

---

# **qsearch**

***Release 2.0.0***

**Marc Grau Davis, Ethan Smith**

**Jan 13, 2022**



**CONTENTS:**

<b>1</b>	<b>Gatesets in qsearch</b>	<b>3</b>
1.1	Provided Gatesets . . . . .	3
1.2	Custom Gatesets . . . . .	4
<b>2</b>	<b>Gates in qsearch</b>	<b>5</b>
2.1	Provided Gates . . . . .	5
2.2	Custom Gates . . . . .	5
<b>3</b>	<b>API Reference</b>	<b>7</b>
3.1	qsearch . . . . .	7
<b>4</b>	<b>Working with nonlinear topologies</b>	<b>113</b>
<b>5</b>	<b>Working with nonstandard gates or qutrits</b>	<b>115</b>
<b>6</b>	<b>Customizing your compilation</b>	<b>117</b>
<b>7</b>	<b>Indices and tables</b>	<b>119</b>
	<b>Python Module Index</b>	<b>121</b>
	<b>Index</b>	<b>123</b>



---

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

---



## 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 topology 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:

- `initial_layer(n)` The single input, `n`, is an integer which describes 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 describes 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.



## 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 create 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 assemble the circuit to OpenQASM or Qiskit.

- `UGate` - represents the gate described 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.**

## API REFERENCE

This page contains auto-generated API reference documentation<sup>1</sup>.

### 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

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

##### Attributes

<code>mirogate</code>
<code>HHL</code>

`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()`

`qsearch.advanced_unitaries.HHL`

---

<sup>1</sup> Created with sphinx-autoapi

## 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.ASEMBLER\_QISKIT

Outputs Python code that generates a Qiskit circuit object.

### qsearch.assemblers.ASEMBLER\_OPENQASM

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

### qsearch.assemblers.ASEMBLER\_IBMOPENQASM

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

### qsearch.assemblers.ASEMBLER\_QUTRIT

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

## Module Contents

### Classes

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

### Functions

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

### Attributes

<i>assemblydict_qiskit</i>
<i>assemblydict_openqasm</i>
<i>assemblydict_ibmopenqasm</i>
<i>assemblydict_qutrit</i>

continues on next page

Table 5 – continued from previous page

---

*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**(*self, 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 “structure” 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**(*self, 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 “structure” 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

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

### Attributes

<i>RUST_ENABLED</i>
---------------------

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(self, 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**(self, 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**(self, 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**(self, 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

<a href="#">Checkpoint</a>	This class is used for storing intermediate state while compiling, to allow an interrupted compilation to resume at a later time.
<a href="#">FileCheckpoint</a>	This Checkpoint will store the state in the file specified in the options as statefile.
<a href="#">ChildCheckpoint</a>	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**(self, 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**(self)

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**(*self*)

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**(*self*, 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**(*self*)

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**(*self*)

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 hierarchical 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 of a `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**(*self*, *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**(*self*, *parentstate*)

Saves the parentstate alongside the child state.

**recover**(*self*)

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**(*self*)

Recovers the parentstate.

**delete**(*self*)

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

**delete\_parent**(*self*)

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

<i>matrix_distance_squared</i> (A, B)	This is a distance function used to compare two matrices. It is phase agnostic and fast to calculate.
<i>matrix_distance</i> (A, B)	The square root of matrix_distance_squared is more analogous to "distance", although for most purposes, working with a distance squared is fine, since inequalities hold.
<i>matrix_distance_squared_jac</i> (U, M, J)	The jacobian version of matrix_distance_squared.
<i>matrix_residuals</i> (A, B, I)	
<i>matrix_residuals_jac</i> (U, M, J)	
<i>matrix_residuals_v2</i> (A, B, I)	
<i>matrix_residuals_v2_jac</i> (U, M, J)	
<i>matrix_residuals_slice</i> (slices, A, B, I)	

continues on next page

Table 9 – continued from previous page

<code>matrix_residuals_slice_jac</code> (slices, A, B, J)
<code>matrix_residuals_blacklist</code> (badrows, badcols, A, B, I)
<code>matrix_residuals_blacklist_jac</code> (badrows, badcols, A, B, J)
<code>distance_with_initial_state</code> (stateA, stateB, A, B)
<code>distance_with_initial_state_jac</code> (stateA, stateB, A, B, J)
<code>residuals_with_initial_state</code> (stateA, stateB, A, B, I)
<code>residuals_with_initial_state_jac</code> (stateA, stateB, U, M, J)
<code>eval_func_from_residuals</code> (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 analogous 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 numpy 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 respect 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

<i>Compiler</i>	This class defines the pattern for compilers that convert a unitary matrix to a circuit that implements that matrix.
<i>SearchCompiler</i>	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**(self, 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 upper 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**(*self*, *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)

---

continues on next page

Table 11 – continued from previous page

<i>default_error_residuals_jac</i> (options)
<i>default_objective</i> (options)
<i>stateprep_initial_state</i> (options)
<i>stateprep_target</i> (options)
<i>default_compiler</i> (options)
<i>identity</i> (U)

**Attributes**

<i>standard_defaults</i>
<i>standard_smart_defaults</i>
<i>stateprep_defaults</i>
<i>stateprep_smart_defaults</i>

```

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 standardized 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, options)

---

*error\_stateprep\_distsq*(circuit, parameters, target, options)

---

*error\_stateprep\_distsq\_jac*(circuit, parameters, target, options)

---

*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, options)

---

*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, options)

---

*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, jacs, options)

---

*cost\_combo\_linear*(circuit, parameters, target, options)

---

*cost\_combo\_linear\_jac*(circuit, parameters, target, jacs, options)

---

```

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

<i>Gate</i>	This class shows the framework for working with quantum gates in Qsearch.
<i>IdentityGate</i>	Represents an identity gate of any number of qudits of any size.
<i>XGate</i>	Represents a parameterized X rotation on one qubit.
<i>YGate</i>	Represents a parameterized Y rotation on one qubit.
<i>ZGate</i>	Represents a parameterized Z rotation on one qubit.

continues on next page

Table 14 – continued from previous page

<i>SXGate</i>	Represents a $\sqrt{X}$ rotation on one qubit, which is equivalent to <code>XGate()</code> with a parameter of $\pi/2$ , up to an overall phase.
<i>ZXZXGate</i>	Represents an arbitrary parameterized single-qubit gate, decomposed into 3 parameterized Z gates separated by $X(\pi/2)$ gates.
<i>XZXGate</i>	Represents a partially parameterized single qubit gate, equivalent to <code>ZXZXGate</code> 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.
<i>U3Gate</i>	Represents an arbitrary parameterized single qubit gate, parameterized in the same way as IBM's U3 gate.
<i>U2Gate</i>	Represents a parameterized single qubit gate, parameterized in the same way as IBM's U2 gate.
<i>U1Gate</i>	Represents an parameterized single qubit gate, parameterized in the same way as IBM's U1 gate.
<i>SingleQutritGate</i>	This gate represents an arbitrary parameterized single-qutrit gate.
<i>CSUMGate</i>	Represents the constant two-qutrit gate CSUM
<i>CPIGate</i>	Represents the constant two-qutrit gate CPI.
<i>CPIPhaseGate</i>	Represents the constant two-qutrit gate CPI with phase differences.
<i>CNOTGate</i>	Represents the constant two-qubit gate CNOT.
<i>CZGate</i>	Represents the constant two-qubit gate Controlled-Z.
<i>ISwapGate</i>	Represents the constant two-qubit gate ISwap.
<i>XXGate</i>	Represents the constant two-qubit gate $XX(\pi/2)$ .
<i>NonadjacentCNOTGate</i>	Represents the two-qubit gate CNOT, but between two qubits that are not necessarily next to each other.
<i>UGate</i>	Represents an arbitrary constant gate, defined by the unitary passed to the initializer.
<i>UpgradedConstantGate</i>	Represents a constant gate, based on the Gate passed to its initializer, but upgraded to act on qudits of a larger size.
<i>CUGate</i>	Represents an arbitrary controlled gate, defined by the unitary passed to the initializer.
<i>CNOTRootGate</i>	Represents the $\sqrt{\text{CNOT}}$ gate. Two $\sqrt{\text{CNOT}}$ gates in a row will form a CNOT gate.
<i>KroneckerGate</i>	Represents the Kronecker product of a list of gates. This is equivalent to performing those gate in parallel in a quantum circuit.
<i>ProductGate</i>	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**(*self*, *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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

##### **mat\_jac**(*self*, *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 *i*th 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**(*self*, *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"*, *gate\_name*, (*\*gate\_parameters*), (*\*gate\_indices*))

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__(self, other)`  
Return self==value.

`__hash__(self)`  
Return hash(self).

`copy(self)`

`_parts(self)`

`__copy__(self)`

`__deepcopy__(self, memo)`

`__repr__(self)`  
Return repr(self).

`validate_structure(self)`

**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**(*self, 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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** `np.ndarray`

**assemble**(*self, 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\_\_(self)**

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(self, 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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**mat\_jac(self, 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 *i*th 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(self, 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, XZZXGate, CNOTGate, and NonadjacentCNOTGate.

**\_\_repr\_\_**(*self*)

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**(*self*, *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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**mat\_jac**(*self*, *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 *i*th 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**(*self*, *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”, *gate*name, (\**gate*parameters), (\**gate*indices))

Where *gate*name corresponds to a gate that an Assembler will recognize, *gate*parameters corresponds to the parameters for the specified gate (usually but not always calculated from *v*), and *gate*indices 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\_\_(self)**

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**(self, *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  $d \times d$  where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**mat\_jac**(self, *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 *i*th 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**(self, *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\_\_(self)**

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(self, 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(self, 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, ZXZXZGate, CNOTGate, and NonadjacentCNOTGate.

**\_\_repr\_\_(self)**

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(self, 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  $d \times d$  where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**mat\_jac(self, 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(self, 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\_\_**(*self*)

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**(*self*, *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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**mat\_jac**(*self*, *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 *i*th 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**(*self*, *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, XZZXGate, CNOTGate, and NonadjacentCNOTGate.

```
__repr__(self)
```

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(self, 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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

```
mat_jac(self, 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 *i*th 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(self, 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\_\_(self, other)**  
Return self==value.

**\_\_repr\_\_(self)**  
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(self, 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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**mat\_jac(self, 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 *i*th 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(self, 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`, `XZZZGate`, `CNOTGate`, and `NonadjacentCNOTGate`.

```
__eq__(self, other)
    Return self==value.

__repr__(self)
    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**(*self*, *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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**mat\_jac**(*self*, *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 *i*th 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**(*self*, *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\_\_**(*self*, *other*)

Return self==value.

**\_\_repr\_\_**(*self*)

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**(*self*, *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**(*self*, *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 *i*th 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**(*self*, *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`, `XZZZGate`, `CNOTGate`, and `NonadjacentCNOTGate`.

**\_\_repr\_\_**(*self*)

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**(*self*, *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**(*self*, *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`, `XZZZGate`, `CNOTGate`, and `NonadjacentCNOTGate`.

**\_\_repr\_\_**(*self*)

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**(*self*, *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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** `np.ndarray`

**assemble**(*self*, *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\_\_(self)**

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(self, 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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**assemble(self, 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\_\_(self)**

Return repr(self).

**class** qsearch.gates.CNOTGateBases: [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\_\_**(*self*, *other*)Return `self==value`.**matrix**(*self*, *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**(*self*, *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\_\_**(*self*)Return `repr(self)`.**class** qsearch.gates.CZGateBases: [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\_\_(self, other)**

Return self==value.

**matrix(self, 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  $d \times d$ , where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**assemble(self, 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"`, *gate*name, (\**gate*parameters), (\**gate*indices))

Where *gate*name corresponds to a gate that an Assembler will recognize, *gate*parameters corresponds to the parameters for the specified gate (usually but not always calculated from v), and *gate*indices 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, ZXZXGate, CNOTGate, and NonadjacentCNOTGate.

