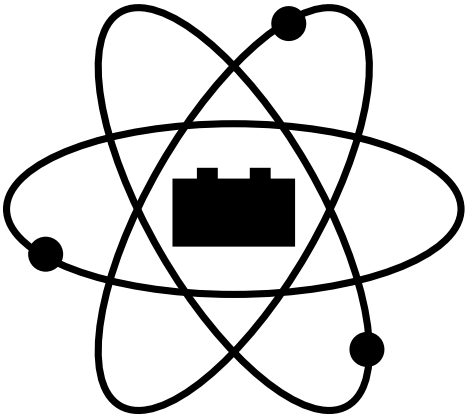


QuBricks

QuBricks Documentation

Release 1.0.0 RC

Matthew Wardrop

March 26, 2015

1 Introduction **3**

2 Installation **5**

3 Quick Start **7**
3.1 Getting Started . . . 7
3.2 Advanced Usage . . . 9
3.3 Operator Basics . . . 17

4 API: Brick and Mortar Classes **21**
4.1 QuantumSystem . . . 21
4.2 Operator . . . 28
4.3 StateOperator . . . 33
4.4 Measurement(s) & MeasurementWrapper . . . 36
4.5 Basis . . . 44
4.6 Integrator . . . 49

5 API: Wall Classes **55**
5.1 QuantumSystem Implementations . . . 55
5.2 Operator and OperatorSet Implementations . . . 55
5.3 StateOperator Implementations . . . 56
5.4 Measurement Implementations . . . 57
5.5 Basis Implementations . . . 58

6 API: Analysis Tools **61**
6.1 Perturbative Analysis . . . 61
6.2 Spectral Analysis . . . 64
6.3 Utilities . . . 65

Python Module Index **67**

Index **69**

Contents:

CHAPTER

ONE

INTRODUCTION

QuBricks is a toolkit for the analysis and simulation of quantum systems in Python. The primary goal of QuBricks is to facilitate insight into quantum systems; rather than to be the fastest or most efficient simulator. As such, the design of QuBricks is not especially geared toward very large or complicated quantum systems. It distinguishes itself from toolkits like QuTip (http://qutip.org) in that before simulations, everything can be expressed symbolically; allowing for analytic observations and computations. Simulations are nonetheless performed numerically, with various optimisations performed to make them more efficient.

Basic operations are unit-tested with reference to a simple two-level system.

CHAPTER

TWO

INSTALLATION

In most cases, installing this module is as easy as:

```
python2 setup.py install
```

If you run Arch Linux, you can instead run:

```
makepkg -i
```

CHAPTER

THREE

QUICK START

In this chapter, the basic behaviour of QuBricks is demonstrated in the context of some simple problems. For specific documentation on methods and their usage, please refer to the API documentation is subsequent chapters.

In the following, we assume that the following has been run in your Python environment.

```
>>> from qubricks import *
>>> from qubricks.wall import *
>>> import numpy as np
```

Note: QuBricks current only works in Python 2, since it depends on python-parameters which in turn is compatible only with Python 2.

3.1 Getting Started

In this section, we motivate the use of QuBricks in simple quantum systems. We will make use of the submodule qubricks.wall, which brings together many of the "bricks" that make up QuBricks into usable standalone tools.

Consider a single isolated electron in a magnetic field. Suppose that this magnetic field was composed of stable magnetic field along the Z axis B_z, and a noisy magnetic field along the X axis $B_x(t)$. The Hamiltonian describing the mechanics is then:

$$H = \mu_B \begin{pmatrix} B_z & B_x(t) \\ B_x(t) & -B_z \end{pmatrix},$$

where B_z is the magnetic field along z and $B_x(t) = \bar{B}_x + \tilde{B}_x(t)$ is a noise field along x centred on some nominal value $\bar{B}_x$ with a time-dependent noisy component $\tilde{B}_x(t)$.

Let us assume that the noise in $B_x(t)$ is white, so that:

$$\left\langle \tilde{B}_x(t)\tilde{B}_x(t') \right\rangle = D\delta(t-t'),$$

We can model such white noise using a Lindblad superoperator.

This is a simple system which can be analytically solved. Evolution under this Hamiltonian will lead to the electron's spin gyrating around an axis between Z and X (i.e. at an angle $\theta = \tan^{-1}(B_z/J_x)$ from the x-axis) at a frequency of $2\sqrt{B_z^2 + B_x^2}$. The effect of high frequency noise in B_x is to progressively increase the mixedness in the Z quadrature until such time as measurements of Z are unbiased. For example, when $B_z = 0$, the return probability for an initially up state is given by: $p = \frac{1}{2}(1 + \cos 2B_x t)$. Since $\left\langle \tilde{B}_x^{\,2} \right\rangle_{\text{avg}} = D/t$, we find by taylor expanding that: $\langle p \rangle = 1 - Dt$. A more careful analysis would have found that:

$$\langle p \rangle = \frac{1}{2}(1 + exp(-2Dt)).$$

It is possible to find approximations for a general θ, but we leave that as an exercise. Alternatively, you can take advantage of QuBricks to simulate these results for us.

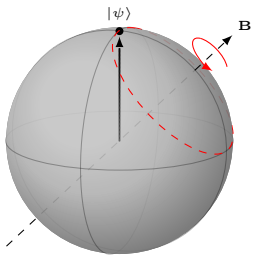


Figure 3.1: Dynamics of a single electron spin under a magnetic field aligned 45 degrees in the XZ plane.

For example, suppose we wanted to examine the case where $B_z = B_x$, as shown in figure 3.1. This can be simulated using the following QuBricks code:

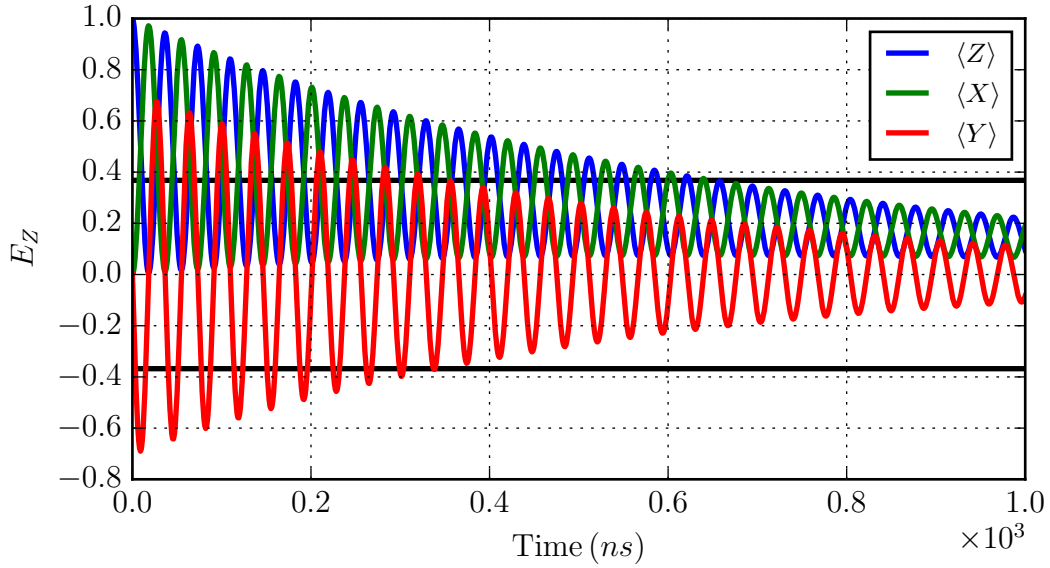
```
q = SimpleQuantumSystem(
  hamiltonian={
    'B':[[1,0],[0,-1]],
    'J':[[0,1],[1,0]]
  },
  parameters={
    'B':(40,'neV'),
    'J':(40,'neV'),
    'T_2': (400,'ns'),
    'D': (lambda T_2,c_hbar: 0.5*c_hbar**2/T_2, '{mu}J^2*ns')
  },
  measurements={
    'E': ExpectationMeasurement( [[1,0],[0,-1]], [[0,1],[1,0]], [[0,-1j],[1j,0]] ),
  },
  derivative_ops={
    'J_noise': LindbladStateOperator(coefficient='D',operator=[[0,1],[1,0]])
  }
)

ts = np.linspace(0,1e-6,1000)
r = q.measure.E.integrate(psi_0s=[ [1,0] ],times=ts, operators=['evolution','J_noise'])
```

We could then plot the results using *matplotlib*:

```
import matplotlib.pyplot as plt
from mplstyles import SampleStyle

style = SampleStyle()

with style:
        # If B=0, plot theoretical exponential decay curve
        if q.p.B == 0:
            p_D = lambda t,D,c_hbar: np.exp(-2*D/c_hbar**2*t)
            plt.plot(ts*1e9,q.p.range(p_D,t=ts),linestyle='--')

        plt.plot(r['time'][0]*1e9,r['expectation'][0,:,0],label="$\\left<Z\\right>$")
        plt.plot(r['time'][0]*1e9,r['expectation'][0,:,1],label="$\\left<X\\right>$")
        plt.plot(r['time'][0]*1e9,r['expectation'][0,:,2],label="$\\left<Y\\right>$")

        # Formatting options
        plt.grid()
```

```
        plt.legend(loc=0)
        plt.hlines([np.exp(-1),-np.exp(-1)],*plt.xlim())
        plt.xlabel('$\mathrm{Time}\,(ns)$')
        plt.ylabel("$E_Z$")

        style.savefig('results.pdf', polish=False, )
```

This would result in the following plot:



The above code takes advantage of several attributes and methods of *QuantumSystem* instances which may not be entirely clear. At this point, you can look them up in the API reference in subsequent chapters.

3.2 Advanced Usage

For more fine-grained control, one can subclass *QuantumSystem*, *Measurement*, *StateOperator* and *Basis* as necessary. For more information about which methods are available on these objects, refer to the API documentation below. The templates for subclassing are shown below.

3.2.1 QuantumSystem

```
from qubricks import QuantumSystem

class CustomSystem(QuantumSystem):
        '''
        Refer to the API documentation for `QuantumSystem` for more information.
        '''

        def init(self, **kwargs):
                '''
                This method can be used by subclasses to initialise the state
                of the `QuantumSystem` instance. Any excess kwargs beyond `parameters`
                passed to q `QuantumSystem` instance will be passed to this method.
```

```
            '''
            pass

    def init_parameters(self):
            '''
            This method can be used by subclasses to add any additional
            parameters required to describe the quantum system. If this
            method returns a dictionary, then it is used to update the
            parameters stored in the `Parameters` instance. This would
            be equivalent to:

            >>> system.p << system.init_parameters()

            Parameters may, of course, be set directly by this method, using
            (for example):

            >>> self.p.x = 1
            '''
            pass

    def init_hamiltonian(self):
            '''
            This method can be used by subclasses to initialise the Hamiltonian
            to be used to describe the Quantum System. The Hamiltonian can
            either be set directly in this method, using:

            >>> self.hamiltonian = <Operator or OperatorSet>

            Alternatively, if this method returns an Operator or OperatorSet
            object, then it will be set as the Hamiltonian for this QuantumSystem
            instance.
            '''
            pass

    def init_bases(self):
            '''
            This method can be used by subclasses to initialise the bases to be
            used by this instance of `QuantumSystem`. Bases can be added directly
            using the `add_basis` method; or, if this method returns a dictionary
            of `Basis` objects (indexed by string names), then they will be added
            as bases of this system.
            '''
            pass

    def init_states(self):
            '''
            This method can be used by subclasses to initialise named states, ensembles,
            and subspaces. This can be done directly, using the corresponding
            `add_state` and `add_subspace` methods. If a dictionary is returned, then
            it is assumed to be a dictionary of states indexed by names, which are then
            added to the system using `add_state`. Note that this assumes that the
            states are represented in the standard basis of this `QuantumSystem` object.
            '''
            pass

    def init_measurements(self):
            '''
            This method can be used by subclasses to initialise the measurements that
```

```
                will be used with this `QuantumSystem` instance. This can be done directly,
                using `add_measurement`; or, if this method returns a dictionary of `Measurement`
                objects indexed by string names, then they will be added as potential measurements
                of this quantum system.
                '''
                pass

        def init_derivative_ops(self):
                '''
                This method can be used by subclasses to initialise the `StateOperator` objects
                use to describe the time derivative of the evolution of the quantum system
                described by this object. Derivative operators may be added directly using
                `add_derivative_op`, or, if a dictionary of `StateOperator` objects is returned
                indexed with string names, then they are added as derivative operators of this objec
                If the operators depend on the hamiltonian or other properties of the quantum system,
                then the operators should be implemented in `get_derivative_ops` instead.

                This method should also initialise the default_derivative_ops property.
                '''
                pass

        def get_derivative_ops(self, components=None):
                '''
                This method can be used by subclasses to specify the `StateOperator` objects
                use to describe the time derivative of the evolution of the quantum system
                described by this object. These operators are added just before integration
                the operators described in `init_derivative_ops` and the 'evolution' operator
                describing Schroedinger evolution. Any properties of this
                `QuantumSystem` instance should not change before integration.

                :param components: The components activated in the Hamiltonian for this integration.
                :type components: iterable
                '''
                pass
```

