsearch.gates*), 30

T

toffoli (*in module qsearch.unitaries*), 85, 88

U

U1Gate (*class in qsearch*), 109
U1Gate (*class in qsearch.gates*), 36
U2Gate (*class in qsearch*), 107
U2Gate (*class in qsearch.gates*), 34
U3CNOTLinear (*class in qsearch.gatesets*), 56
U3CNOTLinear (*in module qsearch.gatesets*), 53
U3Gate (*class in qsearch*), 106

U3Gate (*class in qsearch.gates*), 33
UGate (*class in qsearch*), 119
UGate (*class in qsearch.gates*), 46
update() (*qsearch.Options method*), 95
update() (*qsearch.options.Options method*), 71
update_history_dist() (*in module qsearch.persistent_aposmm*), 76
updated() (*qsearch.Options method*), 95
updated() (*qsearch.options.Options method*), 71
upgrade_qudits (*in module qsearch.utils*), 90
upgrade_qudits() (*in module qsearch.utils*), 92
UpgradedConstantGate (*class in qsearch*), 120
UpgradedConstantGate (*class in qsearch.gates*), 47

V

validate_structure() (*qsearch.Gate method*), 98
validate_structure() (*qsearch.gates.Gate method*), 25
validate_structure() (*qsearch.gates.KroneckerGate method*), 51
validate_structure() (*qsearch.gates.NonadjacentCNOTGate method*), 46
validate_structure() (*qsearch.gates.ProductGate method*), 53
validate_structure() (*qsearch.KroneckerGate method*), 124
validate_structure() (*qsearch.NonadjacentCNOTGate method*), 119
validate_structure() (*qsearch.ProductGate method*), 126

X

XGate (*class in qsearch*), 99
XGate (*class in qsearch.gates*), 26
XXGate (*class in qsearch*), 117
XXGate (*class in qsearch.gates*), 44
XZXZGate (*class in qsearch*), 105
XZXZGate (*class in qsearch.gates*), 32

Y

YGate (*class in qsearch*), 100
YGate (*class in qsearch.gates*), 27

Z

ZGate (*class in qsearch*), 101
ZGate (*class in qsearch.gates*), 28
ZXZXZCNOTLinear (*class in qsearch.gatesets*), 56
ZXZXZCNOTLinear (*in module qsearch.gatesets*), 53
ZXZXZGate (*class in qsearch*), 104
ZXZXZGate (*class in qsearch.gates*), 31