**\_\_repr\_\_(self)**

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\_\_(self, other)**

Return self==value.

**matrix**(*self*, *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**(*self*, *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\_\_**(*self*)

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\_\_**(*self*, *other*)

Return *self*==*value*.

**matrix**(*self*, *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**(*self*, *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`, `XZZZGate`, `CNOTGate`, and `NonadjacentCNOTGate`.

**\_\_repr\_\_**(*self*)

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**(*self*, *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  $d \times d$  where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**assemble**(*self*, *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\_\_**(*self*)

Return repr(*self*).

**validate\_structure**(*self*)

**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**(*self*, *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**(*self*, *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, XZZZGate, CNOTGate, and NonadjacentCNOTGate.

**\_\_repr\_\_**(*self*)

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**(*self*, *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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**assemble**(*self*, *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\_\_(self)**  
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(self, 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(self, 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\_\_(self)**  
Return repr(self).

**class** qsearch.gates.CNOTRootGateBases: [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**(*self*, *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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**assemble**(*self*, *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__`(*self*)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**(*self*, *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  $d \times d$ , where `d` is the intended qudit size (`d` is 2 for qubits, 3 for qutrits, etc.)

**Return type** `np.ndarray`

**mat\_jac**(*self*, *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 `i`th 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**(*self*, *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"`, *gate*name, (\**gate*parameters), (\**gate*indices))

Where *gate*name corresponds to a gate that an Assembler will recognize, *gate*parameters corresponds to the parameters for the specified gate (usually but not always calculated from *v*), and *gate*indices 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**(*self*, *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**(*self*)

**\_\_deepcopy\_\_**(*self*, *memo*)

**\_\_repr\_\_**(*self*)

Return `repr(self)`.

**validate\_structure**(*self*)



**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**(self, 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  $d \times d$ , where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**mat\_jac**(self, 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**(self, 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**(self, \*gates)

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

**Parameters** **gates** – A list of Gates to be appended.

**inserting**(*self*, \**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 beginning of the ProductGate.

**\_\_deepcopy\_\_**(*self*, *memo*)

**\_\_repr\_\_**(*self*)

Return repr(*self*).

**validate\_structure**(*self*)

## 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 ZXZXZ 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 ZXZXZ 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

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

### Functions

<i>linear_topology</i> (double_gate, single_gate, n, d, identity_gate=None, single_alt=None, double_weight=1, single_weight=0, skip_index=None)
<i>fill_row</i> (gate, n)
<i>find_last_3_cnots_linear</i> (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**(*self*, *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**(*self*, *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**(*self*, *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 expected branching factor

**Return type** int

#### **successors**(*self*, *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 current 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__(self, 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**(self, 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**(self, 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**(self, 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**(self, 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=ZXZXZGate())

Bases: [Gateset](#)

A Gateset for working with CNOT and single-qubit gates parameterized with U3Gate and XZZZGate 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(*self*, *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(*self*, *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(*self*, *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 expected branching factor

**Return type** int

**successors(*self*, *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 current 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*=XZZZGate())

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**(self, 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**(self, 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**(self, 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**(self, 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**(self, 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 expected branching factor

**Return type** int

**successors**(*self*, *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 current 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**(*self*, *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**(*self*, *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**(*self*, *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 expected branching factor

**Return type** int

**successors**(*self*, *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 current 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**(self, 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**(self, 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**(self, 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 expected branching factor

**Return type** int

**successors**(self, 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 current 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 specify 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**(self, 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**(self, 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**(self, 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**(*self*, *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**(*self*, *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**(*self*, *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

---

<i>greedy</i> (circ, v, weight, options)	Defines a heuristic that results in greedy search, which focuses solely on minimizing the eval_func, and behaves somewhat similarly to depth first search.
<i>astar</i> (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.
<i>djikstra</i> (circ, v, weight, options)	Defines a heuristic that relies only on the weight, which guarantees 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 solely on minimizing the eval\_func, and behaves somewhat similarly to depth first search.

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 guarantees 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

---

<i>QiskitGateConverter</i>
----------------------------

---

## Functions

---

<code>qiskit_to_qsearch(circ, converter=None)</code>	Convert qiskit code to qsearch <i>structure</i> and parameters
--	--

---

## Attributes

---

<code>qiskit</code>
---------------------

---

`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**(*self*, *gate*, *qubits*, *cbits*)

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

**convert\_cx**(*self*, *gate*, *qubits*, *cbits*)

**convert\_u3**(*self*, *gate*, *qubits*, *cbits*)

**convert\_u2**(*self*, *gate*, *qubits*, *cbits*)

**convert\_rx**(*self*, *gate*, *qubits*, *cbits*)

**convert\_ry**(*self*, *gate*, *qubits*, *cbits*)

**convert\_rz**(*self*, *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

---

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

---

## Functions

---

`cut_end(circ, depth)`

---

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

**class** `qsearch.leap_compiler.LeanCompiler(options=Options())`

Bases: `qsearch.compiler.Compiler`

LeanCompiler is a more scalable search based circuit compiler

LeanCompiler uses fixed structure prefixes to greatly reduce the search space and speed up synthesis at the cost of optimality. Thus it is recommended to use in conjunction with `reoptimizing_compiler.LeanReoptimizing_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 upper 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**(*self*, *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 LeanCompiler to use.

**compile**(*self*, *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

<i>Logger</i>	This class is used to control what level of messages 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 messages get printed, and to where.

**logprint**(*self, 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

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

#### Functions

<i>distance_for_x</i> ( <i>x, options, circuit</i> )	Calculate the distance between circuit and the target for input <i>x</i> based on the distance metric
<i>optimize_worker</i> ( <i>circuit, options, q, x0, error_func</i> )	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**(*self*, *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 optimizer

**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**(*self*, *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**(*self*, *circuit*, *options*)