3.2.2 Measurement

```
from qubricks import Measurement

class CustomMeasurement(Measurement):
        '''
        Refer to the API documentation `Measurement` for more details.
        '''

        def init(self, *args, **kwargs):
                '''
                This method should initialise the Measurement instance in whatever way
                is necessary to prepare the instance for use. Note that any arguments
                passed to the Measurement constructor will also be passed to this method.
                There is no restriction on the method signature for the init method.
                '''
                raise NotImplementedError("Measurement.init has not been implemented.")

        def measure(self, data=None, times=None, psi_0s=None, params={}, **kwargs):
                '''
                This method should return the value of a measurement as a numpy array with
```

```
            data type and shape as specified in `result_type` and `result_shape` respectively.

            .. note:: It is possible to return types other than numpy array and still
                    be compatible with iteration (see MeasurementWrapper) provided you overload
                    the `iterate_results_init` and `iterate_results_add` methods.

            Implementations of `measure` will typically be provided by integration data
            by a `MeasurementWrapper` instance (which will be a structured numpy array
            as returned by `Integrator.integrate) as the value for the `data` keyword.
            A consistent set of values for `times` and `psi_0s` will also be passed.

            .. note:: If an implementation of `measure` omits the `data` keyword, QuBricks
                    assumes that all integration required by the `measure` operator will be
                    performed internally. It can use the reference to a QuantumSystem
                    instance at `Measurement.system` for this purpose. If the `data` keyword
                    is present (for testing/etc), but pre-computed integration data is undesired,
                    override the `is_independent` method to return `True`. If external data
                    is *required*, then simply remove the default value of `data`.

            Apart from the required keywords: `data`, `times`, `psi_0s` and `params`; any additio
            keywords can be specified. Refer to the documentation of `MeasurementWrapper` to
            see how their values will filter through.

            .. note:: Although the keywords `times` and `psi_0s` are necessary, it is not
                    necessary to use these keywords. As such, Measurement operators need not
                    require an integration of the physical system.

            :param data: Data from a QuantumSystem.integrate call, or None.
            :type data: numpy.ndarray or None
            :param times: Sequence of times of interest.
            :type times: iterable
            :param psi_0s: The initial state vectors/ensembles with which to start integrating.
            :type psi_0s: str or iterable
            :param params: Parameter context to use during this measurement. Parameter types can
            :type params: dict
            :param kwargs: Any other keyword arguments not collected explicitly.
            :type kwargs: dict
            '''
            raise NotImplementedError("Measurement.measure has not been implemented.")


    def result_type(self, *args, **kwargs):
            '''
            This method should return an object suitable for use as the dtype
            argument in a numpy array constructor. Otherwise, no restrictions; other than that it
            agree with the data-type returned by `Measurement.measure`.

            This method will receive all arguments and keyword arguments passed to
            `iterate_results_init`, where it is used to initialise the storage of
            measurement results.
            '''
            raise NotImplementedError("Measurement.result_type has not been implemented.")

    def result_shape(self, *args, **kwargs):
            '''
            This method should return a tuple describing the shape of the numpy array to be retu
            by Measurement.measure.
```

```
                This method will receive all arguments and keyword arguments passed to
                `iterate_results_init`, where it is used to initialise the storage of
                measurement results.
                '''
                raise NotImplementedError("Measurement.result_shape has not been implemented.")

#         The following is only needed if you do not want to receive pre-computed integration data.
#          @property
#          def is_independent(self):
#                  '''
#                  `True` if this Measurement instance does all required integration internally (and s
#                  not receive pre-computed integration data). `False` otherwise. The default impleme
#                  `False`.
#                  '''
#                  return False
```

3.2.3 StateOperator

```
from qubricks import StateOperator

class CustomStateOperator(StateOperator):
        '''
        Refer to the API documentation for `StateOperator` for more information.
        '''

        def init(self, **kwargs):
                '''
                This method is called when StateOperator subclasses are
                initialised, which allows subclasses to set themselves up as appropriate.

                :param kwargs: The keyword arguments passed to the StateOperator constructor.
                :type kwargs: dict
                '''
                raise NotImplementedError("StateOperator.init is not implemented.")

        def __call__(self, state, t=0, params={}):
                '''
                StateOperator objects are called on states to effect some desired operation.
                States may be 1-dimensional (as state vectors) or 2-dimensional (as quantum ensembles
                and each subclass should specify in StateOperator.for_state and StateOperator.for_ens
                which kinds of states are supported.
                '''
                raise NotImplementedError("StateOperator.__call__ is not implemented.")

        def restrict(self, *indices):
                '''
                This method should return a new StateOperator restricted to the basis states
                with indices `indices`. See Operator.restrict for more information.
                '''
                raise NotImplementedError("StateOperator.restrict is not implemented.")

        def connected(self, *indices, **params):
                '''
                This method should return the list of basis state indices which would mix
                with the specified basis state indices `indices` under repeated operation of the
                StateOperator. See Operator.connected for more information.
                '''
```

```
                raise NotImplementedError("StateOperator.connected is not implemented.")

        def for_state(self):
                '''
                Should be True if the StateOperator supports 1D vector operations; and False otherwi
                '''
                raise NotImplementedError("StateOperator.for_state is not implemented.")

        def for_ensemble(self):
                '''
                Should be True if the StateOperator supports 2D ensemble operations; and False otherw
                '''
                raise NotImplementedError("StateOperator.process_args is not implemented.")

        def transform(self, transform_op):
                '''
                This method should transform all future operations on arbitrary input states
                according to the transformation `transform_op`. See Operator.transform
                for more information.
                '''
                raise NotImplementedError("StateOperator.transform is not implemented.")

#         The following method is optional
#          def collapse(self, *wrt, **params):
#                  '''
#                  This method is a stub to allow subclasses to simplify themselves when
#                  requested. If implemented, and Operators are used, the `collapse` method
#                  should be used on them also. See Operator.collapse for more information.
#
#                  Note that unless this method is overridden, no simplification occurs.
#                  '''
#                  return self
```

3.2.4 Basis

```
from qubricks import Basis

class CustomBasis(Basis):
        '''
        Refer to the API documentation of `Basis` for more details.
        '''

        def init(self, **kwargs):
                '''
                This method should do whatever is necessary to prepare the
                Basis instance for use. When this method is called by the
                Python __init__ method, you can use `Basis.dim` to access
                the raw value of `dim`. If `dim` is necessary to construct
                the operator, and it is not set, this method should raise an
                exception. All keyword arguments except `dim` and `parameters`
                passed to the `Basis` instance constructor will also be passed to
                this method.
                '''
                pass

        @property
        def operator(self):
```

```
            '''
            This method should return a two dimensional `Operator` object, with basis states as
            columns. The `Operator` object should use the `Parameters` instance provided by the
            Basis instance. The simplest way to ensure this is to use the `Basis.Operator` metho
            '''
            raise NotImplementedError("Basis operator has not been implemented.")

#       The following methods are optional; refer to `Basis` documentation for more information.
#        def state_info(self, state, params={}):
#                '''
#                This method (if implemented) should return a dictionary with more information
#                about the state provided. There are no further constraints upon what might be
#                returned.
#
#                :param state: The state about which information should be returned.
#                :type state: str or iterable
#                :param params: A dictionary of parameter overrides. (see `parameters.Parameters`)
#                :type params: dict
#                '''
#                return NotImplementedError("Basis.state_info has not been implemented.")
#
#        def state_toString(self, state, params={}):
#                '''
#                This method (if implemented) should return a string representation of the
#                provided state, which should then be able to be converted back into the same
#                state using `Basis.state_fromString`.
#
#                :param state: The state which should be represented as a string.
#                :type state: iterable
#                :param params: A dictionary of parameter overrides. (see `parameters.Parameters`)
#                :type params: dict
#                '''
#                raise NotImplementedError("Basis.state_toString has not been implemented.")
#
#        def state_fromString(self, string, params={}):
#                '''
#                This method (if implemented) should return the state as a numerical array that
#                is represented as a string in `string`. Calling `basis.state_toString` should then
#                return the same (or equivalent) string representation.
#
#                :param string: A string representation of a state.
#                :type state: str
#                :param params: A dictionary of parameter overrides. (see `parameters.Parameters`)
#                :type params: dict
#                '''
#                raise NotImplementedError("Basis.state_fromString has not been implemented.")
#
#        def state_latex(self, state, params={}):
#                '''
#                This method (if implemented) should return string that when compiled by
#                LaTeX would represent the state.
#
#                :param state: The state which should be represented as a string.
#                :type state: iterable
#                :param params: A dictionary of parameter overrides. (see `parameters.Parameters`)
#                :type params: dict
#                '''
#                raise NotImplementedError("Basis.state_latex has not been implemented.")
```

3.2.5 Integrator

In rare circumstances, you may find it necessary to subclass the *Integrator* class. Refer to the API documentation for more details on how to do this.

```
from qubricks import Integrator

class CustomIntegrator(Integrator):
    '''
    Refer to the API documentation for `Integrator` for more information.
    '''

    def _integrator(self, f, **kwargs):
        '''
        This method should return the object(s) necessary to perform
        the integration step. `f` is the the function which will return
        the derivative at each step.

        :param f: A function with signature f(t,y) which returns the derivative
            at time `t` for the state `y`. Note that the derivative that is returned
            is that of `_derivative`, but `f` also handles progress reporting.
        :type f: function
        :param kwargs: Any additional keyword arguments passed to the `Integrator`
            constructor.
        :type kwargs: dict
        '''
        pass

    def _integrate(self, integrator, initial, times=None, **kwargs):
        '''
        This method should perform the integration using `integrator`, and
        return a list of two-tuples, each containing
        a time and a corresponding state. The times should be those listed in times,
        which will have been processed into floats.

        :param integrator: Whichever value was returned from `_integrator`.
        :type integrator: object
        :param initial: The state at which to start integrating. Will be the type
            returned by `_state_internal2ode`.
        :type initial: object
        :param times: A sequence of times for which to return the state.
        :type times: list of float
        :param kwargs: Additional keyword arguments passed to `Integrator.start`
            and/or `Integrator.extend`.
        :type kwargs: dict
        '''
        pass

    def _derivative(self, t, y, dim):
        '''
        This method should return the instantaneous derivative at time `t`
        with current state `y` with dimensions `dim` (as returned by
        `_state_internal2ode`. The derivative should be expressed
        in a form understood by the integrator returned by `_integrator`
        as used in `_integrate`.

        :param t: The current time.
        :type t: float
```

```
        :param y: The current state (in whatever form is returned by the integrator).
        :type y: object
        :param dim: The original dimensions of the state (as returned by `_state_internal2ode`).
        :type dim: object
        '''
        pass

    def _state_internal2ode(self, state):
        '''
        This method should return a tuple of a state and its original dimensions in some form.
        The state should be in a form understandable by the integrator returned by `_integrator`,
        and the derivative returned by `_derivative`.

        :param state: The state represented as a numpy array. Maybe 1D or 2D.
        :type state: numpy.ndarray
        '''
        pass

    def _state_ode2internal(self, state, dimensions):
        '''
        This method should restore and return the state (currently represented in the form used by t
        returned by `_integrator`) to its representation as a numpy array using the
        `dimensions` returned by `_state_internal2ode`.

        :param state: The state to re-represented as a numpy array.
        :type state: object
        :param dimensions: The dimensions returned by `_state_internal2ode`.
        :type dimensions: object
        '''
        pass
```

3.3 Operator Basics

One class that is worth discussing in more detail is *Operator*, which is among the most important "bricks" in the QuBricks library. It represents all of the two-dimensional linear operators used in QuBricks. The Operator object is neither directly a symbolic or numeric representation of an operator; but can be used to generate both.

Consider a simple example:

```
>>> op = Operator([[1,2],[3,4]])
>>> op
<Operator with shape (2,2)>
```

To generate a matrix representation of this object for inspection, we have two options depending upon whether we want a symbolic or numeric representation.

```
>>> op() # Numeric representation as a NumPy array
array([[ 1.+0.j, 2.+0.j],
       [ 3.+0.j, 4.+0.j]])
>>> op.symbolic() # Symbolic representation as a SymPy matrix
Matrix([
[1, 2],
[3, 4]])
```

In this case, there is not much difference, of course, since there are no symbolic parameters used.