**gen\_error\_func**(*self*, *circuit*, *options*)

**gen\_error\_jac**(*self*, *circuit*, *options*)

**gen\_error\_residuals**(*self*, *circuit*, *options*)



```

    gen_error_residuals_jac(self, circuit, options)
class qsearch.objectives.MatrixDistanceObjective
    Bases: Objective
    gen_error_func(self, circuit, options)
    gen_error_jac(self, circuit, options)
    gen_error_residuals(self, circuit, options)
    gen_error_residuals_jac(self, circuit, options)
class qsearch.objectives.StateprepObjective
    Bases: Objective
    gen_error_func(self, circuit, options)
    gen_error_jac(self, circuit, options)
    gen_error_residuals(self, circuit, options)
    gen_error_residuals_jac(self, circuit, options)
class qsearch.objectives.BackwardsCompatibleObjective
    Bases: Objective
    gen_error_func(self, circuit, options)
    gen_error_func(self, circuit, options)
    gen_error_jac(self, circuit, options)
    gen_error_residuals(self, circuit, options)
    gen_error_residuals_jac(self, 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

---

<i>Options</i>	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**(*self, \*names*)

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

**\_\_getitem\_\_**(*self, name*)

**\_\_delitem\_\_**(*self, name*)

**\_\_getattr\_\_**(*self, name*)

**\_\_setattr\_\_**(*self, name, value*)

Implement setattr(*self, name, value*).

**\_\_contains\_\_**(*self, name*)

**manually\_entered**(*self, \*names, location='dict', operator='all'*)

**empty\_copy**(*self*)

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

**copy**(*self*)

Create a full copy of an Options object.

**\_\_copy\_\_**(*self*)

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

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

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

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

**\_update\_dict**(*self, otherdict*)

**set\_defaults**(*self, \*\*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**(*self, \*\*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**(*self, \*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**(*self, \*names*)

Removes the defaults for the specified names.

**remove\_smart\_defaults**(*self*, \*names)

Removes the smart\_defaults for the specified names.

**generate\_cache**(*self*)

Caches valuesa for all functions in smart\_defaults.

**save**(*self*, filepath=None)

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

**load**(*self*, 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\_\_**(*self*)

**\_\_setstate\_\_**(*self*, 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

<i>Parallelizer</i>	Base class for all Parallelizers. Parallelizers calculate the value of multiple search nodes in parallel.
<i>LokyParallelizer</i>	A parallelizer based on Loky, a "deadlock-free" ProcessPoolExecutor.
<i>MultiprocessingParallelizer</i>	A Parallelizer based on multiprocessing. Note this cannot be used with the MultiStart_Solvers!
<i>ProcessPoolParallelizer</i>	A Parallelizer based on concurrent.futures.ProcessPoolExecutor.
<i>MPIParallelizer</i>	A distributed MPI based Parallelizer.
<i>SequentialParallelizer</i>	A Parallelizer 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**(*self*, *tuples*)

Calculate the value of search tree nodes in parallel.

**done**(*self*)

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**(*self*, *tuples*)

Calculate the value of search tree nodes in parallel.

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

Bases: `Parallelizer`

A Parallelizer based on multiprocessing. Note this cannot be used with the MultiStart\_Solvers!

**solve\_circuits\_parallel**(*self*, *tuples*)

Calculate the value of search tree nodes in parallel.

**done(self)**  
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(self, tuples)**  
Calculate the value of search tree nodes in parallel.

**done(self)**  
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(self, tuples)**  
Calculate the value of search tree nodes in parallel.

**map\_steps(self, new\_steps)**

**done(self)**  
Finalize/Clean up any state needed to run the Parallelizer.

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

Bases: [Parallelizer](#)

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

**solve\_circuits\_parallel(self, 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

<a href="#"><i>aposmm</i></a> (H, persis_info, gen_specs, libE_info)	APOSMM coordinates multiple local optimization runs, starting from points
<a href="#"><i>update_history_dist</i></a> (H, n)	Updates distances/indices after new points that have been evaluated.
<a href="#"><i>decide_where_to_start_localopt</i></a> (H, n, n_s, rk_const, ld=0, mu=0, nu=0)	APOSMM starts a local optimization runs from a point that:
<a href="#"><i>initialize_APOSMM</i></a> (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 initializing 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., `gato/gtol` 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

<i>PostProcessor</i>	This class is used to modify circuits that have already been synthesized.
<i>BasicSingleQubitReduction_PostProcessor</i>	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.
<i>ParameterTuning_PostProcessor</i>	Attempts to reduce the eval_func value of the circuit simply by tuning the parameters better using stronger Solver parameters.
<i>LEAPReoptimizing_PostProcessor</i>	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**(*self, result, options=None*)

Processes the circuit dictionary and returns a new one.

**Parameters** **result** – A dictionary containing a synthesized circuit. Expect it to contain “struc-



ture” 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**(self, 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**(self, 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**(self, 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**(self, 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

---

<i>Project_Status</i>	Generic enumeration.
<i>Project</i>	The project class wraps most of the functionality of Qsearch as intended to help manage working with Qsearch.

---

### Attributes

---

<i>MPI</i>
------------

---

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.

**\_save**(self)

**\_checkpoint\_path**(self, name)

**add\_compilation**(self, 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\_\_**(*self*, *keyword*, *value*)

**configure\_compiler\_override**(*self*, *keyword*, *value*)

An unsafe method that allows the user to set global Project Options even if there is existing work.

**\_\_getitem\_\_**(*self*, *keyword*)

**\_\_delitem\_\_**(*self*, *keyword*)

**configure**(*self*, **\*\*dictionary**)

Adds multiple options to the global Project Options at once.

**reset**(*self*, *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**(*self*, *name*)

Removes a compilation from a Project.

**Parameters** **name** – The name of the compilation to remove

**clear**(*self*, *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\_\_**(*self*)

**\_\_exit\_\_**(*self*, *exc\_typ*, *exc\_val*, *exc\_tb*)

**set\_defaults**(*self*, *defaults*=*standard\_defaults*)

Updates the Project Options with the standard defaults from *defaults.py*, or a provided dictionary.

**set\_smart\_defaults**(*self*, *smart\_defaults*=*standard\_smart\_defaults*)

Updates the Project Options with the standard smart\_defaults from *defaults.py*, or a provided dictionary

**run**(*self*)

Runs all of the compilations in the Project.

**post\_process**(*self*, *postprocessor*, *name*=None, *options*=None, **\*\*extraargs**)

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**(*self*)

Returns a True if all compilations in the Project have finished and False otherwise.

**finish**(*self*)

Called when done running compilations in order to end MPI tasks.

**status**(*self*, *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

**property** **compilations**(*self*)

The list of names corresponding to compilations on this Project.

**\_compilation\_status**(*self*, *name*)

**\_overall\_status**(*self*)

**get\_result**(*self*, *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**(*self*, *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**(*self*, *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**(*self*, *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**(*self*, *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

<i>Solver</i>	This class is used to wrap numerical optimizers for circuit solving.
<i>CMA_Solver</i>	Uses cmaes gradient-free optimization from the cma package.
<i>COBYLA_Solver</i>	Uses cobyla gradient-free optimization from scipy.
<i>DIY_Solver</i>	An easier way to wrap a numerical optimizer than writing your own Solver class.
<i>BFGS_Jac_Solver</i>	A solver based on the BFGS implementation in scipy. It requires gradients.
<i>LeastSquares_Jac_Solver</i>	Uses the Levenberg-Marquardt least-squares optimizer in scipy.

### Functions

<i>default_solver</i> (options, x0=None)	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.

**abstract** `solve_for_unitary(self, 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__(self, other)`

Return self==value.

**property** `distance_metric(self)`

**class** `qsearch.solvers.CMA_Solver`

Bases: *Solver*

Uses cmaes gradient-free optimization from the cma package.

**solve\_for\_unitary**(self, 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**(*self*, *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**(*self*, *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**(*self*, *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.

**solve\_for\_unitary**(*self*, *circuit*, *options*, *x0=None*)

Finds the best parameters that minimize error\_func or error\_residuals between the unitary from the circuit and options.target.

**property** distance\_metric(*self*)

## 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  $n \times n$  qft matrix.

**Parameters**  $n$  (*int*) –

**qsearch.unitaries.identity**

Returns a  $n \times n$  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 = 2)

---

*arbitrary\_cnot*(qudits, control, target)

---

### Attributes

---

*cnot*

---

*sqrt\_cnot*

---

*swap*

---

*toffoli*

---

*fredkin*

---

*peres*

---

*logical\_or*

---

*full\_adder*

---

*pauli\_x*

---

*pauli\_y*

---

*pauli\_z*

---

continues on next page



Table 41 – continued from previous page

*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 product of the implemented matrix and the hermitian conjugate of the target and the identity.

`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

<code>matrix_product(*LU)</code>	Performs matrix multiplication of a list of matrices.
<code>matrix_kron(*LU)</code>	Performs the kronecker product on a list of matrices.
<code>op_norm(A)</code>	An implementation of the l1-l1 operator norm.
<code>nearest_unitary(A)</code>	Calculate the closest unitary to a given matrix.
<code>index_test(i, di, df)</code>	
<code>downgrade_qudits_residuals(di, df, A, B, I)</code>	

continues on next page

Table 42 – continued from previous page

<code>downgrade_qudits_residuals_jac</code> (di, df, A, B, J)	
<code>generate_stateprep_target_matrix</code> (state)	
<code>re_rot_z</code> (theta, old_z)	
<code>re_rot_z_jac</code> (theta, old_z, multiplier=1)	
<code>q1_unitary</code> (x)	
<code>qt_arb_rot</code> (Theta_1, Theta_2, Theta_3, Phi_1, Phi_2, Phi_3, Phi_4, Phi_5)	Using the parameterization found in <a href="https://journals.aps.org/prd/pdf/10.1103/PhysRevD.38.1994">https://journals.aps.org/prd/pdf/10.1103/PhysRevD.38.1994</a> ,
<code>random_near_identity</code> (n, alpha)	
<code>remap</code> (U, order, d=2)	
<code>upgrade_qudits</code> (U, di=2, df=3)	
<code>endian_reverse</code> (U, d=2)	
<code>mpi_rank</code> ()	
<code>mpi_do_work</code> (comm)	Do the work of a single compilation.
<code>mpi_worker</code> ()	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:

$$\begin{aligned} u_{11} &= \cos[\Theta_1] \cos[\Theta_2] \exp[i\Phi_1] & u_{12} &= \sin[\Theta_1] \exp[i\Phi_3] & u_{13} &= \cos[\Theta_1] \sin[\Theta_2] \exp[i\Phi_4] \\ u_{21} &= \sin[\Theta_2] \sin[\Theta_3] \exp[-i\Phi_4 - i\Phi_5] - \end{aligned}$$

$$\sin[\Theta_1] \cos[\Theta_2] \cos[\Theta_3] \exp[i\Phi_1 + i\Phi_2 - i\Phi_3]$$

$$\begin{aligned} u_{22} &= \cos[\Theta_1] \cos[\Theta_3] \exp[i\Phi_2] & u_{23} &= \cos[\Theta_2] \sin[\Theta_3] \exp[-i\Phi_1 - i\Phi_5] - \end{aligned}$$

$$\sin[\Theta_1] \sin[\Theta_2] \cos[\Theta_3] \exp[i\Phi_2 - i\Phi_3 + i\Phi_4]$$

$$u_{31} = -\sin[\Theta_1] \cos[\Theta_2] \sin[\Theta_3] \exp[i\Phi_1 - i\Phi_3 + i\Phi_5]$$

$$\bullet \sin[\Theta_2] \cos[\Theta_3] \exp[-i\Phi_2 - i\Phi_4]$$

$$\begin{aligned} u_{32} &= \cos[\Theta_1] \sin[\Theta_3] \exp[i\Phi_5] & u_{33} &= \cos[\Theta_2] \cos[\Theta_3] \exp[-i\Phi_1 - i\Phi_2] - \end{aligned}$$

$$\sin[\Theta_1] \sin[\Theta_2] \sin[\Theta_3] \exp[-i\Phi_3 + i\Phi_4 + i\Phi_5]$$

```
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

<i>Options</i>	This class manages options that are passed between various Qsearch objects.
<i>Compiler</i>	This class defines the pattern for compilers that convert a unitary matrix to a circuit that implements that matrix.
<i>SearchCompiler</i>	This Compiler uses an A* search strategy to synthesize a unitary, as described in the paper Towards Optimal Topology Aware Quantum Circuit Synthesis.
<i>Gate</i>	This class shows the framework for working with quantum gates in Qsearch.
<i>IdentityGate</i>	Represents an identity gate of any number of qudits of any size.
<i>XGate</i>	Represents a parameterized X rotation on one qubit.
<i>YGate</i>	Represents a parameterized Y rotation on one qubit.
<i>ZGate</i>	Represents a parameterized Z rotation on one qubit.
<i>SXGate</i>	Represents a $\sqrt{X}$ rotation on one qubit, which is equivalent to XGate() with a parameter of $\pi/2$ , up to an overall phase.
<i>ZXZXZGate</i>	Represents an arbitrary parameterized single-qubit gate, decomposed into 3 parameterized Z gates separated by X( $\pi/2$ ) gates.
<i>XZXZGate</i>	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.
<i>U3Gate</i>	Represents an arbitrary parameterized single qubit gate, parameterized in the same way as IBM's U3 gate.
<i>U2Gate</i>	Represents a parameterized single qubit gate, parameterized in the same way as IBM's U2 gate.
<i>U1Gate</i>	Represents an parameterized single qubit gate, parameterized in the same way as IBM's U1 gate.
<i>SingleQutritGate</i>	This gate represents an arbitrary parameterized single-qutrit gate.
<i>CSUMGate</i>	Represents the constant two-qutrit gate CSUM
<i>CPIGate</i>	Represents the constant two-qutrit gate CPI.
<i>CPIPhaseGate</i>	Represents the constant two-qutrit gate CPI with phase differences.
<i>CNOTGate</i>	Represents the constant two-qubit gate CNOT.
<i>CZGate</i>	Represents the constant two-qubit gate Controlled-Z.
<i>ISwapGate</i>	Represents the constant two-qubit gate ISwap.
<i>XXGate</i>	Represents the constant two-qubit gate XX( $\pi/2$ ).
<i>NonadjacentCNOTGate</i>	Represents the two-qubit gate CNOT, but between two qubits that are not necessarily next to each other.
<i>UGate</i>	Represents an arbitrary constant gate, defined by the unitary passed to the initializer.

continues on next page

Table 44 – continued from previous page

<i>UpgradedConstantGate</i>	Represents a constant gate, based on the Gate passed to its initializer, but upgraded to act on qudits of a larger size.
<i>CUGate</i>	Represents an arbitrary controlled gate, defined by the unitary passed to the initializer.
<i>CNOTRootGate</i>	Represents the sqrt(CNOT) gate. Two sqrt(CNOT) gates in a row will form a CNOT gate.
<i>KroneckerGate</i>	Represents the Kronecker product of a list of gates. This is equivalent to performing those gate in parallel in a quantum circuit.
<i>ProductGate</i>	Represents a matrix product of Gates. This is equivalent to performing those gates sequentially in a quantum circuit.
<i>Project</i>	The project class wraps most of the functionality of Qsearch as intended to help manage working with Qsearch.

## 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**(self, \*names)

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

**\_\_getitem\_\_**(self, name)

**\_\_delitem\_\_**(self, name)

**\_\_getattr\_\_**(self, name)

**\_\_setattr\_\_**(self, name, value)

Implement setattr(self, name, value).

**\_\_contains\_\_**(self, name)

**manually\_entered**(self, \*names, location='dict', operator='all')

**empty\_copy**(self)

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

**copy**(self)

Create a full copy of an Options object.

**\_\_copy\_\_**(self)

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

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

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

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

**\_update\_dict**(*self*, *otherdict*)

**set\_defaults**(*self*, *\*\*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**(*self*, *\*\*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**(*self*, *\*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**(*self*, *\*names*)

Removes the defaults for the specified names.

**remove\_smart\_defaults**(*self*, *\*names*)

Removes the smart\_defaults for the specified names.

**generate\_cache**(*self*)

Caches valuesa for all functions in smart\_defaults.

**save**(*self*, *filepath=None*)

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

**load**(*self*, *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\_\_**(*self*)

**\_\_setstate\_\_**(*self*, *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**(*self*, *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 upper 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**(self, 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**(self, 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  $d \times d$ , where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**mat\_jac**(self, 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 *i*th 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**(self, 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\_\_(self, other)**  
Return self==value.

**\_\_hash\_\_(self)**  
Return hash(self).

**copy(self)**

**\_parts(self)**

**\_\_copy\_\_(self)**

**\_\_deepcopy\_\_(self, memo)**

**\_\_repr\_\_(self)**  
Return repr(self).

**validate\_structure(self)**

**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(self, 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(self, 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\_\_(self)**

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(self, 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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**mat\_jac(self, 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 *i*th 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(self, 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`, `XZZZGate`, `CNOTGate`, and `NonadjacentCNOTGate`.

```
__repr__(self)
    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**(*self*, *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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** `np.ndarray`

**mat\_jac**(*self*, *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 *i*th 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**(*self*, *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`, `XZZZGate`, `CNOTGate`, and `NonadjacentCNOTGate`.

**\_\_repr\_\_(self)**

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(self, 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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** `np.ndarray`

**mat\_jac(self, 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 *i*th 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**(*self*, *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\_\_**(*self*)

Return `repr(self)`.

**class** qsearch.**SXGate**

Bases: `Gate`

Represents a  $\sqrt{X}$  rotation on one qubit, which is equivalent to `XGate()` with a parameter 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**(*self*, *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**(*self*, *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, ZXZXZGate, CNOTGate, and NonadjacentCNOTGate.

**\_\_repr\_\_**(*self*)

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**(*self*, *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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**mat\_jac**(*self*, *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 *i*th 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**(*self*, *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, ZXZXZGate, CNOTGate, and NonadjacentCNOTGate.