Creating an Operator object with named parameters can be done in two ways. Either you must create a dictionary

relating parameter names to matrix forms, or you can create a SymPy symbolic matrix. In both cases, one then passes this to the Operator constructor. For example:

```
>>> op = Operator('B':[[1,0],[0,-1]], 'J':[[0,1],[1,0]])
>>> op.symbolic()
Matrix([
[B, J],
[J, -B]])
>>> op.symbolic(J=10)
Matrix([
[ B, 10],
[10, -B]])
>>> op()
ValueError: Operator requires use of Parameters object; but none specified.
```

When representing Operator objects symbolically, we can override some parameters and perform parameter substitution. We see that attempting to generate a numeric representation of the Operator object failed, because it did not know how to assign a value to B and J. Normally, Operator objects will have a reference to a *Parameters* instance (from *python-parameters*) passed to it in the constructor phase, for which these parameters can be extracted. This will in most cases be handled for you by QuantumSystem (see *QuantumSystem* in the API chapters), but for completeness there are two keyword arguments you can pass to Operator instances: *parameters*, which shold be a reference to an existing Parameters instance, and *basis*, which should be a reference to an existing Basis object or None (see *Basis* in the API chapters). For now, let us manually add it for demonstration purposes.

```
>>> from parameters import Parameters
>>> p = Parameters()
>>> p(B=2,J=1)
< Parameters with 2 definitions >
>>> op = Operator('B':[[1,0],[0,-1]], 'J':[[0,1],[1,0]],parameters=p)
>>> op()
array([[ 2.+0.j,  1.+0.j],
       [ 1.+0.j, -2.+0.j]])
>>> op(J=10,B=1)
array([[  1.+0.j, 10.+0.j],
       [ 10.+0.j, -1.+0.j]])
```

We see in the above that we can take advantage of temporary parameter overrides for numeric representations too [note that a parameters instance is still necessary for this].

The *Parameters* instance allows one to have parameters which are functions of one another, which allows for time and/or context dependent operators.

Operator objects support basic arithmetic: addition, subtraction, and multiplication using the standard python syntax. The inverse operation can be performed using the inverse method:

```
>>> op.inverse()
```

The Kronecker tensor product can be applied using the tensor method:

```
>>> op.tensor(other_op)
```

To apply an Operator object to a vector, you can either use the standard inbuilt multiplication operations, or use the slightly more optimised apply method.

If you are only interested in how certain parameters affect the operator, then to improve performance you can "collapse" the Operator down to only include variables which depend upon those variables.

```
>>> op.collapse('t',J=1)
```

The result of the above command would substitute all variables (with a parameter override of $J = 1$) that do not depend upon t with their numerical value, and then perform various optimisations to make further substitutions more efficient. This is used, for example, by the integrator.

The last set of key methods of the Operator object are the connected and restrict methods. Operator.connected will return the set of all indicies (of the basis vectors in which the Operator is represented) that are connected by non-zero matrix elements, subject to the provided parameter substitution. Note that this comparison is done with the numerical values of the parameters.

```
>>> op = Operator('B':[[1,0],[0,-1]], 'J':[[0,1],[1,0]],parameters=p)
>>> op.connected(0)
0,1
>>> op.connected(0, J=0)
0
```

The restrict method returns a new Operator object which keeps only the entries in the old Operator object which correspond to the basis elements indicated by the indicies.

```
>>> op = Operator('B':[[1,0],[0,-1]], 'J':[[0,1],[1,0]],parameters=p)
>>> op.restrict(0)
<Operator with shape (1, 1)>
>>> op.symbolic()
Matrix([[B]])
```

For more detail, please refer to the API documentation for *Operator*.

CHAPTER

FOUR

API: BRICK AND MORTAR CLASSES

4.1 QuantumSystem

class qubricks.system.**QuantumSystem**(*parameters=None*, ***kwargs*)
Bases: object

The *QuantumSystem* class is used to describe particular quantum systems, and provides utility functions which assists in the analysis of these systems. While it is possible to directly instantiate a *QuantumSystem* object, and to programmatically add the description of the quantum system of interest, it is more common to subclass it or to use a ready-made subclass from *qubricks.wall.systems*.

Parameters

- **parameters** (*Parameters, str or None*) – An object used to initialise a Parameters instance for use in this object.
- **kwargs** (*dict*) – Additional keyword arguments to pass to *init*, which may be useful for subclasses.

Specifying parameters: Every QuantumSystem instance requires access to a *Parameters* instance in order to manage the physical quantities associated with the represented quantum system. When instantiating *QuantumSystem* or one of its subclasses, this is managed by passing as a value to the *parameters* keyword one of the following: - A *Parameters* instance, which is then used directly. - A string, in which case QuantumSystem attempts to load a parameters

configuration from the path indicated in the string.

- *None*, in which case an empty *Parameters* instance is constructed.

Subclassing: If you choose to subclass *QuantumSystem* in order to represent a quantum system of interest, it should only be necessary to implement some or all of the following methods:

- init(self, **kwargs)
- init_parameters(self)
- init_hamiltonian(self)
- init_bases(self)
- init_states(self)
- init_measurements(self)
- init_derivative_ops(self)
- get_derivative_ops(self,components=None)

Documentation pertaining to the behaviour of these methods is available below. Importantly, these methods will always be called in the order given above.

Integration and Measurement: Perhaps among the most common operations you might want to perform with your quantum system is simulated time-evolution and measurement.

Integration is handled as documented in the *QuantumSystem.integrate* method.

Measurement is handled using the *measure* attribute, which points to a *Measurements* object. For example, if a measurement named 'fidelity' has been added to this QuantumSystem, you could use:

```
>>> system.measure.fidelity(...)
```

For more, see the documentation for *Measurements*.

H (**components*)
This method returns an *Operator* instance representing the Hamiltonian of the system. If *components* is specified, only the components of the OperatorSet object listed are included, otherwise the Operator object returns its default set. Note that if the Hamiltonian object is simply an Operator, the Operator is simply returned as is.

Parameters **components** (*tuple*) – Sequence of component names.

Operator (*components*, *basis=None*, *exact=False*)
This method is a shorthand way of creating an *Operator* instance that shares the same *Parameters* instance as this *QuantumSystem*. This is shorthand for:

```
>>> Operator(components, parameters=self.p, basis=self.basis(basis) if basis is not None els
```

For more documentation, refer to *Operator*.

OperatorSet (*operatorMap*, *defaults=None*)
This method is a shorthand way of creating an *OperatorSet* instance. This method first checks through the values of *operatorMap* and converts anything that is not an *Operator* to an *Operator* using:

```
>>> system.Operator(operatorMap[key])
```

Then, it creates an OperatorSet instance:

```
>>> OperatorSet(operatorMap, defaults=defaults)
```

For more documentation, see *Operator* and *OperatorSet*.

add_basis (*name*, *basis*)
This method is used to add a basis. The first *StandardBasis* instance to be added will become the default basis used to describe this *QuantumSystem*.

Parameters

- **name** (*str*) – The name used to reference this basis in the future.
- **basis** (*Basis*) – A *Basis* instance to be associated with the above name.

add_derivative_op (*name*, *state_op*)
This method adds an association between a string *name* and a *StateOperator* object, for use as a derivative operator.

Parameters

- **name** (*str*) – The name to be used to refer to the provided *StateOperator*.
- **state_op** (*StateOperator*) – The *StateOperator* to be stored.

add_measurement (*name*, *measurement*)
This method add a *Measurement* instance to the *Measurements* container associated with this Quantum System object. It can then be called using:

```
>>> system.measure.<name>
```

See *Measurements* for more information.

add_state (*name*, *state*, *basis=None*, *params={}*)
This method allows a string name to be associated with a state. This name can then be used in *QuantumSystem.state* to retrieve the state. States are assumed to be in the basis specified, and are transformed as necessary to be in the standard basis of this *QuantumSystem* instance.

Parameters

- **name** (*str*) – String name to be associated with this state.
- **state** (*Operator or iterable object convertible to numpy array*) – State or ensemble to be recorded.
- **basis** (*str, Basis or None*) – The basis of the input *state*.
- **params** (*dict*) – The parameter overrides to use during basis transformation.

add_subspace (*name*, *subspace*, *basis=None*, *params={}*)
This method adds a new association between a string and a sequence of states (which can be in turn be the names of named states).

Parameters

- **name** (*str*) – The name to associate with this subspace.
- **subspace** (*str or list of str, Operators or sequences convertible to numpy array*) – The input subspace to be stored.
- **basis** (*str, Basis or None*) – The basis in which the subspace is represented.
- **params** (*dict*) – The parameter overrides to use during basis transformation.

bases
A sorted list of the names of the bases configured for this *QuantumSystem* instance.

basis (*basis*)
This method returns the *Basis* object associated with the provided name. If *basis* is a *Basis* object, then it is simply returned; and if *basis* is None, a *StandardBasis* is returned (or if a StandardBasis instance has been added to this instance, then it is returned instead).

Parameters **basis** – A name of a basis, a *Basis* instance, or None.

default_derivative_ops
The list (or sequence) of names corresponding to the derivative operators that will be used by default (if no operator names are otherwise supplied).

derivative_ops (*ops=None*, *components=None*)
This method returns a dictionary of named *StateOperator* objects, which are used to calculate the instantaneous time derivative by *QuantumSystem.integrate* and related methods. This method picks upon the operators created by *init_derivative_ops* and *get_derivative_ops*, as well as the default 'evolution' operator which describes evolution under the Schroedinger equation.

Parameters

- **ops** (*iterable of strings*) – A sequence of operators to include in the returned dictionary.
- **components** (*iterable of strings*) – A sequence of component names to enable in the *OperatorSet* describing the Hamiltonian (if applicable).

dim
The dimension of this *QuantumSystem*; or equivalently the number of basis vectors.

ensemble(*state*, *input=None*, *output=None*, *threshold=False*, *evaluate=True*, *params={}*)
This method returns a ensemble 2D vector (numpy array) that is associated with the input *state*. If the state associated with *state* is not an ensemble, then the outer product is taken and returned. Just like *QuantumSystem.state*, this method allows basis conversion of states.

Parameters

- **state** (*str or Operator or sequence convertible to numpy array*) – The input state. If this is a string, this should refer to a named state as in *states* or *ensembles*.
- **input** (*str, Basis or None*) – The basis of the input states.
- **output** (*str, Basis or None*) – The basis into which this method should convert output states.
- **threshold** (*bool or float*) – Parameter to control thresholding (see *Basis.transform* documentation)
- **evaluate** (*bool*) – If input type is Operator, whether the Operator should be numerically evaluated.
- **params** (*dict*) – Parameter overrides to use during evaluation and basis transformation.

ensembles
A sorted list of names that are associated with ensembles. Ensembles are added using *QuantumSystem.add_state*, where a 2D state is automatically identified as an ensemble.

get_derivative_ops(*components=None*)
This method can be used by subclasses to specify the *StateOperator* objects use to describe the time derivative of the evolution of the quantum system described by this object. These operators are added just before integration the operators described in *init_derivative_ops* and the 'evolution' operator describing Schroedinger evolution. Any properties of this *QuantumSystem* instance should not change before integration.

Parameters **components** (*iterable*) – The components activated in the Hamiltonian for this integration.

get_integrator(*initial=None*, *input=None*, *output=None*, *threshold=False*, *components=None*, *operators=None*, *time_ops={}*, *params={}*, *integrator='RealIntegrator'*, ***kwargs*)
This method is shorthand for manually creating an *Integrator* instance. It converts all operators and states into a consistent basis and form for use in the integrator.

Parameters

- **initial** (*iterable*) – A sequence of initial states to use (including string names of known states and ensembles; see *QuantumSystem.state*).
- **input** (*str, Basis or None*) – Basis of input states (including string name of stored Basis; see *QuantumSystem.basis*).
- **output** (*str, Basis or None*) – Basis to use during integration (including string name of stored Basis; see *QuantumSystem.basis*)
- **threshold** (*bool or float*) – Parameter to control thresholding during basis transformations (see *Basis.transform*).
- **components** (*iterable of str*) – A sequence of component names to enable for this *Integrator* (see *QuantumSystem.H*).

- **operators** (*iterable of str*) – A sequence of operator names to use during integration (see *QuantumSystem.derivative_ops*). Additional operators can be added using *Integrator.add_operator* on the returned *Integrator* instance.
- **time_ops** (*dict*) – A dictionary of *StateOperator* instances indexed by times (see *Integrator.time_ops*).
- **params** (*dict*) – Parameter overrides to use during basis transformations and integration.
- **integrator** (*str or classobj*) – The integrator class to use as a Class; either as a string (in which case it is imported from *qubricks.wall*) or as a Class to be instantiated.
- **kwargs** (*dict*) – Additional keyword arguments to pass to *Integrator* constructor.

hamiltonian

The *Operator* or *OperatorSet* object used to describe this quantum system. If not yet specified, this will be *None*. The Hamiltonian can be specified using:

```
>>> system.hamiltonian = <Operator or OperatorSet>
```

Note: The Hamiltonian is expected to be represented such that the basis vectors form the standard basis. This is, in practice, not a limitation; but is important to remember when transforming bases.