```
__repr__(self)
```

Return repr(self).

```
class qsearch.ZXZXZGate
```

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(self, 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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

```
mat_jac(self, 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 *i*th 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(self, 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`, `XZZZGate`, `CNOTGate`, and `NonadjacentCNOTGate`.

**\_\_repr\_\_(self)**

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(self, 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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** `np.ndarray`

**mat\_jac(self, 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 *i*th 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**(*self*, *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\_\_**(*self*, *other*)

Return self==value.

**\_\_repr\_\_**(*self*)

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**(*self*, *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  $d \times d$  where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**mat\_jac**(*self*, *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 *i*th 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**(*self*, *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\_\_**(*self*, *other*)

Return `self==value`.

**\_\_repr\_\_**(*self*)

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**(*self*, *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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**mat\_jac**(*self*, *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 `i`th 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**(*self*, *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\_\_**(*self*, *other*)

Return `self==value`.

**\_\_repr\_\_**(*self*)

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**(*self*, *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**(*self*, *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 *i*th 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**(*self*, *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\_\_**(*self*)

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**(*self*, *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**(*self*, *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\_\_**(*self*)

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**(*self*, *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**(*self*, *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\_\_(self)**

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(self, 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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**assemble(self, 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__(self)
    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__(self, other)
    Return self==value.
```

**matrix**(self, 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  $d \times d$ , where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**assemble**(self, 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__(self)
    Return repr(self).
```

**class** qsearch.CZGateBases: [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\_\_**(*self*, *other*)

Return `self==value`.

**matrix**(*self*, *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**(*self*, *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\_\_**(*self*)

Return `repr(self)`.

**class** qsearch.ISwapGateBases: [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\_\_(self, other)**

Return self==value.

**matrix(self, 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  $d \times d$ , where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**assemble(self, 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"`, `gate_name`, (`*gate_parameters`), (`*gate_indices`))

Where `gate_name` corresponds to a gate that an Assembler will recognize, `gate_parameters` corresponds to the parameters for the specified gate (usually but not always calculated from v), and `gate_indices` 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, ZXZXZGate, CNOTGate, and NonadjacentCNOTGate.

**\_\_repr\_\_(self)**

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\_\_(self, other)**

Return self==value.

**matrix**(*self*, *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**(*self*, *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\_\_**(*self*)

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**(*self*, *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**(*self*, *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`, `XZZZGate`, `CNOTGate`, and `NonadjacentCNOTGate`.

**\_\_repr\_\_**(*self*)

Return `repr(self)`.

**validate\_structure**(*self*)

**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 `dtype="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**(*self*, *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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**assemble**(*self*, *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\_\_**(*self*)

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**(*self*, *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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** `np.ndarray`