---

init(***kwargs*)

This method can be used by subclasses to initialise the state of the *QuantumSystem* instance. Any excess kwargs beyond *parameters* passed to q *QuantumSystem* instance will be passed to this method.

init_bases()

This method can be used by subclasses to initialise the bases to be used by this instance of *QuantumSystem*. Bases can be added directly using the *add_basis* method; or, if this method returns a dictionary of *Basis* objects (indexed by string names), then they will be added as bases of this system.

init_derivative_ops()

This method can be used by subclasses to initialise the *StateOperator* objects use to describe the time derivative of the evolution of the quantum system described by this object. Derivative operators may be added directly using *add_derivative_op*, or, if a dictionary of *StateOperator* objects is returned indexed with string names, then they are added as derivative operators of this object. If the operators depend on the hamiltonian or other properties of the quantum system, then the operators should be implemented in *get_derivative_ops* instead.

This method should also initialise the default_derivative_ops property.

init_hamiltonian()

This method can be used by subclasses to initialise the Hamiltonian to be used to describe the Quantum System. The Hamiltonian can either be set directly in this method, using:

```
>>> self.hamiltonian = <Operator or OperatorSet>
```

Alternatively, if this method returns an Operator or OperatorSet object, then it will be set as the Hamiltonian for this QuantumSystem instance.

init_measurements()

This method can be used by subclasses to initialise the measurements that will be used with this *QuantumSystem* instance. This can be done directly, using *add_measurement*; or, if this method returns a dictionary of *Measurement* objects indexed by string names, then they will be added as potential measurements of this quantum system.

init_parameters()

This method can be used by subclasses to add any additional parameters required to describe the quantum

system. If this method returns a dictionary, then it is used to update the parameters stored in the *Parameters* instance. This would be equivalent to:

```
>>> system.p << system.init_parameters()
```

Parameters may, of course, be set directly by this method, using (for example):

```
>>> self.p.x = 1
```

init_states()
This method can be used by subclasses to initialise named states, ensembles, and subspaces. This can be done directly, using the corresponding *add_state* and *add_subspace* methods. If a dictionary is returned, then it is assumed to be a dictionary of states indexed by names, which are then added to the system using *add_state*. Note that this assumes that the states are represented in the standard basis of this *QuantumSystem* object.

integrate(*times*, *psi_0s*, ***kwargs*)
This method constructs an *Integrator* instance with initial states defined as *psi_0s* using *get_integrator*, and then calls *start* on that instance with times specified as *times*.

Parameters

- **times** (*iterable of floats or str*) – The times for which to return the instantaneous state. All values are passed through the *Parameters* instance, allowing for times to be expressions of parameters.
- **psi_0s** (*list or tuple of numpy arrays*) – A sequence of initial states (or ensembles).

This method is equivalent to: >>> system.get_integrator(initial=psi_0s, **kwargs).start(times)

For more documentation, see *get_integrator* and *Integrator.start*.

measurements
A sorted list of the names of the measurements associated with this *QuantumSystem*.

p
A reference to the *Parameters* instance used by this object.

show()
This method prints (to stdout) a basic overview of the *QuantumSystem* that includes: - the dimension of the model - the representation of the *Parameters* instance - the names of any bases - the names of any states - the names of any ensembles - the names of any subspaces - the names of any derivative operators

state(*state*, *input=None*, *output=None*, *threshold=False*, *evaluate=True*, *params={}*)
This method returns a state vector (numpy array) that is associated with the input *state*. As well as retrieving named states from storage, this method also allows basis conversions of the state. Note that states can be 1D or 2D (states or ensembles).

Parameters

- **state** (*str or Operator or sequence convertible to numpy array*) – The input state. If this is a string, this should refer to a named state as in *states*.
- **input** (*str, Basis or None*) – The basis of the input states.
- **output** (*str, Basis or None*) – The basis into which this method should convert output states.
- **threshold** (*bool or float*) – Parameter to control thresholding (see *Basis.transform* documentation)
- **evaluate** (*bool*) – If input type is Operator, whether the Operator should be numerically evaluated.

- **params** (*dict*) – Parameter overrides to use during evaluation and basis transformation.

state_fromString(*state*, *input=None*, *output=None*, *threshold=False*, *params={}*)
This method creates a state object from a string representation, as interpreted by the *Basis.state_fromString* method. Otherwise, this method acts as the *QuantumSystem.state* method.

Parameters

- **state** (*str*) – The string representation of the state.
- **input** (*str, Basis or None*) – The basis to use to interpret the string.
- **output** (*str, Basis or None*) – The basis into which this method should convert output states.
- **threshold** (*bool or float*) – Parameter to control thresholding (see *Basis.transform* documentation)
- **params** (*dict*) – Parameter overrides to use during evaluation and basis transformation.

state_projector(*state*, ***kwargs*)
This method returns a projector onto the state provided. This method is equivalent to:

```
>>> system.subspace_projector([state], **kwargs)
```

Refer to *subspace_projector* for more information.

state_toString(*state*, *input=None*, *output=None*, *threshold=False*, *params={}*)
This method create a string representation of a state object, using *Basis.state_toString*. Otherwise, this method acts like *QuantumSystem.state*.

Parameters

- **state** (*str or Operator or sequence convertible to numpy array*) – The input state. If this is a string, this should refer to a named state as in *states*.
- **input** (*str, Basis or None*) – The basis of the input states.
- **output** (*str, Basis or None*) – The basis into which this method should convert output states, and which should be used to create the string representation.
- **threshold** (*bool or float*) – Parameter to control thresholding (see *Basis.transform* documentation)
- **params** (*dict*) – Parameter overrides to use during evaluation and basis transformation.

Converts a state object to its string representation, as interpreted by the Basis.state_toString method. As with QuantumSystem.state, basis conversions can also be done.

states
A sorted list of the names of the states configured for this *QuantumSystem* instance.

subspace(*subspace*, *input=None*, *output=None*, *threshold=False*, *evaluate=True*, *params={}*)
This method returns the subspace associated with the input *subspace*. A subspace is a list of states (but *not* ensembles). The subspace is the span of the states. Otherwise, this method acts like *QuantumSystem.state*.

Parameters

- **subspace** (*str or list of str, Operators or sequences convertible to numpy array*) – The input subspace. If this is a string, this should refer to a named state as in *states*.
- **input** (*str, Basis or None*) – The basis of the input states.
- **output** (*str, Basis or None*) – The basis into which this method should convert output states.

- **threshold** (*bool or float*) – Parameter to control thresholding (see *Basis.transform* documentation)
- **evaluate** (*bool*) – If input type is Operator, whether the Operator should be numerically evaluated.
- **params** (*dict*) – Parameter overrides to use during evaluation and basis transformation.

subspace_projector(*subspace*, *input=None*, *output=None*, *invert=False*, *threshold=False*, *evaluate=True*, *params={}*)

This method returns a projector onto the provided subspace (which can be the name of a named subspace, or a list of named states. If *invert* is *True*, then the projector returned is onto the subspace orthogonal to the provided one.

Parameters

- **subspace** (*str or list of str, Operators or sequences convertible to numpy array*) – The input subspace. If this is a string, this should refer to a named state as in *states*.
- **input** (*str, Basis or None*) – The basis of the provided subspace.
- **output** (*str, Basis or None*) – The basis into which this method should convert output subspaces.
- **invert** (*bool*) – *True* if the projector returned should be onto the subspace, and *False* if the projector should be off the subspace.
- **threshold** (*bool or float*) – Parameter to control thresholding (see *Basis.transform* documentation)
- **evaluate** (*bool*) – If input type is Operator, whether the Operator should be numerically evaluated.
- **params** (*dict*) – Parameter overrides to use during evaluation and basis transformation.

subspaces

A sorted list of named subspaces. The subspaces can be extracted using the *QuantumSystem.subspace* method.

use_ensemble(*ops=None*, *ensemble=False*)

This method is used to check whether integration should proceed using ensembles or state vectors. It first checks which kind of evolution is supported by all of the *StateOperators*, as heralded by the *StateOperator.for_state* and *StateOperator.for_ensemble* methods. It then checks that there is a match between the type of state being used and the type of states supported by the operators. Note that if *ensemble* is *False*, indicating that the state is a ordinary state vector, then it is assumed that if this method returns *True*, indicating ensemble evolution is to be used, that the states will be converted using the outer-product with themselves to ensembles. Note that this is already done by *get_integrator*.

Parameters

- **ops** (*iterable*) – A sequence of Operator objects or names of Operators. The list can be mixed.
- **ensemble** (*bool*) – *True* one or more of the input states are to be ensembles. *False* otherwise.

4.2 Operator

class qubricks.operator.**Operator**(*components*, *parameters=None*, *basis=None*, *exact=False*)

Bases: object

Operator is the base class used by QuBricks to facilitate dynamic generation of matrices (with partial support for n-dimensional operations when *exact* is False). Operator objects wrap around a dictionary of "components" which are indexed by a function of parameters. When evaluated, Operator objects evaluate the parameters using a Parameters instance, and then add the various components together. Operator objects support arithmetic (addition, subtraction and multiplication); basis transformations; restriction to a subspace of the basis; inversion (where appropriate); and the tensor (Kronecker) product.

Parameters

- **components** (*dict or numpy.ndarray or sympy.Matrix*) – Specification of the operator form
- **parameters** (*parameters.Parameters*) – Parameters instance
- **basis** (*Basis or None*) – The basis in which Operator is represented
- **exact** (*bool*) – `True` if Operator is to maintain an exact representation of numbers.

Operator Specifications: The first and simplest way to construct an Operator object is to wrap an Operator around a pre-existing numpy array or sympy Matrix.

```
>>> p = Parameters()
>>> a = numpy.array([[1,2,3],[4,5,6]])
>>> op = Operator(a, parameters=p)
>>> x,y,z = sympy.var('x,y,z')
>>> b = sympy.Matrix([[x,y**2,z+x],[y+z,x**2,x*y*z]])
>>> op2 = Operator(b, parameters=p)
```

The first example above demonstrates wrapping a static numeric matrix into an Operator object; while the second demonstrates a conversion of an already symbolic operator into an Operator object.

The other way to define specify the form of an Operator object is to create a dictionary with keys of (functions of) parameters and values corresponding to the representation of those parameters in the matrix/array. For example:

```
>>> d = {'x':[[1,0,0],[0,0,0]], 'sin(y)':[[0,0,0],[0,0,3]], None: [[1,1,1],[2,2,2]]}
>>> op = Operator(d, parameters=p)
```

The above code snippet represents the below matrix:

```
[x+1, 1 ,       1      ]
[ 2 , 2 , 2 + sin(y) ]
```

Evaluating an Operator: Operator objects can be evaluated to numeric matricies (whereby parameters) are substituted in from the Parameters instance, or symbolic sympy.Matrix objects.

Numeric evaluation looks like calling the operator:

```
>>> d = {'x':[[1,0,0],[0,0,0]], 'sin(y)':[[0,0,0],[0,0,3]], None: [[1,1,1],[2,2,2]]}
>>> op = Operator(d, parameters=p)
>>> op(x=2, y=0)
array([[ 3.+0.j,  1.+0.j,  1.+0.j],
       [ 2.+0.j,  2.+0.j,  2.+0.j]])
```

Note: Providing parameters as shown above makes use of functionality in the Parameters instance, where they are called parameter overrides. Consequently, you can also supply functions of other parameters here. You can also supply united quantities. See the Parameters documentation for more.

Symbolic evaluation looks like:

```
>>> op.symbolic()
Matrix([
[x + 1, 1,             1],
[    2, 2, 3*sin(y) + 2]])
```

Parameters can also be specified during symbolic evaluation:

```
>>> op.symbolic(y=0)
Matrix([
[x + 1, 1, 1],
[    2, 2, 2]])
```

Note: Parameter substitution during symbolic evaluation makes use of the *subs* function of sympy objects. It too supports substitution with functions of parameters. See the sympy documentation for more.

It is also possible to *apply* an Operator to a vector without ever explicitly evaluating the Operator. This may lead to faster runtimes. See the documentation for the *apply* method.

Operator arithmetic: Operator objects can be added, subtracted and multiplied like any other Pythonic numeric object.

Supported operations are: - Addition: `op1+op2` - Subtraction: `op1-op2` - Multiplication: `op1*op2` - Scaling: `k*op1`, with *k* a scalar constant.

The rest of the functionality of the Operator object is described in the method documentation below.

`apply` (*state*, *symbolic=False*, *left=True*, *params={}*)

This method returns the object resulting from multiplication by this Operator without ever fully constructing the Operator; making it potentially a little faster.

Parameters

- **state** (*numpy.array or object*) – An array with suitable dimensions for being pre- (or post-, if left is False) multipled by the Operator represented by this object.
- **symbolic** (*bool*) – `True` if multiplication should be done symbolically using sympy, and `False` otherwise (uses numpy).
- **left** (*bool*) – `True` if the state should be multiplied from the left; and `False` otherwise.
- **params** (*dict*) – A dictionary of parameter overrides.