**assemble**(*self*, *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__(self)
```

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(self, 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(self, 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\_\_(self)**

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(self, 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  $d \times d$  where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**assemble(self, 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\_\_(self)**

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**(*self*, *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  $d \times d$ , where *d* is the intended qudit size (*d* is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**mat\_jac**(*self*, *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 *i*th 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**(*self*, *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**(*self*, *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(self)`

`__deepcopy__(self, memo)`

`__repr__(self)`

Return repr(self).

`validate_structure(self)`

**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**(self, 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  $d \times d$ , where d is the intended qudit size (d is 2 for qubits, 3 for qutrits, etc.)

**Return type** np.ndarray

**mat\_jac**(self, 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**(self, 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**(*self*, \**gates*)

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

**Parameters** *gates* – A list of Gates to be appended.

**inserting**(*self*, \**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 beginning of the ProductGate.

**\_\_deepcopy\_\_**(*self*, *memo*)

**\_\_repr\_\_**(*self*)

Return repr(*self*).

**validate\_structure**(*self*)

**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.

**\_save**(*self*)

**\_checkpoint\_path**(*self*, *name*)

**add\_compilation**(*self*, *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\_\_**(*self*, *keyword*, *value*)

**configure\_compiler\_override**(*self*, *keyword*, *value*)

An unsafe method that allows the user to set global Project Options even if there is existing work.

**\_\_getitem\_\_**(*self*, *keyword*)

**\_\_delitem\_\_**(*self*, *keyword*)

**configure**(*self*, *\*\*dictionary*)

Adds multiple options to the global Project Options at once.

**reset**(*self*, *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**(*self*, *name*)

Removes a compilation from a Project.

**Parameters** *name* – The name of the compilation to remove

**clear**(*self*, *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\_\_**(*self*)

**\_\_exit\_\_**(*self*, *exc\_typ*, *exc\_val*, *exc\_tb*)

**set\_defaults**(*self*, *defaults=standard\_defaults*)

Updates the Project Options with the standard defaults from defaults.py, or a provided dictionary.

**set\_smart\_defaults**(*self*, *smart\_defaults=standard\_smart\_defaults*)

Updates the Project Options with the standard smart\_defaults from defaults.py, or a provided dictionary

**run**(*self*)

Runs all of the compilations in the Project.

**post\_process**(*self*, *postprocessor*, *name=None*, *options=None*, *\*\*extraargs*)

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**(*self*)

Returns a True if all compilations in the Project have finished and False otherwise.

**finish**(*self*)

Called when done running compilations in order to end MPI tasks.

**status**(*self*, *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

**property compilations**(*self*)

The list of names corresponding to compilations on this Project.

**\_compilation\_status**(*self*, *name*)

**\_overall\_status**(*self*)

**get\_result**(*self*, *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**(*self*, *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**(*self*, *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**(*self*, *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**(*self*, *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



## 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.



## **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.





## **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!



## 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](#), 13
- [qsearch.compiler](#), 15
- [qsearch.defaults](#), 16
- [qsearch.evaluation](#), 18
- [qsearch.gates](#), 19
- [qsearch.gatesets](#), 46
- [qsearch.heuristics](#), 56
- [qsearch.integrations](#), 56
- [qsearch.leap\\_compiler](#), 57
- [qsearch.logging](#), 59
- [qsearch.multistart\\_solvers](#), 59
- [qsearch.objectives](#), 60
- [qsearch.options](#), 61
- [qsearch.parallelizers](#), 63
- [qsearch.persistent\\_aposmm](#), 65
- [qsearch.post\\_processing](#), 68
- [qsearch.project](#), 70
- [qsearch.solvers](#), 73
- [qsearch.unitaries](#), 74
- [qsearch.utils](#), 78



## Symbols

- `__contains__()` (*qsearch.Options* method), 82
- `__contains__()` (*qsearch.options.Options* method), 62
- `__copy__()` (*qsearch.Gate* method), 85
- `__copy__()` (*qsearch.Options* method), 82
- `__copy__()` (*qsearch.gates.Gate* method), 22
- `__copy__()` (*qsearch.options.Options* method), 62
- `__deepcopy__()` (*qsearch.Gate* method), 85
- `__deepcopy__()` (*qsearch.KroneckerGate* method), 108
- `__deepcopy__()` (*qsearch.ProductGate* method), 109
- `__deepcopy__()` (*qsearch.gates.Gate* method), 22
- `__deepcopy__()` (*qsearch.gates.KroneckerGate* method), 44
- `__deepcopy__()` (*qsearch.gates.ProductGate* method), 46
- `__delitem__()` (*qsearch.Options* method), 82
- `__delitem__()` (*qsearch.Project* method), 109
- `__delitem__()` (*qsearch.options.Options* method), 62
- `__delitem__()` (*qsearch.project.Project* method), 71
- `__enter__()` (*qsearch.Project* method), 110
- `__enter__()` (*qsearch.project.Project* method), 71
- `__eq__()` (*qsearch.CNOTGate* method), 99
- `__eq__()` (*qsearch.CZGate* method), 100
- `__eq__()` (*qsearch.Gate* method), 85
- `__eq__()` (*qsearch.ISwapGate* method), 101
- `__eq__()` (*qsearch.U1Gate* method), 95
- `__eq__()` (*qsearch.U2Gate* method), 94
- `__eq__()` (*qsearch.U3Gate* method), 93
- `__eq__()` (*qsearch.XXGate* method), 101
- `__eq__()` (*qsearch.gates.CNOTGate* method), 36
- `__eq__()` (*qsearch.gates.CZGate* method), 37
- `__eq__()` (*qsearch.gates.Gate* method), 22
- `__eq__()` (*qsearch.gates.ISwapGate* method), 37
- `__eq__()` (*qsearch.gates.U1Gate* method), 32
- `__eq__()` (*qsearch.gates.U2Gate* method), 31
- `__eq__()` (*qsearch.gates.U3Gate* method), 30
- `__eq__()` (*qsearch.gates.XXGate* method), 38
- `__eq__()` (*qsearch.gatesets.Gateset* method), 48
- `__eq__()` (*qsearch.solvers.Solver* method), 73
- `__exit__()` (*qsearch.Project* method), 110
- `__exit__()` (*qsearch.project.Project* method), 71
- `__getattr__()` (*qsearch.Options* method), 82
- `__getattr__()` (*qsearch.options.Options* method), 62
- `__getitem__()` (*qsearch.Options* method), 82
- `__getitem__()` (*qsearch.Project* method), 109
- `__getitem__()` (*qsearch.options.Options* method), 62
- `__getitem__()` (*qsearch.project.Project* method), 71
- `__getstate__()` (*qsearch.Options* method), 83
- `__getstate__()` (*qsearch.options.Options* method), 63
- `__hash__()` (*qsearch.Gate* method), 85
- `__hash__()` (*qsearch.gates.Gate* method), 22
- `__repr__()` (*qsearch.CNOTGate* method), 99
- `__repr__()` (*qsearch.CNOTRootGate* method), 106
- `__repr__()` (*qsearch.CPIGate* method), 98
- `__repr__()` (*qsearch.CPIPhaseGate* method), 99
- `__repr__()` (*qsearch.CSUMGate* method), 97
- `__repr__()` (*qsearch.CUGate* method), 106
- `__repr__()` (*qsearch.CZGate* method), 100
- `__repr__()` (*qsearch.Gate* method), 85
- `__repr__()` (*qsearch.ISwapGate* method), 101
- `__repr__()` (*qsearch.IdentityGate* method), 86
- `__repr__()` (*qsearch.KroneckerGate* method), 108
- `__repr__()` (*qsearch.NonadjacentCNOTGate* method), 103
- `__repr__()` (*qsearch.ProductGate* method), 109
- `__repr__()` (*qsearch.SXGate* method), 90
- `__repr__()` (*qsearch.SingleQutritGate* method), 96
- `__repr__()` (*qsearch.U1Gate* method), 95
- `__repr__()` (*qsearch.U2Gate* method), 94
- `__repr__()` (*qsearch.U3Gate* method), 93
- `__repr__()` (*qsearch.UGate* method), 104
- `__repr__()` (*qsearch.UpgradedConstantGate* method), 105
- `__repr__()` (*qsearch.XGate* method), 87
- `__repr__()` (*qsearch.XXGate* method), 102
- `__repr__()` (*qsearch.XZXZGate* method), 92
- `__repr__()` (*qsearch.YGate* method), 88
- `__repr__()` (*qsearch.ZGate* method), 89
- `__repr__()` (*qsearch.ZZXZGate* method), 91
- `__repr__()` (*qsearch.gates.CNOTGate* method), 36
- `__repr__()` (*qsearch.gates.CNOTRootGate* method), 43
- `__repr__()` (*qsearch.gates.CPIGate* method), 35
- `__repr__()` (*qsearch.gates.CPIPhaseGate* method), 35
- `__repr__()` (*qsearch.gates.CSUMGate* method), 34

\_\_repr\_\_() (*qsearch.gates.CUGate* method), 42  
 \_\_repr\_\_() (*qsearch.gates.CZGate* method), 37  
 \_\_repr\_\_() (*qsearch.gates.Gate* method), 22  
 \_\_repr\_\_() (*qsearch.gates.ISwapGate* method), 38  
 \_\_repr\_\_() (*qsearch.gates.IdentityGate* method), 23  
 \_\_repr\_\_() (*qsearch.gates.KroneckerGate* method), 44  
 \_\_repr\_\_() (*qsearch.gates.NonadjacentCNOTGate* method), 40  
 \_\_repr\_\_() (*qsearch.gates.ProductGate* method), 46  
 \_\_repr\_\_() (*qsearch.gates.SXGate* method), 27  
 \_\_repr\_\_() (*qsearch.gates.SingleQutritGate* method), 33  
 \_\_repr\_\_() (*qsearch.gates.U1Gate* method), 32  
 \_\_repr\_\_() (*qsearch.gates.U2Gate* method), 31  
 \_\_repr\_\_() (*qsearch.gates.U3Gate* method), 30  
 \_\_repr\_\_() (*qsearch.gates.UGate* method), 41  
 \_\_repr\_\_() (*qsearch.gates.UpgradedConstantGate* method), 42  
 \_\_repr\_\_() (*qsearch.gates.XGate* method), 24  
 \_\_repr\_\_() (*qsearch.gates.XXGate* method), 39  
 \_\_repr\_\_() (*qsearch.gates.XZXZGate* method), 29  
 \_\_repr\_\_() (*qsearch.gates.YGate* method), 25  
 \_\_repr\_\_() (*qsearch.gates.ZGate* method), 26  
 \_\_repr\_\_() (*qsearch.gates.ZZXZGate* method), 28  
 \_\_setattr\_\_() (*qsearch.Options* method), 82  
 \_\_setattr\_\_() (*qsearch.options.Options* method), 62  
 \_\_setitem\_\_() (*qsearch.Project* method), 109  
 \_\_setitem\_\_() (*qsearch.project.Project* method), 71  
 \_\_setstate\_\_() (*qsearch.Options* method), 83  
 \_\_setstate\_\_() (*qsearch.options.Options* method), 63  
 \_checkpoint\_path() (*qsearch.Project* method), 109  
 \_checkpoint\_path() (*qsearch.project.Project* method), 70  
 \_cnot (*qsearch.CNOTGate* attribute), 99  
 \_cnot (*qsearch.gates.CNOTGate* attribute), 36  
 \_cnr (*qsearch.CNOTRootGate* attribute), 106  
 \_cnr (*qsearch.gates.CNOTRootGate* attribute), 43  
 \_compilation\_status() (*qsearch.Project* method), 110  
 \_compilation\_status() (*qsearch.project.Project* method), 72  
 \_cpi (*qsearch.CPIGate* attribute), 97  
 \_cpi (*qsearch.gates.CPIGate* attribute), 34  
 \_csum (*qsearch.CSUMGate* attribute), 96  
 \_csum (*qsearch.gates.CSUMGate* attribute), 33  
 \_gate (*qsearch.CZGate* attribute), 100  
 \_gate (*qsearch.ISwapGate* attribute), 101  
 \_gate (*qsearch.XXGate* attribute), 101  
 \_gate (*qsearch.gates.CZGate* attribute), 37  
 \_gate (*qsearch.gates.ISwapGate* attribute), 37  
 \_gate (*qsearch.gates.XXGate* attribute), 38  
 \_options\_actual\_parameters (in module *qsearch.options*), 62  
 \_overall\_status() (*qsearch.Project* method), 110

\_overall\_status() (*qsearch.project.Project* method), 72  
 \_parts() (*qsearch.Gate* method), 85  
 \_parts() (*qsearch.KroneckerGate* method), 107  
 \_parts() (*qsearch.gates.Gate* method), 22  
 \_parts() (*qsearch.gates.KroneckerGate* method), 44  
 \_save() (*qsearch.Project* method), 109  
 \_save() (*qsearch.project.Project* method), 70  
 \_update\_dict() (*qsearch.Options* method), 83  
 \_update\_dict() (*qsearch.options.Options* method), 62

## A

add\_compilation() (*qsearch.Project* method), 109  
 add\_compilation() (*qsearch.project.Project* method), 70  
 aposmm() (in module *qsearch.persistent\_aposmm*), 65  
 appending() (*qsearch.gates.KroneckerGate* method), 44  
 appending() (*qsearch.gates.ProductGate* method), 45  
 appending() (*qsearch.KroneckerGate* method), 107  
 appending() (*qsearch.ProductGate* method), 109  
 arbitrary\_cnot (in module *qsearch.unitaries*), 75  
 arbitrary\_cnot() (in module *qsearch.unitaries*), 78  
 assemble() (*qsearch.assemblers.Assembler* method), 9  
 assemble() (*qsearch.assemblers.DictionaryAssembler* method), 9  
 assemble() (*qsearch.CNOTGate* method), 99  
 assemble() (*qsearch.CNOTRootGate* method), 106  
 assemble() (*qsearch.CPIGate* method), 97  
 assemble() (*qsearch.CPIPhaseGate* method), 98  
 assemble() (*qsearch.CSUMGate* method), 97  
 assemble() (*qsearch.CUGate* method), 105  
 assemble() (*qsearch.CZGate* method), 100  
 assemble() (*qsearch.Gate* method), 84  
 assemble() (*qsearch.gates.CNOTGate* method), 36  
 assemble() (*qsearch.gates.CNOTRootGate* method), 43  
 assemble() (*qsearch.gates.CPIGate* method), 34  
 assemble() (*qsearch.gates.CPIPhaseGate* method), 35  
 assemble() (*qsearch.gates.CSUMGate* method), 33  
 assemble() (*qsearch.gates.CUGate* method), 42  
 assemble() (*qsearch.gates.CZGate* method), 37  
 assemble() (*qsearch.gates.Gate* method), 21  
 assemble() (*qsearch.gates.IdentityGate* method), 22  
 assemble() (*qsearch.gates.ISwapGate* method), 38  
 assemble() (*qsearch.gates.KroneckerGate* method), 44  
 assemble() (*qsearch.gates.NonadjacentCNOTGate* method), 39  
 assemble() (*qsearch.gates.ProductGate* method), 45  
 assemble() (*qsearch.gates.SingleQutritGate* method), 33  
 assemble() (*qsearch.gates.SXGate* method), 26  
 assemble() (*qsearch.gates.U1Gate* method), 32  
 assemble() (*qsearch.gates.U2Gate* method), 30  
 assemble() (*qsearch.gates.U3Gate* method), 29  
 assemble() (*qsearch.gates.UGate* method), 40



- `assemble()` (*qsearch.gates.UpgradeConstantGate method*), 41  
`assemble()` (*qsearch.gates.XGate method*), 23  
`assemble()` (*qsearch.gates.XXGate method*), 39  
`assemble()` (*qsearch.gates.XZXZGate method*), 28  
`assemble()` (*qsearch.gates.YGate method*), 24  
`assemble()` (*qsearch.gates.ZGate method*), 25  
`assemble()` (*qsearch.gates.ZZXZGate method*), 27  
`assemble()` (*qsearch.IdentityGate method*), 85  
`assemble()` (*qsearch.ISwapGate method*), 101  
`assemble()` (*qsearch.KroneckerGate method*), 107  
`assemble()` (*qsearch.NonadjacentCNOTGate method*), 103  
`assemble()` (*qsearch.ProductGate method*), 108  
`assemble()` (*qsearch.Project method*), 111  
`assemble()` (*qsearch.project.Project method*), 72  
`assemble()` (*qsearch.SingleQutritGate method*), 96  
`assemble()` (*qsearch.SXGate method*), 89  
`assemble()` (*qsearch.U1Gate method*), 95  
`assemble()` (*qsearch.U2Gate method*), 94  
`assemble()` (*qsearch.U3Gate method*), 92  
`assemble()` (*qsearch.UGate method*), 103  
`assemble()` (*qsearch.UpgradeConstantGate method*), 104  
`assemble()` (*qsearch.XGate method*), 86  
`assemble()` (*qsearch.XXGate method*), 102  
`assemble()` (*qsearch.XZXZGate method*), 91  
`assemble()` (*qsearch.YGate method*), 87  
`assemble()` (*qsearch.ZGate method*), 88  
`assemble()` (*qsearch.ZZXZGate method*), 90  
`Assembler` (class in *qsearch.assemblers*), 9  
`ASSEMBLER_IBMOPENQASM` (in module *qsearch.assemblers*), 8, 9  
`ASSEMBLER_OPENQASM` (in module *qsearch.assemblers*), 8, 9  
`ASSEMBLER_QISKIT` (in module *qsearch.assemblers*), 8, 9  
`ASSEMBLER_QUTRIT` (in module *qsearch.assemblers*), 8, 9  
`assemblydict_ibmopenqasm` (in module *qsearch.assemblers*), 9  
`assemblydict_openqasm` (in module *qsearch.assemblers*), 9  
`assemblydict_qiskit` (in module *qsearch.assemblers*), 9  
`assemblydict_qutrit` (in module *qsearch.assemblers*), 9  
`astar()` (in module *qsearch.heuristics*), 56
- ## B
- `Backend` (class in *qsearch.backends*), 10  
`BackwardsCompatibleObjective` (class in *qsearch.objectives*), 61
- `BasicSingleQubitReduction_PostProcessor` (class in *qsearch.post\_processing*), 69  
`BasicSingleQubitReduction_PostProcessor` (in module *qsearch.post\_processing*), 68  
`BFGS_Jac_Solver` (class in *qsearch.solvers*), 74  
`branching_factor()` (*qsearch.gatesets.Gateset method*), 48  
`branching_factor()` (*qsearch.gatesets.QubitCNOTLinear method*), 50  
`branching_factor()` (*qsearch.gatesets.QubitCZLinear method*), 51  
`branching_factor()` (*qsearch.gatesets.QubitISwapLinear method*), 52  
`branching_factor()` (*qsearch.gatesets.QubitXXLinear method*), 53
- ## C
- `Checkpoint` (class in *qsearch.checkpoints*), 11  
`ChildCheckpoint` (class in *qsearch.checkpoints*), 12  
`ChildCheckpoint` (in module *qsearch.checkpoints*), 11  
`clear()` (*qsearch.Project method*), 110  
`clear()` (*qsearch.project.Project method*), 71  
`CMA_Solver` (class in *qsearch.solvers*), 73  
`cnot` (in module *qsearch.unitaries*), 74, 77  
`CNOTGate` (class in *qsearch*), 99  
`CNOTGate` (class in *qsearch.gates*), 35  
`CNOTRootGate` (class in *qsearch*), 106  
`CNOTRootGate` (class in *qsearch.gates*), 42  
`COBYLA_Solver` (class in *qsearch.solvers*), 73  
`compilations` (*qsearch.Project property*), 110  
`compilations` (*qsearch.project.Project property*), 72  
`compile()` (*qsearch.Compiler method*), 83  
`compile()` (*qsearch.compiler.Compiler method*), 15  
`compile()` (*qsearch.compiler.SearchCompiler method*), 16  
`compile()` (*qsearch.leap\_compiler.LeapCompiler method*), 58  
`compile()` (*qsearch.leap\_compiler.SubCompiler method*), 58  
`compile()` (*qsearch.post\_processing.LEAPReoptimizing\_PostProcessor method*), 69  
`compile()` (*qsearch.SearchCompiler method*), 84  
`Compiler` (class in *qsearch*), 83  
`Compiler` (class in *qsearch.compiler*), 15  
`COMPLETE` (*qsearch.project.Project\_Status attribute*), 70  
`complete()` (*qsearch.Project method*), 110  
`complete()` (*qsearch.project.Project method*), 71  
`configure()` (*qsearch.Project method*), 109  
`configure()` (*qsearch.project.Project method*), 71  
`configure_compiler_override()` (*qsearch.Project method*), 109  
`configure_compiler_override()` (*qsearch.project.Project method*), 71

`constraint_distsq()` (in module *qsearch.evaluation*), 19  
`constraint_distsq_jac()` (in module *qsearch.evaluation*), 19  
`convert()` (*qsearch.integrations.QiskitGateConverter* method), 57  
`convert_cx()` (*qsearch.integrations.QiskitGateConverter* method), 57  
`convert_rx()` (*qsearch.integrations.QiskitGateConverter* method), 57  
`convert_ry()` (*qsearch.integrations.QiskitGateConverter* method), 57  
`convert_rz()` (*qsearch.integrations.QiskitGateConverter* method), 57  
`convert_u2()` (*qsearch.integrations.QiskitGateConverter* method), 57  
`convert_u3()` (*qsearch.integrations.QiskitGateConverter* method), 57  
`copy()` (*qsearch.Gate* method), 85  
`copy()` (*qsearch.gates.Gate* method), 22  
`copy()` (*qsearch.Options* method), 82  
`copy()` (*qsearch.options.Options* method), 62  
`cost_combo_linear()` (in module *qsearch.evaluation*), 19  
`cost_combo_linear_jac()` (in module *qsearch.evaluation*), 19  
`cost_linear()` (in module *qsearch.evaluation*), 19  
`cost_linear_jac()` (in module *qsearch.evaluation*), 19  
`CPIGate` (class in *qsearch*), 97  
`CPIGate` (class in *qsearch.gates*), 34  
`CPIPhaseGate` (class in *qsearch*), 98  
`CPIPhaseGate` (class in *qsearch.gates*), 35  
`CSUMGate` (class in *qsearch*), 96  
`CSUMGate` (class in *qsearch.gates*), 33  
`CUGate` (class in *qsearch*), 105  
`CUGate` (class in *qsearch.gates*), 42  
`cut_end()` (in module *qsearch.leap\_compiler*), 58  
`CZGate` (class in *qsearch*), 99  
`CZGate` (class in *qsearch.gates*), 36  
`default_eval_func()` (in module *qsearch.defaults*), 17  
`default_heuristic()` (in module *qsearch.defaults*), 17  
`default_logger()` (in module *qsearch.defaults*), 17  
`default_num_tasks()` (in module *qsearch.parallelizers*), 64  
`default_objective()` (in module *qsearch.defaults*), 17  
`default_solver()` (in module *qsearch.solvers*), 73  
`DefaultQubit` (in module *qsearch.gatesets*), 46, 55  
`DefaultQutrit` (in module *qsearch.gatesets*), 46, 55  
`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*), 59  
`distance_metric` (*qsearch.solvers.LeastSquares\_Jac\_Solver* property), 74  
`distance_metric` (*qsearch.solvers.Solver* property), 73  
`distance_with_initial_state()` (in module *qsearch.comparison*), 15  
`distance_with_initial_state_jac()` (in module *qsearch.comparison*), 15  
`DIY_Solver` (class in *qsearch.solvers*), 74  
`dijkstra()` (in module *qsearch.heuristics*), 56  
`done()` (*qsearch.parallelizers.MPIParallelizer* method), 65  
`done()` (*qsearch.parallelizers.MultiprocessingParallelizer* method), 64  
`done()` (*qsearch.parallelizers.Parallelizer* method), 64  
`done()` (*qsearch.parallelizers.ProcessPoolParallelizer* method), 65  
`downgrade_qudits_residuals()` (in module *qsearch.utils*), 80  
`downgrade_qudits_residuals_jac()` (in module *qsearch.utils*), 80

## D

`decide_where_to_start_localopt()` (in module *qsearch.persistent\_aposmm*), 67  
`Default` (in module *qsearch.gatesets*), 46, 55  
`default_checkpoint()` (in module *qsearch.defaults*), 17  
`default_compiler()` (in module *qsearch.defaults*), 17  
`default_error_func()` (in module *qsearch.defaults*), 17  
`default_error_jac()` (in module *qsearch.defaults*), 17  
`default_error_residuals()` (in module *qsearch.defaults*), 17  
`default_error_residuals_jac()` (in module *qsearch.defaults*), 17

## E

`empty_copy()` (*qsearch.Options* method), 82  
`empty_copy()` (*qsearch.options.Options* method), 62  
`endian_reverse` (in module *qsearch.utils*), 78  
`endian_reverse()` (in module *qsearch.utils*), 80  
`error_distsq()` (in module *qsearch.evaluation*), 18  
`error_distsq_jac()` (in module *qsearch.evaluation*), 19  
`error_stateprep_distsq()` (in module *qsearch.evaluation*), 19  
`error_stateprep_distsq_jac()` (in module *qsearch.evaluation*), 19  
`eval_func_from_residuals()` (in module *qsearch.comparison*), 15

`evaluate_step()` (in module *qsearch.parallelizers*), 64

## F

`FileCheckpoint` (class in *qsearch.checkpoints*), 12

`FileCheckpoint` (in module *qsearch.checkpoints*), 11

`fill_row()` (in module *qsearch.gatesets*), 55

`filtered()` (*qsearch.Options* method), 82

`filtered()` (*qsearch.options.Options* method), 62

`find_last_3_cnots_linear()` (in module *qsearch.gatesets*), 55

`finish()` (*qsearch.Project* method), 110

`finish()` (*qsearch.project.Project* method), 71

`flatten_intermediate()` (in module *qsearch.assemblers*), 9

`fredkin` (in module *qsearch.unitaries*), 74, 77

`full_adder` (in module *qsearch.unitaries*), 74, 77

## G

`Gate` (class in *qsearch*), 84

`Gate` (class in *qsearch.gates*), 21

`Gateset` (class in *qsearch.gatesets*), 48

`gen_error_func()` (*qsearch.objectives.BackwardsCompatibleObjective* method), 61

`gen_error_func()` (*qsearch.objectives.MatrixDistanceObjective* method), 61

`gen_error_func()` (*qsearch.objectives.Objective* method), 60

`gen_error_func()` (*qsearch.objectives.StateprepObjective* method), 61

`gen_error_jac()` (*qsearch.objectives.BackwardsCompatibleObjective* method), 61

`gen_error_jac()` (*qsearch.objectives.MatrixDistanceObjective* method), 61