For example:

```
>>> op = Operator([[1,2],[3,4]])
>>> op.apply([1,2])
array([  5.+0.j,  11.+0.j])
```

`basis`

A reference to the current Basis of the Operator; or None if it has no specified Basis.

`change_basis` (*basis=None*, *threshold=False*, *params={}*)

This method returns a copy of this Operator object expressed in the new basis. The threshold parameter allows elements in the resulting Operator which differ from zero only by numerical noise to be set to zero. For more information, refer to the documentation for Basis.transform.

Parameters

- **basis** (*Basis or None*) – The basis in which to represent this Operator.
- **threshold** (*bool or float*) – A boolean indicating whether to threshold to minimise numerical error, or a float indicating the threshold level.

- **params** (*dict*) – Parameter overrides to use during the basis transformation.

Returns A reference to the transformed Operator.

clean(*threshold*)
This method zeroes out all elements of the components which are different from zero only by a magnitude less than *threshold*. One must use this function with caution, as it does not take into account the value of the parameter multiplying the matrix form.

Parameters

- **threshold** – A threshold value.
- **type** – float

collapse(**wrt*, ***params*)
This method returns a new Operator object that is a simplification of this one. It collapses and simplifies this Operator object by assuming certain parameters are going to be fixed and non-varying. As many parameters as possible are collapsed into the constant component of the operator. All other entries are analytically simplified as much as possible. If no parameters are specified, then only the simplification is performed. A full collapse to a numerical matrix should be achieved by evaluating it numerically, as described in the class description.

Parameters

- **wrt** (*tuple*) – A sequence of parameter names.
- **params** (*dict*) – Parameter overrides.

For example:

```
>>> op.collapse('x',y=1)
```

This will lead to a new Operator being formed which is numeric except for terms involving *x*, when *y* is set to *1*.

connected(**indices*, ***params*)
This method returns a set of indices that represents the rows/columns that mix with the specified indices if this operator were to be multiplied by itself. This method requires that the Operator object be square.

Parameters

- **indices** (*tuple*) – A sequence of zero-indexed indices to test for connectedness.
- **params** (*dict*) – A parameter override context.

For example:

```
>>> op.connected(1,2,3,x=12,y=(3,'ms'))
{1,2,3}
```

The above output would suggest that in the context of x and y set as indicated, the specified subspace is closed under repeated self-multiplication of the Operator.

exact
A boolean value indicating whether the Operator should maintain exact representations of numbers.

inverse()
This method computes and returns the inverse of the Operator object. This may be very slow. If you do not need a symbolic inversion, then simply numerically evaluate the Operator object and take a numerical inverse using numpy.

p
A reference to the internal Parameter object.

restrict(**indices*)
This method returns a copy of the Operator restricted to the basis elements specified as row/column indices. This method requires that the shape of the Operator is square.

Parameters **indices** (*tuple*) – A sequence of zero-indexed indices which correspond to the rows/columns to keep.

For example:

```
>>> op = Operator([[1,2,3],[4,5,6],[7,8,9]]).restrict(1,2)
>>> op()
array([[ 5.+0.j,  6.+0.j],
       [ 8.+0.j,  9.+0.j]])
```

shape
A tuple representing the dimensions of the Operator.

symbolic(***params*)
This method returns a symbolic representation of the Operator object, as documented in the general class documentation.

Parameters **params** (*dict*) – A dictionary of parameter overrides.

For example:

```
>>> op.symbolic(x=2,y='sin(x)')
```

tensor(*other*)
This method returns a new Operator object that is the component-wise tensor (or Kronecker) product of this Operator with *other*.

Parameters **other** (*Operator*) – Another Operator with which to perform the tensor product

transform(*transform_op=None*)
This method returns a copy of this Operator instance with its components transformed according to the supplied transform_op function. This can effect a basis transformation without providing any information as to the basis of the new Operator. If you want to transform basis, it is probably better you use: Operator.change_basis.

Parameters **transform_op** (*callable*) – A function which maps numpy.ndarray and sympy.MatrixBase instances to themselves, potentially with some transformation.

class qubricks.operator.**OperatorSet**(*components=None*, *defaults=None*, ***kwargs*)
Bases: object

OperatorSet objects a container for multiple Operator objects, such that one can construct different combinations of the elements of the OperatorSet at runtime. This is useful, for example, if you have a Hamiltonian with various different couplings, and you want to consider various combinations of them.

Parameters

- **components** (*dict of Operator instances or None*) – A dictionary of Operator objects, indexed by a string representing the name of the corresponding Operator object.
- **defaults** – A list of component names which should be compiled into an operator when a custom list is not supplied. If defaults is None, then all components are used.

Type list of str or None

Creating an OperatorSet instance:

```
>>> ops = OperatorSet ( components = { 'op1': Operator(...),
                                       'op2': Operator(...) } )
```

Extracting Operator instances: Usually, one wants to call OperatorSet objects, with a list of keys to be compiled into a single Operator object. e.g. operatorset('name1','name2',...) . For example:

```
>>> ops('op1') # This is just op1 on its own
>>> ops('op1','op2') # This is the sum of op1 and op2
>>> ops() # Same as above, since no defaults provided.
```

Individual components can also be accessed using: `operatorset['name']`.

Subclassing OperatorSet: To subclass *OperatorSet* simply override the *init* method to construct whichever *Operators* you desire. You should initialise *OperatorSet.components* to be a dictionary; but otherwise, there are no constraints. Note that you can add elements to an OperatorSet without subclassing it.

apply(*state*, *symbolic=False*, *left=True*, *params=None*, *components=None*)

This method applies the Operator.apply method from each of the stored Operators, and sums up their results. The only difference from the Operator.apply method is the *components* keyword, which allows you to specify which operators are used. If *components* is not specified, then the default components are used.

Parameters

- **state** (*numpy.array or object*) – An array with suitable dimensions for being pre- (or post-, if left is False) multipled by the Operator represented by this object.
- **symbolic** (*bool*) – `True` if multiplication should be done symbolically using sympy, and `False` otherwise (uses numpy).
- **left** (*bool*) – `True` if the state should be multiplied from the left; and `False` otherwise.
- **params** (*dict*) – A dictionary of parameter overrides.
- **components** (*iterable*) – A list of components to use.

init()

Subclasses may use this hook to initialise themselves. It is called after *OperatorSet.components* and *OperatorSet.defaults* are set to their passed values, with *Operator.components* guaranteed to be a dictionary.

class qubricks.operator.**OrthogonalOperator**(*components*, *parameters=None*, *basis=None*, *exact=False*)

Bases: qubricks.operator.Operator

OrthogonalOperator is a subclass of Operator for representing Orthogonal matrices. The only difference is that the inversion process is simplified, using the result that the inverse of Q is simply its transpose.

Apart from the inversion operation, all other operations will result in a normal Operator object being returned.

inverse()

Returns An OrthogonalOperator object which is the inverse of this one.

This method computes the inverse of the Operator object. This may be very slow. If you do not need a symbolic inversion, then simply call the Operator object and take a numerical inverse using numpy.

4.3 StateOperator

class qubricks.stateoperator.**StateOperator**(*parameters=None*, *basis=None*, ***kwargs*)

Bases: object

StateOperator objects are not to be confused with Operator objects. StateOperator objects can encode arbitrary operations on states; including those which cannot be described by a linear map. Often, but not always, the

internal mechanisms of a StateOperator will *use* an Operator object. The StateOperator object itself is an abstract class, and must be subclassed before it is used. In order to conform to the expectations of the QuBricks framework, StateOperators should implement basis transformations and subspace restrictions.

Parameters

- **parameters** (*Parameters*) – A referenced to a Parameters instance, which can be shared among multiple objects. This is overridden when adding to a QuantumSystem instance, but is helpful for testing purposes.
- **kwargs** (*dict*) – Keyword arguments which are passed onto the StateOperator.init method, which must be implemented by subclasses.

Subclassing StateOperator:

A subclass of StateOperator must implement the following methods::

- init(self, **kwargs)
- __call__(self, state, t=0, params={})
- restrict(self, *indices)
- connected(self, *indices, **params)
- transform(self, transform_op)

And the following properties::

- for_state
- for_ensemble

And may, if desired, implement::

- collapse(self, *wrt, **params)

For more documentation about what these methods should return, see the documentation below.

Applying a StateOperator instance to a State: StateOperator objects are applied to states by calling them with the state as a passed parameter. For example:

```
>>> stateoperator(state)
```

It is also possible to pass a parameter context. Often, such as when performing numerical integration, you want to treat time specially, and so the time can also be passed on its own.

```
>>> stateoperator(state, t=0, params=param_dict)
```

Parameters may be supplied in any format that is supported by the Parameters module.

Note: The integration time is passed by the integrator via *t*, and not as part of the parameter context.

Operator(*operator*, *basis=None*, *exact=False*)

This method is a shorthand for constructing Operator objects which refer to the same Parameters instance as this StateOperator.

Parameters

- **components** (*Operator, dict, numpy.ndarray or sympy.MatrixBase*) – Specification for Operator.
- **basis** (*Basis or None*) – The basis in which the Operator is represented.

- **exact** (*bool*) – True if Operator should maintain exact representations of numbers, and False otherwise.

If *components* is already an Operator object, it is returned with its Parameters reference updated to point the Parameters instance associated with this StateOperator. Otherwise, a new Operator is constructed according to the specifications, again with a reference to this StateOperator's Parameters instance.

For more information, refer to the documentation for Operator.

basis
A reference to the current Basis of the Operator; or None if it has no specified Basis.

change_basis(*basis=None*, *threshold=False*)
Returns a copy of this StateOperator object expressed in the specified basis. The behaviour of the threshold parameter is described in the Basis.transform documentation; but this allows elements which differ from zero only by numerical noise to be set to zero.

Parameters

- **basis** (*Basis or None*) – The basis in which the new StateOperator should be represented.
- **threshold** (*bool or float*) – Whether a threshold should be used to limit the effects of numerical noise (if boolean), or the threshold to use (if float). See Basis.transform for more information.

collapse(**wrt*, ***params*)
This method is a stub to allow subclasses to simplify themselves when requested. If implemented, and Operators are used, the *collapse* method should be used on them also. See Operator.collapse for more information.

Note that unless this method is overridden, no simplification occurs.

connected(**indices*, ***params*)
This method should return the list of basis state indices which would mix with the specified basis state indices *indices* under repeated operation of the StateOperator. See Operator.connected for more information.

for_ensemble
Should be True if the StateOperator supports 2D ensemble operations; and False otherwise.

for_state
Should be True if the StateOperator supports 1D vector operations; and False otherwise.

init(***kwargs*)
This method is called when StateOperator subclasses are initialised, which allows subclasses to set themselves up as appropriate.

Parameters **kwargs** (*dict*) – The keyword arguments passed to the StateOperator constructor.

p
Returns a reference to the internal Parameter instance.

restrict(**indices*)
This method should return a new StateOperator restricted to the basis states with indices *indices*. See Operator.restrict for more information.

transform(*transform_op*)
This method should transform all future operations on arbitrary input states according to the transformation *transform_op*. See Operator.transform for more information.

4.4 Measurement(s) & MeasurementWrapper

class qubricks.measurement.**Measurement**(**args*, ***kwargs*)
Bases: object

A Measurement instance is an object which encodes the logic required to extract information from a QuantumSystem object. It specifies the algorithm to be performed to extract the measurement outcome, the type of the measurement results, and also provides methods to initialise and add results to storage when performing the same measurement iteratively. Measurement is an abstract class, and should be subclassed for each new class of measurements.

While Measurement objects can be used directly, they are typically used in conjunction with Measurements and MeasurementWrapper, as documented in those classes.

Any arguments and/or keyword arguments passed to the Measurement constructor are passed to Measurement.init.

Subclassing Measurement:

A subclass of Measurement must implement the following methods:

- init
- measure
- result_type
- result_shape

A subclass *may* also override the following methods to further customise behaviour:

- is_independent
- iterate_results_init
- iterate_results_add
- iterate_is_complete
- iterate_continue_mask

Documentation for these methods is provided below.

Applying Measurement instances: Although not normally used directly, you can use a Measurement instance directly on the results of an QuantumSystem integration, for example:

```
>>> measurement(data=system.integrate(...))
```

Calling a Measurement instance is an alias for the Measurement.measure method. If the measurement instance is configured to perform its own integration:

```
>>> measurement(times=..., psi_0s=..., ...)
```

Note that if the measurement instance needs access to the QuantumSystem instance, you can setup the reference using:

```
>>> measurement.system = system
```

init()
This method should initialise the Measurement instance in whatever way is necessary to prepare the instance for use. Note that any arguments passed to the Measurement constructor will also be passed to this method. There is no restriction on the method signature for the init method.

is_independent
True if this Measurement instance does all required integration internally (and so should not receive pre-computed integration data). *False* otherwise. The default implementation is *False*.

iterate_continue_mask(*results*)
This method returns a mask function (see *MeasurementWrapper* documentation), which in turn based on the *results* object (as initialised by *iterate_results_init*) returns True or False for a given set of indices indicating whether there already exists data for those indices.

Parameters **results** (*object*) – The results storage object (see Measurement.iterate_results_init).

iterate_is_complete(*results*)
This method returns *True* when the results object is completely specified (results have been added for all indices; and *False* otherwise.

Parameters **results** (*object*) – The results storage object (see Measurement.iterate_results_init).

iterate_results_add(*results=None*, *result=None*, *indices=None*, *params={}*)
This method adds a measurement result *result* from *Measurement.measure* to the *results* object initialised in *Measurement.iterate_results_init*. It should put this result into storage at the appropriate location for the provided *indices*.

Parameters

- **results** (*object*) – The storage object in which to place the result (as from *Measurement.iterate_results_init*).
- **result** (*object*) – The result to be stored (as from *Measurement.measure*).
- **indices** (*tuple*) – The indices at which to store this result.
- **params** (*dict*) – The parameter context for this measurement result.

iterate_results_init(*ranges=[]*, *shape=None*, *params={}*, **args*, ***kwargs*)
This method is called by *MeasurementWrapper.iterate_yielder* to initialise the storage of the measurement results returned by this object. By default, this method returns a numpy array with dtype as specified by *result_type* and shape returned by *result_shape*, with all entries set to np.nan objects. If necessary, you can overload this method to provide a different storage container This is a generic initialisation for the Measurement object. It can be overloaded if necessary.

Parameters

- **ranges** – The range specifications provided to MeasurementWrapper.iterate_yielder.
- **shape** (*tuple*) – The shape of the resulting evaluated ranges.
- **params** (*dict*) – The parameter context of the ranges.
- **args** (*tuple*) – Any additional arguments passed to MeasurementWrapper.iterate_yielder.
- **kwargs** (*dict*) – Any additional keyword arguments passed to MeasurementWrapper.iterate_yielder.

measure(*data=None*, *times=None*, *psi_0s=None*, *params={}*, ***kwargs*)
This method should return the value of a measurement as a numpy array with data type and shape as specified in *result_type* and *result_shape* respectively.

Note: It is possible to return types other than numpy array and still be compatible with iteration (see MeasurementWrapper) provided you overload the *iterate_results_init* and *iterate_results_add* methods.

Implementations of *measure* will typically be provided by integration data by a *MeasurementWrapper* instance (which will be a structured numpy array as returned by *Integrator.integrate) as the value for the 'data* keyword. A consistent set of values for *times* and *psi_0s* will also be passed.

Note: If an implementation of *measure* omits the *data* keyword, QuBricks assumes that all integration required by the *measure* operator will be performed internally. It can use the reference to a QuantumSystem instance at *Measurement.system* for this purpose. If the *data* keyword is present (for testing/etc), but pre-computed integration data is undesired, override the *is_independent* method to return *True*. If external data is *required*, then simply remove the default value of *data*.

Apart from the required keywords: *data*, *times*, *psi_0s* and *params*; any additional keywords can be specified. Refer to the documentation of *MeasurementWrapper* to see how their values will filter through.

Note: Although the keywords *times* and *psi_0s* are necessary, it is not necessary to use these keywords. As such, Measurement operators need not require an integration of the physical system.

Parameters

- **data** (*numpy.ndarray or None*) – Data from a QuantumSystem.integrate call, or None.
- **times** (*iterable*) – Sequence of times of interest.
- **psi_0s** (*str or iterable*) – The initial state vectors/ensembles with which to start integrating.
- **params** (*dict*) – Parameter context to use during this measurement. Parameter types can be anything supported by Parameters.
- **kwargs** (*dict*) – Any other keyword arguments not collected explicitly.

result_shape(**args*, ***kwargs*)

This method should return a tuple describing the shape of the numpy array to be returned by Measurement.measure.

This method will receive all arguments and keyword arguments passed to *iterate_results_init*, where it is used to initialise the storage of measurement results.

result_type(**args*, ***kwargs*)

This method should return an object suitable for use as the dtype argument in a numpy array constructor. Otherwise, no restrictions; other than that it must also agree with the data-type returned by *Measurement.measure*.

This method will receive all arguments and keyword arguments passed to *iterate_results_init*, where it is used to initialise the storage of measurement results.

system

A reference to a QuantumSystem instance. If a system instance is not provided, and an attempt to access this property is made, a RuntimeException is raised.

You can specify a QuantumSystem using:

```
>>> measurement.system = system
```

class qubricks.measurement.**MeasurementIterationResults**(*ranges*, *ranges_eval*, *results*, *runtime=None*, *path=None*, *samplers={}*)

Bases: object

MeasurementIterationResults is class designed to store the results of measurements applied iteratively over a range of different values (see *MeasurementWrapper.iterate_yielder*). Apart from its role as a data structure, it also provides methods for saving and loading the data to/from disk.

Parameters

- **ranges** (*dict or list of dict*) – The specification of ranges passed ultimately to the Parameters instance.
- **ranges_eval** (*numpy.ndarray*) – The values of the parameters after evaluation from the above specification.
- **results** (*dict*) – A dictionary of measurement results, with keys of measurement names.
- **runtime** (*float*) – An optional number indicating in seconds the time taken to generate the results.
- **path** (*str*) – The location to use as a storage location by default.
- **samplers** (*dict of callables*) – A dictionary of named samplers (see *parameters.Parameters.range*) for future use with *ranges*, since functions cannot be serialised.

Constructing a MeasurementIterationResults object: Manually constructing a MeasurementIterationResults instance is unusual, since this is handled for you by the MeasurementWrapper iteration methods. However, this is possible using:

```
>>> results = MeasurementIterationResults(ranges=..., ranges_eval=..., results=..., runtime=
```

Accessing results: To access the data stored in a MeasurementIterationResults instance, simply access the relevant attributes. The available attributes are:

- ranges
- ranges_eval
- results
- runtime
- path
- samplers

Each of these attributes corresponds to the documented parameters described above.

For example:

```
>>> mresults = results.results['measurement_name']
```

Note that all of these attributes can be freely overridden. Check out the *MeasurementIterationResults.update* method for an alternative to updating these results.

continue_mask (*measurements={}*)
This method provides a mask for the *parameters.iteration.RangesIterator* instance called in *MeasurementWrapper.iterate_yielder*. The provided mask calls the *Measurement.iterate_continue_mask* method with the appropriate results for each of the measurements provided in *measurements*.

Parameters **measurements** (*dict*) – A dictionary of measurement objects with keys indicating their names.

is_complete (*measurements={}*)
This method calls the *Measurement.iterate_is_complete* method with the appropriate results for each of the measurements provided. If False for any of these measurements, False is returned.

Parameters **measurements** (*dict*) – A dictionary of measurement objects with keys indicating their names.

classmethod load (*path*, *samplers={}*)
This method will load and populate a new MeasurementIterationResults object from previously saved data. If provided, *samplers* will be used to convert string names of samplers in the ranges to functions.

Parameters

- **path** (*str*) – A path to the file's destination. If not provided, the earlier provided path is used.
- **samplers** (*str*) – A dictionary of functions (or callables) indexed by string names.

For example:

```
>>> results = MeasurementIterationResults.load('data.dat')
```

save(*path=None*, *samplers=None*)

This method will save this MeasurementIterationResults object to disk at the specified path, trading the sampler functions in the ranges attribute with their names extract from samplers (if possible), or by using their inspected name (using the *__name__* attribute). The resulting file is a "shelf" object from the python *shelve* module.

Parameters

- **path** (*str*) – A path to the file's destination. If not provided, the earlier provided path is used.
- **samplers** (*str*) – A dictionary of functions (or callables) indexed by string names.

For example:

```
>>> results.save('data.dat')
```

update(***kwargs*)

This method allows you to update the stored data of this *MeasurementIterationResults* instance. Simply call this method with keyword arguments of the relevant attributes. For example:

```
>>> results.update(results=..., path=..., ...)
```

Note that you can update multiple attributes at once. The one special argument is "runtime", which will increment that attribute with the specified value, rather than replacing it. For example:

```
>>> results.runtime
231.211311
>>> results.update(runtime=2.2)
>>> results.runtime
233.411311
```

class qubricks.measurement.**MeasurementWrapper**(*system*, *measurements={}*)

Bases: object

The *MeasurementWrapper* class wraps around one or more *Measurement* objects to provide a consistent API for performing (potentially) multiple measurements at once. There are also performance benefits to be had, since wherever possible integration results are shared between the *Measurement* instances.

Parameters

- **system** (*QuantumSystem*) – A QuantumSystem instance.
- **measurements** – A dictionary of *Measurement* objects indexed by a string name.

Constructing MeasurementWrapper objects: The syntax for creating a *MeasurementWrapper* object is:

```
>>> wrapper = MeasurementWrapper(system, {'name': NamedMeasurement, ...})
```

where *NamedMeasurement* is a *Measurement* instance.

Adding Measurement objects: If you want to add additional *Measurement* objects after creating the *MeasurementWrapper* instance, use the *add_measurements* method. Refer to the documentation below for more information.

Performing Measurements: There are three basic procedures which you can use to perform measurements on the reference "system" *QuantumSystem* instance.

The first of these is *MeasurementWrapper.on*, which applies the measurement(s) to a pre-computed data. The second is *MeasurementWrapper.integrate*, which applies the measurement(s) to data computed on the fly. And the last is the iteration procedures: *MeasurementWrapper.iterate*, *MeasurementWrapper.iterate_yielder*, and *MeasurementWrapper.iterate_to_file*; each of which allows you to perform measurements over a range of parameter contexts.

Each of these methods is documented in more detail below.

add_measurements(***measurements*)

This method adds named measurements to the *MeasurementWrapper*. The syntax for this is:

```
>>> wrapper.add_measurements(name=NamedMeasurement, ...)
```

where "name" is any valid measurement name, and *NamedMeasurement* is a *Measurement* instance.

integrate(*times=None*, *psi_0s=None*, *params={}*, ***kwargs*)

This method performs an integration of the *QuantumSystem* referenced when this object was constructed, and then calls *Measurement.on* on that data. If all *Measurement* objects hosted are "independent" (have *Measurement.is_independent* as *True*), then no integration is performed.

Parameters

- **times** (*iterable*) – Times for which to report the state during integration.
- **psi_0s** (*list*) – Initial state vectors / ensembles for the integration. (See *QuantumSystem.state*.
- **params** – Parameter overrides to use during integration. (See *parameters.Parameters* documentation).
- **kwargs** (*dict*) – Additional keyword arguments to pass to *QuantumSystem.integrate* and *Measurement.measure*.

Note: Only keyword arguments prepended with '**int_**' are forwarded to

---

QuantumSystem.integrate, with the prefix removed. These keywords are not also passed to *Measurement.measure*.

For example:

```
>>> wrapper.integrate(times=['T'], psi_0s=['logical0'])
```

iterate(**args*, ***kwargs*)

This method is a wrapper around the *Measurement.iterate_yielder* method in the event that one only cares about the final result, and does not want to deal with interim results. This method simply waits until the iteration process is complete, and returns the last result.

All arguments and keyword arguments are passed to *MeasurementWrapper.iterate_yielder*.

iterate_to_file(*path*, *samplers={}*, *yield_every=60*, **args*, ***kwargs*)

This method wraps around *Measurement.iterate_yielder* in order to continue a previous measurement collection process (if it did not finish successfully) and to iteratively write the most recent results to a file. This method modifies the default *yield_every* of the *iterate_yielder* method to 60 seconds, so that file IO

is not the limiting factor of performance, and so that at most around a minute's worth of processing is lost in the event that something goes wrong.

Parameters

- **path** (*str*) – The path at which to save results. If this file exists, attempts are made to continue the measurement acquisition.
- **samplers** (*dict*) – A dictionary of samplers to be used when loading and saving the *MeasurementIterationResults* object. (see *MeasurementIterationResults.load* and *MeasurementIterationResults.save*)
- **yield_every** (*number or None*) – The minimum time between attempts to save the results. (see *iterate_yielder*)
- **args** (*tuple*) – Additional arguments to pass to *iterate_yielder*.
- **kwargs** (*dict*) – Additional keyword arguments to pass to *iterate_yielder*.