`gen_error_jac()` (*qsearch.objectives.Objective* method), 60

`gen_error_jac()` (*qsearch.objectives.StateprepObjective* method), 61

`gen_error_residuals()` (*qsearch.objectives.BackwardsCompatibleObjective* method), 61

`gen_error_residuals()` (*qsearch.objectives.MatrixDistanceObjective* method), 61

`gen_error_residuals()` (*qsearch.objectives.Objective* method), 60

`gen_error_residuals()` (*qsearch.objectives.StateprepObjective* method), 61

`gen_error_residuals_jac()` (*qsearch.objectives.BackwardsCompatibleObjective* method), 61

`gen_error_residuals_jac()` (*qsearch.objectives.MatrixDistanceObjective* method), 61

`gen_error_residuals_jac()` (*qsearch.objectives.Objective* method), 60

`gen_error_residuals_jac()` (*qsearch.objectives.StateprepObjective* method), 61

`gen_error_residuals_jac()` (*qsearch.objectives.BackwardsCompatibleObjective* method), 61

`gen_error_residuals_jac()` (*qsearch.objectives.MatrixDistanceObjective* method), 61

`gen_error_residuals_jac()` (*qsearch.objectives.Objective* method), 60

`gen_error_residuals_jac()` (*qsearch.objectives.StateprepObjective* method), 61

`gen_eval_func()` (*qsearch.objectives.Objective* method), 60

`general_swap` (in module *qsearch.unitaries*), 75

`general_swap()` (in module *qsearch.unitaries*), 77

`generate_cache()` (*qsearch.Options* method), 83

`generate_cache()` (*qsearch.options.Options* method), 63

`generate_HHL()` (in module *qsearch.advanced\_unitaries*), 7

`generate_miro()` (in module *qsearch.advanced\_unitaries*), 7

`generate_stateprep_target_matrix()` (in module *qsearch.utils*), 80

`get_options()` (*qsearch.Project* method), 111

`get_options()` (*qsearch.project.Project* method), 72

`get_result()` (*qsearch.Project* method), 110

`get_result()` (*qsearch.project.Project* method), 72

`get_reusable_executor` (in module *qsearch.parallelizers*), 64

`get_target()` (*qsearch.Project* method), 111

`get_target()` (*qsearch.project.Project* method), 72

`get_time()` (*qsearch.Project* method), 111

`get_time()` (*qsearch.project.Project* method), 72

`greedy()` (in module *qsearch.heuristics*), 56

`HHL` (in module *qsearch.advanced\_unitaries*), 7

`identity` (in module *qsearch.unitaries*), 75

`identity()` (in module *qsearch.defaults*), 17

`identity()` (in module *qsearch.unitaries*), 77

`IdentityGate` (class in *qsearch*), 85

`IdentityGate` (class in *qsearch.gates*), 22

`index_test()` (in module *qsearch.utils*), 80

`initial_layer()` (*qsearch.gatesets.Gateset* method), 48

`initial_layer()` (*qsearch.gatesets.QubitCNOTAdjacencyList* method), 54

`initial_layer()` (*qsearch.gatesets.QubitCNOTLinear* method), 50

`initial_layer()` (*qsearch.gatesets.QubitCNOTRing* method), 51

`initial_layer()` (*qsearch.gatesets.QubitCZLinear* method), 51

`initial_layer()` (*qsearch.gatesets.QubitISwapLinear* method), 52

`initial_layer()` (*qsearch.gatesets.QubitXXLinear* method), 53

`initial_layer()` (*qsearch.gatesets.QutritCNOTLinear method*), 55  
`initial_layer()` (*qsearch.gatesets.QutritCPIPhaseLinear method*), 54  
`initial_layer()` (*qsearch.gatesets.U3CNOTLinear method*), 49  
`initial_layer()` (*qsearch.gatesets.ZXZXZCNOTLinear method*), 49  
`initialize_APOSM()` (in module *qsearch.persistent\_aposmm*), 68  
`inserting()` (*qsearch.gates.ProductGate method*), 46  
`inserting()` (*qsearch.ProductGate method*), 109  
`ISwapGate` (class in *qsearch*), 100  
`ISwapGate` (class in *qsearch.gates*), 37

## K

`KroneckerGate` (class in *qsearch*), 106  
`KroneckerGate` (class in *qsearch.gates*), 43

## L

`LeapCompiler` (class in *qsearch.leap\_compiler*), 58  
`LEAPReoptimizing_PostProcessor` (class in *qsearch.post\_processing*), 69  
`LEAPReoptimizing_PostProcessor` (in module *qsearch.post\_processing*), 68  
`LeastSquares_Jac_Solver` (class in *qsearch.solvers*), 74  
`linear_topology()` (in module *qsearch.gatesets*), 55  
`load()` (*qsearch.Options method*), 83  
`load()` (*qsearch.options.Options method*), 63  
`Logger` (class in *qsearch.logging*), 59  
`logical_or` (in module *qsearch.unitaries*), 74, 77  
`logprint()` (*qsearch.logging.Logger method*), 59  
`LokyParallelizer` (class in *qsearch.parallelizers*), 64  
`LokyParallelizer` (in module *qsearch.parallelizers*), 63

## M

`make_required()` (*qsearch.Options method*), 83  
`make_required()` (*qsearch.options.Options method*), 62  
`manually_entered()` (*qsearch.Options method*), 82  
`manually_entered()` (*qsearch.options.Options method*), 62  
`map_steps()` (*qsearch.parallelizers.MPIParallelizer method*), 65  
`mat_jac()` (*qsearch.Gate method*), 84  
`mat_jac()` (*qsearch.gates.Gate method*), 21  
`mat_jac()` (*qsearch.gates.KroneckerGate method*), 44  
`mat_jac()` (*qsearch.gates.ProductGate method*), 45  
`mat_jac()` (*qsearch.gates.SingleQutritGate method*), 32  
`mat_jac()` (*qsearch.gates.U1Gate method*), 31  
`mat_jac()` (*qsearch.gates.U2Gate method*), 30  
`mat_jac()` (*qsearch.gates.U3Gate method*), 29  
`mat_jac()` (*qsearch.gates.XGate method*), 23  
`mat_jac()` (*qsearch.gates.XZXZGate method*), 28  
`mat_jac()` (*qsearch.gates.YGate method*), 24  
`mat_jac()` (*qsearch.gates.ZGate method*), 25  
`mat_jac()` (*qsearch.KroneckerGate method*), 107  
`mat_jac()` (*qsearch.ProductGate method*), 108  
`mat_jac()` (*qsearch.SingleQutritGate method*), 95  
`mat_jac()` (*qsearch.U1Gate method*), 94  
`mat_jac()` (*qsearch.U2Gate method*), 93  
`mat_jac()` (*qsearch.U3Gate method*), 92  
`mat_jac()` (*qsearch.XGate method*), 86  
`mat_jac()` (*qsearch.XZXZGate method*), 91  
`mat_jac()` (*qsearch.YGate method*), 87  
`mat_jac()` (*qsearch.ZGate method*), 88  
`mat_jac()` (*qsearch.ZXZXZGate method*), 90  
`matrix()` (*qsearch.CNOTGate method*), 99  
`matrix()` (*qsearch.CNOTRootGate method*), 106  
`matrix()` (*qsearch.CPIGate method*), 97  
`matrix()` (*qsearch.CPIPhaseGate method*), 98  
`matrix()` (*qsearch.CSUMGate method*), 96  
`matrix()` (*qsearch.CUGate method*), 105  
`matrix()` (*qsearch.CZGate method*), 100  
`matrix()` (*qsearch.Gate method*), 84  
`matrix()` (*qsearch.gates.CNOTGate method*), 36  
`matrix()` (*qsearch.gates.CNOTRootGate method*), 43  
`matrix()` (*qsearch.gates.CPIGate method*), 34  
`matrix()` (*qsearch.gates.CPIPhaseGate method*), 35  
`matrix()` (*qsearch.gates.CSUMGate method*), 33  
`matrix()` (*qsearch.gates.CUGate method*), 42  
`matrix()` (*qsearch.gates.CZGate method*), 37  
`matrix()` (*qsearch.gates.Gate method*), 21  
`matrix()` (*qsearch.gates.IdentityGate method*), 22  
`matrix()` (*qsearch.gates.ISwapGate method*), 37  
`matrix()` (*qsearch.gates.KroneckerGate method*), 43  
`matrix()` (*qsearch.gates.NonadjacentCNOTGate method*), 39  
`matrix()` (*qsearch.gates.ProductGate method*), 45  
`matrix()` (*qsearch.gates.SingleQutritGate method*), 32  
`matrix()` (*qsearch.gates.SXGate method*), 26  
`matrix()` (*qsearch.gates.U1Gate method*), 31  
`matrix()` (*qsearch.gates.U2Gate method*), 30  
`matrix()` (*qsearch.gates.U3Gate method*), 29  
`matrix()` (*qsearch.gates.UGate method*), 40  
`matrix()` (*qsearch.gates.UpgradedConstantGate method*), 41  
`matrix()` (*qsearch.gates.XGate method*), 23  
`matrix()` (*qsearch.gates.XXGate method*), 38  
`matrix()` (*qsearch.gates.XZXZGate method*), 28  
`matrix()` (*qsearch.gates.YGate method*), 24  
`matrix()` (*qsearch.gates.ZGate method*), 25  
`matrix()` (*qsearch.gates.ZXZXZGate method*), 27  
`matrix()` (*qsearch.IdentityGate method*), 85  
`matrix()` (*qsearch.ISwapGate method*), 101  
`matrix()` (*qsearch.KroneckerGate method*), 107



- `matrix()` (*qsearch.NonadjacentCNOTGate* method), 102
- `matrix()` (*qsearch.ProductGate* method), 108
- `matrix()` (*qsearch.SingleQutritGate* method), 95
- `matrix()` (*qsearch.SXGate* method), 89
- `matrix()` (*qsearch.U1Gate* method), 94
- `matrix()` (*qsearch.U2Gate* method), 93
- `matrix()` (*qsearch.U3Gate* method), 92
- `matrix()` (*qsearch.UGate* method), 103
- `matrix()` (*qsearch.UpgradeConstantGate* method), 104
- `matrix()` (*qsearch.XGate* method), 86
- `matrix()` (*qsearch.XXGate* method), 101
- `matrix()` (*qsearch.XZXZGate* method), 91
- `matrix()` (*qsearch.YGate* method), 87
- `matrix()` (*qsearch.ZGate* method), 88
- `matrix()` (*qsearch.ZZXZGate* method), 90
- `matrix_distance()` (in module *qsearch.comparison*), 14
- `matrix_distance_squared` (in module *qsearch.utils*), 78
- `matrix_distance_squared()` (in module *qsearch.comparison*), 14
- `matrix_distance_squared_jac` (in module *qsearch.utils*), 78
- `matrix_distance_squared_jac()` (in module *qsearch.comparison*), 14
- `matrix_kron()` (in module *qsearch.utils*), 79
- `matrix_product()` (in module *qsearch.utils*), 79
- `matrix_residuals` (in module *qsearch.utils*), 78
- `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*), 78
- `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*), 61
- `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*, 13
  - qsearch.compiler*, 15
  - qsearch.defaults*, 16
  - qsearch.evaluation*, 18
  - qsearch.gates*, 19
  - qsearch.gatesets*, 46
  - qsearch.heuristics*, 56
  - qsearch.integrations*, 56
  - qsearch.leap\_compiler*, 57
  - qsearch.logging*, 59
  - qsearch.multistart\_solvers*, 59
  - qsearch.objectives*, 60
  - qsearch.options*, 61
  - qsearch.parallelizers*, 63
  - qsearch.persistent\_aposmm*, 65
  - qsearch.post\_processing*, 68
  - qsearch.project*, 70
  - qsearch.solvers*, 73
  - qsearch.unitaries*, 74
  - qsearch.utils*, 78
- `MPI` (in module *qsearch.parallelizers*), 64
- `MPI` (in module *qsearch.project*), 70
- `MPI` (in module *qsearch.utils*), 79
- `mpi_do_work()` (in module *qsearch.utils*), 80
- `mpi_rank()` (in module *qsearch.utils*), 80
- `mpi_worker()` (in module *qsearch.utils*), 80
- `MPIParallelizer` (class in *qsearch.parallelizers*), 65
- `MPIParallelizer` (in module *qsearch.parallelizers*), 63
- `MultiprocessingParallelizer` (class in *qsearch.parallelizers*), 64
- `MultiprocessingParallelizer` (in module *qsearch.parallelizers*), 63
- `MultiStart_Solver` (class in *qsearch.multistart\_solvers*), 59
- N**
  - `NaiveMultiStart_Solver` (class in *qsearch.multistart\_solvers*), 60
  - `native_from_object` (in module *qsearch*), 84
  - `native_from_object` (in module *qsearch.gates*), 21
  - `NativeBackend` (class in *qsearch.backends*), 11
  - `NativeBackend` (in module *qsearch.backends*), 10
  - `nearest_unitary()` (in module *qsearch.utils*), 79
  - `NonadjacentCNOTGate` (class in *qsearch*), 102
  - `NonadjacentCNOTGate` (class in *qsearch.gates*), 39
  - `NOTBEGUN` (*qsearch.project.Project\_Status* attribute), 70
- O**
  - `Objective` (class in *qsearch.objectives*), 60
  - `op_norm()` (in module *qsearch.utils*), 79
  - `optimize_worker()` (in module *qsearch.multistart\_solvers*), 59
  - `Options` (class in *qsearch*), 82

Options (class in *qsearch.options*), 62

## P

Parallelizer (class in *qsearch.parallelizers*), 64

ParameterTuning\_PostProcessor (class in *qsearch.post\_processing*), 69

ParameterTuning\_PostProcessor (in module *qsearch.post\_processing*), 68

pauli\_x (in module *qsearch.unitaries*), 77

pauli\_y (in module *qsearch.unitaries*), 77

pauli\_z (in module *qsearch.unitaries*), 77

peres (in module *qsearch.unitaries*), 74, 77

post\_process() (*qsearch.Project* method), 110

post\_process() (*qsearch.project.Project* method), 71

post\_process\_circuit()  
(*qsearch.post\_processing.BasicSingleQubitReduction\_PostProcessor*  
method), 69

post\_process\_circuit()  
(*qsearch.post\_processing.LEAPReoptimizing\_PostProcessor*  
method), 69

post\_process\_circuit()  
(*qsearch.post\_processing.ParameterTuning\_PostProcessor*  
method), 69

post\_process\_circuit()  
(*qsearch.post\_processing.PostProcessor*  
method), 68

PostProcessor (class in *qsearch.post\_processing*), 68

prepare\_circuit() (*qsearch.backends.Backend*  
method), 10

prepare\_circuit() (*qsearch.backends.NativeBackend*  
method), 11

prepare\_circuit() (*qsearch.backends.PythonBackend*  
method), 11

prepare\_circuit() (*qsearch.backends.SmartDefaultBackend*  
method), 10

process\_initializer() (in module *qsearch.parallelizers*), 64

ProcessPoolParallelizer (class in *qsearch.parallelizers*), 65

ProcessPoolParallelizer (in module *qsearch.parallelizers*), 63

ProductGate (class in *qsearch*), 108

ProductGate (class in *qsearch.gates*), 44

PROGRESS (*qsearch.project.Project\_Status* attribute), 70

Project (class in *qsearch*), 109

Project (class in *qsearch.project*), 70

Project\_Status (class in *qsearch.project*), 70

PythonBackend (class in *qsearch.backends*), 10

PythonBackend (in module *qsearch.backends*), 10

## Q

q1\_unitary() (in module *qsearch.utils*), 80

qft (in module *qsearch.unitaries*), 75

qft() (in module *qsearch.unitaries*), 77

qiskit (in module *qsearch.integrations*), 57

qiskit\_to\_qsearch() (in module *qsearch.integrations*), 57

QiskitGateConverter (class in *qsearch.integrations*), 57

QiskitImportError, 57

qsearch  
module, 7

qsearch.advanced\_unitaries  
module, 7

qsearch.assemblers  
module, 8

qsearch.backends  
module, 10

qsearch.checkpoints  
module, 11

qsearch.comparison  
module, 13

qsearch.compiler  
module, 15

qsearch.defaults  
module, 16

qsearch.evaluation  
module, 18

qsearch.gates  
module, 19

qsearch.gatesets  
module, 46

qsearch.heuristics  
module, 56

qsearch.integrations  
module, 56

qsearch.leap\_compiler  
module, 57

qsearch.logging  
module, 59

qsearch.multistart\_solvers  
module, 59

qsearch.objectives  
module, 60

qsearch.options  
module, 61

qsearch.parallelizers  
module, 63

qsearch.persistent\_aposmm  
module, 65

qsearch.post\_processing  
module, 68

qsearch.project  
module, 70

qsearch.solvers  
module, 73

qsearch.unitaries  
module, 74

- qsearch.utils  
     module, 78  
 qt\_arb\_rot() (in module qsearch.utils), 80  
 QubitCNOTAdjacencyList (class in qsearch.gatesets), 53  
 QubitCNOTAdjacencyList (in module qsearch.gatesets), 46  
 QubitCNOTLinear (class in qsearch.gatesets), 49  
 QubitCNOTLinear (in module qsearch.gatesets), 46  
 QubitCNOTRing (class in qsearch.gatesets), 50  
 QubitCNOTRing (in module qsearch.gatesets), 46  
 QubitCZLinear (class in qsearch.gatesets), 51  
 QubitISwapLinear (class in qsearch.gatesets), 52  
 QubitXXLinear (class in qsearch.gatesets), 53  
 QutritCNOTLinear (class in qsearch.gatesets), 55  
 QutritCNOTLinear (in module qsearch.gatesets), 46  
 QutritCPIPhaseLinear (class in qsearch.gatesets), 54  
 QutritCPIPhaseLinear (in module qsearch.gatesets), 46
- ## R
- random\_near\_identity() (in module qsearch.utils), 80  
 re\_rot\_z() (in module qsearch.utils), 80  
 re\_rot\_z\_jac() (in module qsearch.utils), 80  
 recover() (qsearch.checkpoints.Checkpoint method), 11  
 recover() (qsearch.checkpoints.ChildCheckpoint method), 13  
 recover() (qsearch.checkpoints.FileCheckpoint method), 12  
 recover\_parent() (qsearch.checkpoints.ChildCheckpoint method), 13  
 remap (in module qsearch.utils), 78  
 remap() (in module qsearch.utils), 80  
 remove\_compilation() (qsearch.Project method), 110  
 remove\_compilation() (qsearch.project.Project method), 71  
 remove\_defaults() (qsearch.Options method), 83  
 remove\_defaults() (qsearch.options.Options method), 62  
 remove\_smart\_defaults() (qsearch.Options method), 83  
 remove\_smart\_defaults() (qsearch.options.Options method), 62  
 reset() (qsearch.Project method), 110  
 reset() (qsearch.project.Project method), 71  
 residuals\_blacklist() (in module qsearch.evaluation), 19  
 residuals\_blacklist\_jac() (in module qsearch.evaluation), 19  
 residuals\_difference() (in module qsearch.evaluation), 19  
 residuals\_difference\_jac() (in module qsearch.evaluation), 19  
 residuals\_product() (in module qsearch.evaluation), 19  
 residuals\_product\_jac() (in module qsearch.evaluation), 19  
 residuals\_slice() (in module qsearch.evaluation), 19  
 residuals\_slice\_jac() (in module qsearch.evaluation), 19  
 residuals\_with\_initial\_state() (in module qsearch.comparison), 15  
 residuals\_with\_initial\_state\_jac() (in module qsearch.comparison), 15  
 rot\_x (in module qsearch.unitaries), 75  
 rot\_x() (in module qsearch.unitaries), 77  
 rot\_x\_jac (in module qsearch.unitaries), 75  
 rot\_x\_jac() (in module qsearch.unitaries), 77  
 rot\_y (in module qsearch.unitaries), 75  
 rot\_y() (in module qsearch.unitaries), 77  
 rot\_y\_jac (in module qsearch.unitaries), 75  
 rot\_y\_jac() (in module qsearch.unitaries), 77  
 rot\_z (in module qsearch.unitaries), 75  
 rot\_z() (in module qsearch.unitaries), 77  
 rot\_z\_jac (in module qsearch.unitaries), 75  
 rot\_z\_jac() (in module qsearch.unitaries), 77  
 run() (qsearch.Project method), 110  
 run() (qsearch.project.Project method), 71  
 RUST\_ENABLED (in module qsearch.backends), 10
- ## S
- save() (qsearch.checkpoints.Checkpoint method), 11  
 save() (qsearch.checkpoints.ChildCheckpoint method), 13  
 save() (qsearch.checkpoints.FileCheckpoint method), 12  
 save() (qsearch.Options method), 83  
 save() (qsearch.options.Options method), 63  
 save\_parent() (qsearch.checkpoints.ChildCheckpoint method), 13  
 search\_layers() (qsearch.gatesets.Gateset method), 48  
 search\_layers() (qsearch.gatesets.QubitCNOTAdjacencyList method), 54  
 search\_layers() (qsearch.gatesets.QubitCNOTLinear method), 50  
 search\_layers() (qsearch.gatesets.QubitCNOTRing method), 51  
 search\_layers() (qsearch.gatesets.QubitCZLinear method), 51  
 search\_layers() (qsearch.gatesets.QubitISwapLinear method), 52  
 search\_layers() (qsearch.gatesets.QubitXXLinear method), 53  
 search\_layers() (qsearch.gatesets.QutritCNOTLinear method), 55  
 search\_layers() (qsearch.gatesets.QutritCPIPhaseLinear method), 54

`search_layers()` (*qsearch.gatesets.U3CNOTLinear* method), 49  
`search_layers()` (*qsearch.gatesets.ZXZXZCNOTLinear* method), 49  
`SearchCompiler` (class in *qsearch*), 83  
`SearchCompiler` (class in *qsearch.compiler*), 15  
`SequentialParallelizer` (class in *qsearch.parallelizers*), 65  
`SequentialParallelizer` (in module *qsearch.parallelizers*), 63  
`set_defaults()` (*qsearch.Options* method), 83  
`set_defaults()` (*qsearch.options.Options* method), 62  
`set_defaults()` (*qsearch.Project* method), 110  
`set_defaults()` (*qsearch.project.Project* method), 71  
`set_smart_defaults()` (*qsearch.Options* method), 83  
`set_smart_defaults()` (*qsearch.options.Options* method), 62  
`set_smart_defaults()` (*qsearch.Project* method), 110  
`set_smart_defaults()` (*qsearch.project.Project* method), 71  
`single_task()` (in module *qsearch.parallelizers*), 64  
`SingleQutritGate` (class in *qsearch*), 95  
`SingleQutritGate` (class in *qsearch.gates*), 32  
`SmartDefaultBackend` (class in *qsearch.backends*), 10  
`SmartDefaultBackend` (in module *qsearch.backends*), 10  
`solve_circuits_parallel()` (*qsearch.parallelizers.LokyParallelizer* method), 64  
`solve_circuits_parallel()` (*qsearch.parallelizers.MPIParallelizer* method), 65  
`solve_circuits_parallel()` (*qsearch.parallelizers.MultiprocessingParallelizersuccessors()* method), 64  
`solve_circuits_parallel()` (*qsearch.parallelizers.Parallelizer* method), 64  
`solve_circuits_parallel()` (*qsearch.parallelizers.ProcessPoolParallelizer* method), 65  
`solve_circuits_parallel()` (*qsearch.parallelizers.SequentialParallelizer* method), 65  
`solve_for_unitary()` (*qsearch.multistart\_solvers.MultiStart\_Solver* method), 60  
`solve_for_unitary()` (*qsearch.multistart\_solvers.NaiveMultiStart\_Solver* method), 60  
`solve_for_unitary()` (*qsearch.solvers.BFGS\_Jac\_Solver* method), 74  
`solve_for_unitary()` (*qsearch.solvers.CMA\_Solver* method), 73  
`solve_for_unitary()` (*qsearch.solvers.COBYLA\_Solver* method), 73  
`solve_for_unitary()` (*qsearch.solvers.DIY\_Solver* method), 74  
`solve_for_unitary()` (*qsearch.solvers.LeastSquares\_Jac\_Solver* method), 74  
`solve_for_unitary()` (*qsearch.solvers.Solver* method), 73  
`Solver` (class in *qsearch.solvers*), 73  
`sqrt_cnot` (in module *qsearch.unitaries*), 74, 77  
`sqrt_x` (in module *qsearch.unitaries*), 77  
`standard_defaults` (in module *qsearch*), 83  
`standard_defaults` (in module *qsearch.defaults*), 16, 17  
`standard_smart_defaults` (in module *qsearch*), 83  
`standard_smart_defaults` (in module *qsearch.defaults*), 16, 17  
`stateprep_defaults` (in module *qsearch.defaults*), 16, 17  
`stateprep_initial_state()` (in module *qsearch.defaults*), 17  
`stateprep_smart_defaults` (in module *qsearch.defaults*), 17  
`stateprep_target()` (in module *qsearch.defaults*), 17  
`StateprepObjective` (class in *qsearch.objectives*), 61  
`status()` (*qsearch.Project* method), 110  
`status()` (*qsearch.project.Project* method), 71  
`SubCompiler` (class in *qsearch.leap\_compiler*), 58  
`successors()` (*qsearch.gatesets.Gateset* method), 48  
`successors()` (*qsearch.gatesets.QubitCNOTLinear* method), 50  
`successors()` (*qsearch.gatesets.QubitCZLinear* method), 52  
`successors()` (*qsearch.gatesets.QubitISwapLinear* method), 52  
`successors()` (*qsearch.gatesets.QubitXXLinear* method), 53  
`swap` (in module *qsearch.unitaries*), 74, 77  
`SXGate` (class in *qsearch*), 89  
`SXGate` (class in *qsearch.gates*), 26  

## T

`toffoli` (in module *qsearch.unitaries*), 74, 77  

## U

`U1Gate` (class in *qsearch*), 94  
`U1Gate` (class in *qsearch.gates*), 31  
`U2Gate` (class in *qsearch*), 93  
`U2Gate` (class in *qsearch.gates*), 30  
`U3CNOTLinear` (class in *qsearch.gatesets*), 49  
`U3CNOTLinear` (in module *qsearch.gatesets*), 46  
`U3Gate` (class in *qsearch*), 92



U3Gate (class in *qsearch.gates*), 29  
 UGate (class in *qsearch*), 103  
 UGate (class in *qsearch.gates*), 40  
 update() (*qsearch.Options* method), 82  
 update() (*qsearch.options.Options* method), 62  
 update\_history\_dist() (in module *qsearch.persistent\_aposmm*), 67  
 updated() (*qsearch.Options* method), 82  
 updated() (*qsearch.options.Options* method), 62  
 upgrade\_qudits (in module *qsearch.utils*), 78  
 upgrade\_qudits() (in module *qsearch.utils*), 80  
 UpgradedConstantGate (class in *qsearch*), 104  
 UpgradedConstantGate (class in *qsearch.gates*), 41

## V

validate\_structure() (*qsearch.Gate* method), 85  
 validate\_structure() (*qsearch.gates.Gate* method), 22  
 validate\_structure() (*qsearch.gates.KroneckerGate* method), 44  
 validate\_structure() (*qsearch.gates.NonadjacentCNOTGate* method), 40  
 validate\_structure() (*qsearch.gates.ProductGate* method), 46  
 validate\_structure() (*qsearch.KroneckerGate* method), 108  
 validate\_structure() (*qsearch.NonadjacentCNOTGate* method), 103  
 validate\_structure() (*qsearch.ProductGate* method), 109

## X

XGate (class in *qsearch*), 86  
 XGate (class in *qsearch.gates*), 23  
 XXGate (class in *qsearch*), 101  
 XXGate (class in *qsearch.gates*), 38  
 XZXZGate (class in *qsearch*), 91  
 XZXZGate (class in *qsearch.gates*), 28

## Y

YGate (class in *qsearch*), 87  
 YGate (class in *qsearch.gates*), 24

## Z

ZGate (class in *qsearch*), 88  
 ZGate (class in *qsearch.gates*), 25  
 ZXZXZCNOTLinear (class in *qsearch.gatesets*), 49  
 ZXZXZCNOTLinear (in module *qsearch.gatesets*), 46  
 ZXZXZGate (class in *qsearch*), 90  
 ZXZXZGate (class in *qsearch.gates*), 27