Measurement.iterate_to_file saves the results of the Measurement.iterate method to a python shelve file at *path*; all other arguments are passed through to the Measurement.iterate method.

iterate_yielder(*ranges*, *params={}*, *masks=None*, *nprocs=None*, *yield_every=0*, *results=None*, *progress=True*, ***kwargs*)

This method iterates over the possible Cartesian products of the parameter ranges provided, at each step running the *MeasurementWrapper.integrate* in the resulting parameter context. After every *yield_every* seconds, this method will flag that it needs to yield the results currently accumulated (as a *MeasurementIterationResults* object) when the next measurement result has finished computing. This means that you can, for example, progressively save (or plot) the results as they are taken. Note that if the processing of the results is slow, this can greatly increase the time it takes to finish the iteration.

Parameters

- **ranges** (*list or dict*) – A valid ranges specification (see *parameters.iteration.RangesIterator*)
- **params** (*dict*) – Parameter overrides to use (see *parameters.Parameters.range*)
- **masks** (*list of callables*) – List of masks to use to filter indices to compute. (see *parameters.iteration.RangesIterator*)
- **nprocs** (*number or None*) – Number of processes to spawn (if 0 or 1 multithreading is not enabled) (see *parameters.iteration.RangesIterator*)
- **yield_every** (*number or None*) – Minimum number of seconds to go without returning the next result. To yield the value after every successful computation, use yield_every=0 . If yield_every is None, results are returned only after every computation has succeeded. By default, yield_every = 0.
- **results** (*MeasurementIterationResults*) – Previously computed MeasurementIterationResults object to extend.
- **progress** (*bool or callable*) – Whether progress information should be shown (True or False); or a callable. (see *parameters.iteration.RangesIterator* for more)
- **kwargs** (*dict*) – Additional keyword arguments to be passed to *MeasurementWrapper.integrate* (and also to *Measurement.iterate_results_init*.

on(*data*, ***kwargs*)

This method applies the *Measurement.measure* method to *data* for every *Measurement* stored in this object. If there are two or more *Measurement* objects, this method returns a dictionary of *Measurement.measure* results; with keys being the measurement names. If there is only one *Measurement* object, the return value of *Measurement.measure* is returned.

Parameters

- **data** (*numpy.ndarray*) – Data from an *QuantumSystem* integration.
- **kwargs** (*dict*) – Additional kwargs to pass to *Measurement.measure*.

For example: >>> wrapper.on(data, mykey=myvalue)

Note that if *data* is not *None*, then *psi_0s* and *times* are extracted from *data*, and passed to *Measurement.measure* as well.

Also note that if a measurement has *Measurement.is_independent* being *True*, only the *psi_0s* and *times* will be forwarded from *data*.

class qubricks.measurement.**Measurements**(*system*)
Bases: object

Measurements is a designed to simplify the Measurement evaluation process by acting as a host for multiple named Measurement objects. This object is used as the *measure* attribute of *QuantumSystem* objects.

Parameters **system** (*QuantumSystem*) – A QuantumSystem instance.

Constructing a Measurements object: If you want to create a Measurements instance separate from the one hosted by *QuantumSystem* objects, use the following:

```
>>> measurements = Measurements(system)
```

Adding and removing Measurement objects: To add a Measurement object to a Measurements instance, you simply provide it a string name, and use (for example):

```
>>> measurements._add("name", NamedMeasurement)
```

where *NamedMeasurement* is a subclass of *Measurement*.

To remove a *Measurement* from *Measurements*, use:

```
>>> measurements._remove("name")
```

The underscores preceding these methods' names are designed to prevent name clashes with potential Measurement names.

When a *Measurement* instance is added to *Measurements*, its internal "system" attribute is updated to point to the *QuantumSystem* used by *Measurements*.

Extracting a Measurement object: Once added to the *Measurements* object, a *Measurement* object can be accessed using attribute notation, or by calling the *Measurements* instance. For example:

```
>>> system.measure.name
```

Or to bundle multiple measurements up into the same evaluation:

```
>>> system.measure("measurement_1", "measurement_2", ...)
```

In both cases, the return type is **not** a *Measurement* instance, but rather a *MeasurementWrapper* instance, which can be used to perform the *Measurement.measure* operations in a simplified and consistent manner. See the *MeasurementWrapper* documentation for more information.

Inspecting a Measurements instance: To see a list of the names of measurements stored in a *Measurements* instance, you can use:

```
>>> measurements._names
```

To get a reference of the dictionary internally used by *Measurements* to store and retrieve hosted *Measurement* objects, use:

```
>>> measurements._measurements
```

4.5 Basis

class qubricks.basis.**Basis**(*dim=None*, *parameters=None*, ***kwargs*)
Bases: object

A Basis instance describes a particular basis, and allows transformations of objects (such as 'Operator's) from one basis to another. A Basis is an abstract class, and must be subclassed to be useful.

Parameters

- **dim** (*int or None*) – The dimension of the basis. If not specified, the dimension will be extracted from the Operator returned by Basis.operator; except during *Basis.init*, where *Basis.dim* will return the raw value stored (e.g. None).
- **parameters** (*parameters.Parameters*) – A Parameters instance, if required.
- **kwargs** (*dict*) – Additional keyword arguments to pass to *Basis.init*.

Subclassing Basis: Subclasses of Basis must implement the following methods, which function according to their respective documentation below: - init - operator Subclasses may optionally implement: - state_info - state_toString - state_fromString - state_latex These latter methods are used to allow convenient conversion of strings to states and also later representation of states as strings/LaTeX. Otherwise, these methods are not required. Since they are not used except when the user desires to change the state's representation, the implementer has a lot of freedom about the way these functions work, and what they return. The documentation for these methods indicates the way in which the original author intended for them to function.

Operator(*components*, *basis=None*, *exact=False*)
This method is a shorthand for constructing Operator objects which refer to the same Parameters instance as this Basis instance.

Parameters

- **components** (*Operator, dict, numpy.ndarray or sympy.MatrixBase*) – Specification for Operator.
- **basis** (*Basis or None*) – The basis in which the Operator is represented.
- **exact** (*bool*) – True if Operator should maintain exact representations of numbers, and False otherwise.

If *components* is already an Operator object, it is returned with its Parameters reference updated to point the Parameters instance associated with this Basis instance. Otherwise, a new Operator is constructed according to the specifications, again with a reference to this Basis's Parameters instance.

For more information, refer to the documentation for Operator.

dim
The dimension of the basis; or equivalently, the number of basis states.

init(***kwargs*)
This method should do whatever is necessary to prepare the Basis instance for use. When this method is called by the Python __init__ method, you can use *Basis.dim* to access the raw value of *dim*. If *dim* is necessary to construct the operator, and it is not set, this method should raise an exception. All keyword arguments except *dim* and *parameters* passed to the *Basis* instance constructor will also be passed to this method.

operator
This method should return a two dimensional *Operator* object, with basis states as columns. The *Operator* object should use the *Parameters* instance provided by the Basis instance. The simplest way to ensure this is to use the *Basis.Operator* method.

p
A reference to the Parameters instance used by this object.

state_fromString(*string*, *params={}*)
This method (if implemented) should return the state as a numerical array that is represented as a string in *string*. Calling *basis.state_toString* should then return the same (or equivalent) string representation.

Parameters

- **string** – A string representation of a state.
- **params** (*dict*) – A dictionary of parameter overrides. (see *parameters.Parameters*)

state_fromSymbolic(*expr*)
This method converts a sympy representation of a quantum state into an array or vector (as used by QuBricks). It uses internally *Basis.state_fromString* to recognise ket and bra names, and to substitute them appropriately with the right state vectors.

> **Warning:** Support for conversion from symbolic representations is not fully baked, but seems to work reasonably well.

state_info(*state*, *params={}*)
This method (if implemented) should return a dictionary with more information about the state provided. There are no further constraints upon what might be returned.

Parameters

- **state** (*str or iterable*) – The state about which information should be returned.
- **params** (*dict*) – A dictionary of parameter overrides. (see *parameters.Parameters*)

state_latex(*state*, *params={}*)
This method (if implemented) should return string that when compiled by LaTeX would represent the state.

Parameters

- **state** (*iterable*) – The state which should be represented as a string.
- **params** (*dict*) – A dictionary of parameter overrides. (see *parameters.Parameters*)

state_toString(*state*, *params={}*)
This method (if implemented) should return a string representation of the provided state, which should then be able to be converted back into the same state using *Basis.state_fromString*.

Parameters

- **state** (*iterable*) – The state which should be represented as a string.
- **params** (*dict*) – A dictionary of parameter overrides. (see *parameters.Parameters*)

state_toSymbolic(*state*)
This method is a stub, and may be implemented in the future to provide the logical inverse of *Basis.state_fromSymbolic*.

states(***params*)
This method returns the basis states (columns of the *Operator* returned by *basis.operator*) as a list. The Operator is first evaluated with the parameter overrides in params.

Parameters **params** (*dict*) – A dictionary of parameter overrides. (see *parameters.Parameters*)

transform(*state*, *inverse=False*, *threshold=False*, *params={}*)

This method allows one to transform states from the standard basis to this basis; or, if the inverse flag is provided, to transform from this basis to the standard basis. This is chained in the Basis.transform_to and Basis.transform_from methods to convert states between bases. State objects can be Operator or numpy array objects; and can be one or two dimensional. The basis states are evaluated in the parameter context specified in *params* before being used in this method.

This method can automatically try to set elements in the transformed object that are different from zero by some small amount to zero, in the hope of ignoring numerical error. If threshold is *False*, no attempts to clean the transformed state are made. If a numerical threshold is provided, any elements of the resulting transformed state with amplitude less than the supplied value will be set to zero. If threshold is set to True, the transformation operation attempts to determine the threshold automatically. This automatic algorithm looks for the smallest entry in *Basis.operator* and then multiplies it by 10**-8. This value is then used as the threshold. One should use this feature with caution.

Parameters

- **state** (*1D or 2D Operator or numpy.ndarray*) – The state to be transformed.
- **inverse** (*bool*) – *True* for transformation from this basis to the standard basis, and *False* for transformation to this basis from the standard basis.
- **threshold** (*bool or float*) – True or False to specify that the threshold should be automatically determined or not used respectively. If a float is provided, that value is used as the threshold.
- **params** (*dict*) – The parameter overrides to use during the transformation (see *Operator*).

transform_from(*state*, *basis=None*, *threshold=False*, *params={}*)

This method transforms the given state to this basis from the basis provided in *basis* (which must be a Basis instance). If *basis* is note provided, the standard basis is assumed.

Parameters

- **state** (*1D or 2D Operator or numpy.ndarray*) – The state to be transformed.
- **basis** (*Basis or None*) – The basis into which the state should be transformed.
- **threshold** (*bool or float*) – True or False to specify that the threshold should be automatically determined or not used respectively. If a float is provided, that value is used as the threshold.
- **params** (*dict*) – The parameter overrides to use during the transformation (see *Operator*).

transform_op(*basis=None*, *inverse=False*, *threshold=False*, *params={}*)

This method returns a function which can be used to transform any 1D or 2D *Operator* or numpy array to (from) this basis from (to) the basis provided in *basis*, if *inverse* is False (True). If basis is not provided, the standard basis is assumed.

Parameters

- **state** (*1D or 2D Operator or numpy.ndarray*) – The state to be transformed.
- **basis** (*Basis or None*) – The basis into which the state should be transformed.
- **inverse** (*bool*) – *True* for transformation from this basis to the *basis* provided, and *False* for transformation to this basis from the the *basis* provided.
- **threshold** (*bool or float*) – True or False to specify that the threshold should be automatically determined or not used respectively. If a float is provided, that value is used as the threshold.
- **params** (*dict*) – The parameter overrides to use during the transformation (see *Operator*).

For example:

```
>>> f = Basis.transform_op()
>>> state_transformed = f(state)
```

transform_to(*state*, *basis=None*, *threshold=False*, *params={}*)

This method transforms the given state from this basis to the basis provided in *basis* (which must be a Basis instance). If *basis* is note provided, the standard basis is assumed.

Parameters

- **state** (*1D or 2D Operator or numpy.ndarray*) – The state to be transformed.
- **basis** (*Basis or None*) – The basis into which the state should be transformed.
- **threshold** (*bool or float*) – True or False to specify that the threshold should be automatically determined or not used respectively. If a float is provided, that value is used as the threshold.
- **params** (*dict*) – The parameter overrides to use during the transformation (see *Operator*).

class qubricks.basis.**QubricksBasis**

Bases: sympy.physics.quantum.operator.Operator

This object is used internally to support symbolic representations of states.

default_assumptions = {'nonzero': True, 'prime': False, 'commutative': False, 'nonpositive': False, 'composite': Fa

is_algebraic = False

is_commutative = False

is_complex = False

is_composite = False

is_even = False

is_imaginary = False

is_integer = False

is_irrational = False

is_negative = False

is_noninteger = False

is_nonnegative = False

is_nonpositive = False

is_nonzero = True

is_odd = False

is_positive = False

is_prime = False

is_rational = False

is_real = False

is_transcendental = False

is_zero = False

class qubricks.basis.**QubricksBra**
Bases: sympy.physics.quantum.state.Bra

This object is used to represent states analytically.

For example:

```
>>> bra = QubricksBra('0')
```

These objects then obey standard arithmetic, for example:

```
>>> 2*bra
2<0|
```

You can convert from a symbolic representation of states to a QuBricks array using *Basis.state_fromSymbolic*.

default_assumptions = {'nonzero': True, 'prime': False, 'commutative': False, 'nonpositive': False, 'composite': Fa

classmethod dual_class()

is_algebraic = False

is_commutative = False

is_complex = False

is_composite = False

is_even = False

is_imaginary = False

is_integer = False

is_irrational = False

is_negative = False

is_noninteger = False

is_nonnegative = False

is_nonpositive = False

is_nonzero = True

is_odd = False

is_positive = False

is_prime = False

is_rational = False

is_real = False

is_transcendental = False

is_zero = False

class qubricks.basis.**QubricksKet**
Bases: sympy.physics.quantum.state.Ket

This object is used to represent states analytically.

For example:

```
>>> ket = QubricksKet('0')
```

These objects then obey standard arithmetic, for example:

```
>>> 2*ket
2|0>
```

You can convert from a symbolic representation of states to a QuBricks array using *Basis.state_fromSymbolic*.

default_assumptions = {'nonzero': True, 'prime': False, 'commutative': False, 'nonpositive': False, 'composite': Fa

classmethod dual_class()

is_algebraic = False

is_commutative = False

is_complex = False

is_composite = False

is_even = False

is_imaginary = False

is_integer = False

is_irrational = False

is_negative = False

is_noninteger = False

is_nonnegative = False

is_nonpositive = False

is_nonzero = True

is_odd = False

is_positive = False

is_prime = False

is_rational = False

is_real = False

is_transcendental = False

is_zero = False

strlabel

4.6 Integrator

class qubricks.integrator.**Integrator**(*identifier=None*, *initial=None*, *t_offset=0*, *operators=None*, *parameters=None*, *params={}*, *error_rel=1e-08*, *error_abs=1e-08*, *time_ops={}*, *progress=False*, ***kwargs*)

Bases: object

Integrator instances perform a numerical integration on arbitrary initial states using *StateOperator* objects to describe the instantaneous derivative. It is itself an abstract class, which must be subclassed. This allows the separation of logic from actual integration machinery.

Parameters

- **identifier** (*object*) – An object to identify this integrator from others. Can be left unspecified.
- **initial** (*list/tuple of numpy.arrays*) – A sequence of states/ensembles to use as initial states in the integration.
- **t_offset** (*object*) – Normally integration starts from t=0. Use this to specify a time offset. Can be any value understood by a *Parameters* instance.
- **parameters** (*Parameters or None*) – A *Parameters* instance for *Integrator* to use.
- **params** (*dict*) – Parameter overrides to use during when evaluating *StateOperator* objects.
- **error_rel** (*float*) – The maximum relative error allowable.
- **error_abs** (*float*) – The maximum absolute error allowable.
- **time_ops** (*dict*) – A dictionary of *StateOperator* objects to be applied at the time indicated by their index (which can be any object understood by *Parameters*).
- **progress** (*bool or IntegratorCallback*) – *True* if progress should be shown using the fallback callback, *False* if not, or and *IntegratorCallback* instance. This is used to report integrator progress.
- **kwargs** – Additional keyword arguments to pass to the *_integrator* method.

Subclassing Integrator:

Subclasses of *Integrator* must implement the following methods:

- _integrator(self, f, **kwargs)
- _integrate(self, integrator, initial, times=None, **kwargs)
- _derivative(self, t, y, dim)
- _state_internal2ode(self, state)
- _state_ode2internal(self, state, dimensions)

The documentation for these methods is available using:

```
>>> help(Integrator.<name of function>)
```

Their documentation will not appear in a complete API listing because they are private methods.

add_operator (*operator*)
This method appends the provided *StateOperator* to the list of operators to be used during integrations.

Parameters **operator** (*StateOperator*) – The operator to add to the list of operators contributing to the instantaneous derivative.

add_time_op (*time*, *time_op*)
This method adds a time operator *time_op* at time *time*. Note that there can only be one time operator for any given time. A "time operator" is just a *StateOperator* that will be applied at a particular time. Useful in constructing ideal pulse sequences.

Parameters

- **time** (*object*) – Time can be either a float or object interpretable by a *Parameters* instance.
- **time_op** (*StateOperator*) – The *StateOperator* instance to be applied at time *time*.

error_abs
The maximum absolute error permitted in the integrator. This can be set using:

```
>>> integrator.error_abs = <float>
```

error_rel
The maximum relative error permitted in the integrator. This can be set using:

```
>>> integrator.error_rel = <float>
```

extend(*times=None*, ***kwargs*)
This method extends an integration with returned states for the times of interest specified. This method requires that *Integrator.start* has already been called at least once, and that at least some of the times in *times* are after the latest times already integrated. Any previous times are ignored. Any additional keyword arguments are passed to the *_integrate* method. See the documentation for your *Integrator* instance for more information.

Parameters

- **times** (*iterable*) – A sequence of times, which can be any objected understood by *Parameters*. Should all be larger than the last time of the previous results.
- **kwargs** (*dict*) – Additional keyword arguments to send to the integrator.

get_operators(*indices=None*)
This method returns the operators of *Integrator.operators* restricted to the indicies specified (using *StateOperator.restrict*).

This is used internally by *Integrator* to optimise the integration process (by restricting integration to the indices which could possibly affect the state).

Parameters **indicies** (*interable of int*) – A sequence of basis indices.

get_progress_callback()
This method returns the *IntegratorCallback* object that will be used by *Integrator*. Note that if a callback has not been specified, and *Integrator.progress_callback* is *False*, then an impotent *IntegratorCallback* object is returned, which has methods that do nothing when called.

get_time_ops(*indices=None*)
This method returns the "time operators" of *Integrator.time_ops* restricted to the indicies specified (using *StateOperator.restrict*).

This is used internally by *Integrator* to optimise the integration process (by restricting integration to the indices which could possibly affect the state).

Parameters **indicies** (*interable of int*) – A sequence of basis indices.

initial
The initial state. Ignored in *Integrator.extend*. Can be set using:

```
>>> integrator.initial = <list of states>
```

int_kwargs
A reference to the dictionary of extra keyword arguments to pass to the *_integrator* initialisation method; which in turn can use these keyword arguments to initialise the integration.

operators
A reference to the list of operators (each of which is a *StateOperator*) used internally. To add operators you can directly add to this list, or use (much safer):

```
>>> integrator.operators = [<StateOperator>, <StateOperator>, ...]
```

Or, alternatively:

```
>>> integrator.add_operator( <StateOperator> )
```

p
A reference to the *Parameters* instance used by this object. This reference can be updated using:

```
>>> integrator.p = <Parameters instance>
```

params
A reference to the parameter overrides to be used by the *StateOperator* objects used by *Integrator*. The parameter overrides can be set using:

```
>>> integrator.params = { .... }
```

See *parameters.Parameters* for more information about parameter overrides.

progress_callback
The currently set progress callback. This can be *True*, in which case the default fallback callback is used; *False*, in which case the callback is disabled; or a manually created instance of *IntegratorCallback*. To retrieve the *IntegratorCallback* that will be used (including the fallback), use *Integrator.get_progress_callback*.

The progress callback instance can be set using:

```
>>> integrator.progress_callback = <True, False, or IntegratorCallback instance>
```

reset()
This method resets the internal stored *Integrator.results* to *None*, effectively resetting the *Integrator* object to its pre-integration status.

```
>>> integrator.reset()
```

results
The currently stored results. Used to continue integration in *Integrator.extend*. While it is possible to overwrite the results, the new value is not checked, and care should be taken.

```
>>> integrator.results = <valid results object>
```

start(*times=None*, ***kwargs*)
This method starts an integration with returned states for the times of interest specified. Any additional keyword arguments are passed to the *_integrate* method. See the documentation for your *Integrator* instance for more information.

Parameters

- **times** (*iterable*) – A sequence of times, which can be any objected understood by *Parameters*.
- **kwargs** (*dict*) – Additional keyword arguments to send to the integrator.

t_offset
The initial time to use in the integration. Ignored in *Integrator.extend*. Can be set using:

```
>>> integrator.t_offset = <time>
```

time_ops
A reference to the dictionary of time operators. Can be updated directly by adding to the dictionary, or (much more safely) using:

```
>>> self.time_ops = {'T': <StateOperator>, ...}
```

The above is shorthand for:

```
        >>> for time, time_op in {'T': <StateOperator>, ...}:
                self.add_time_op(time, time_op)
```

class qubricks.integrator.**IntegratorCallback**
Bases: object

onComplete(*identifier=None*, *message=None*, *status=0*)

onProgress(*progress*, *identifier=None*)

onStart()

class qubricks.integrator.**Progress**
Bases: object

class qubricks.integrator.**ProgressBarCallback**
Bases: qubricks.integrator.IntegratorCallback

onComplete(*identifier=None*, *message=None*, *status=0*)

onProgress(*progress*, *identifier=None*)

onStart()

CHAPTER

FIVE

API: WALL CLASSES

5.1 QuantumSystem Implementations

class qubricks.wall.systems.**SimpleQuantumSystem**(*parameters=None*, ***kwargs*)
Bases: qubricks.system.QuantumSystem

SimpleQuantumSystem is a subclass of *QuantumSystem* that enables you to initialise a *QuantumSystem* instance in one line, by passing keyword arguments to the constructor. Otherwise, it is indistinguishable.

Parameters

- **hamiltonian** (*Operator or numpy.array or list*) – The Hamiltonian to use for this QuantumSystem. Can be an Operator or an array.
- **bases** (*dict of Basis*) – A dictionary of bases to add to the QuantumSystem.
- **states** (*dict of arrays*) – A dictionary of states to add to the QuantumSystem.
- **measurements** (*dict of Measurement*) – A dictionary of 'Measurement's to add to the QuantumSystem.
- **derivative_ops** (*dict of StateOperator*) – A dictionary of 'StateOperator's to add to the QuantumSystem.

For more documentation, see *QuantumSystem*.

init(*hamiltonian=None*, *bases=None*, *states=None*, *measurements=None*, *derivative_ops=None*)
Configure any custom properties/attributes using kwargs passed to __init__.

init_bases()

init_derivative_ops(*components=None*)
Setup the derivative operators to be implemented on top of the basic quantum evolution operator.

init_hamiltonian()

init_measurements()

init_states()

5.2 Operator and OperatorSet Implementations

class qubricks.wall.operators.**PauliOperatorSet**(*components=None*, *defaults=None*, ***kwargs*)
Bases: qubricks.operator.OperatorSet

This subclass of *OperatorSet* does not store any components directly, but generates tensor products of Pauli I,X,Y and Z upon request. For example:

```
>>> p = PauliOperatorSet()
>>> p['X']()
array([[ 0.+0.j,  1.+0.j],
       [ 1.+0.j,  0.+0.j]])
>>> p['XX']()
array([[ 0.+0.j,  0.+0.j,  0.+0.j,  1.+0.j],
       [ 0.+0.j,  0.+0.j,  1.+0.j,  0.+0.j],
       [ 0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j],
       [ 1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j]])
```

As with any *OperatorSet*, the return type is an *Operator* instance.

Note: You can still use the aggregation and *apply* methods, such as

```
>>> p('XX', 'YY', 'ZZ')
```

will sum p['XX'], p['YY'] and p['ZZ'] together.

---

init()

5.3 StateOperator Implementations

class qubricks.wall.stateoperators.**DummyStateOperator**(*parameters=None*, *basis=None*, ***kwargs*)

Bases: qubricks.stateoperator.StateOperator

An StateOperator instance that does nothing to the state, but which forces the integrator to work as if it were necessary to evolve ensemble states.

collapse(**wrt*, ***params*)

connected(**indices*, ***params*)

for_ensemble

for_state

init(***kwargs*)

restrict(**indices*)

transform(*transform_op*)

class qubricks.wall.stateoperators.**LindbladStateOperator**(*parameters=None*, *basis=None*, ***kwargs*)

Bases: qubricks.stateoperator.StateOperator

A StateOperator instance that effects a single-termed Lindblad master equation. This will cause decay in a simple two level system proportional to: exp(-8*coefficient*t)

collapse(**wrt*, ***params*)

connected(**indices*, ***params*)

for_ensemble

for_state

init(*coefficient*, *operator*)

optimise_coefficient()

restrict(**indices*)

transform(*transform_op*)

class qubricks.wall.stateoperators.**SchrodingerStateOperator**(*parameters=None*, *basis=None*, ***kwargs*)

Bases: qubricks.stateoperator.StateOperator

A StateOperator instance that effects Schroedinger evolution of the (quantum) state.

collapse(**wrt*, ***params*)

connected(**indices*, ***params*)

for_ensemble

for_state

init(*H*)

restrict(**indices*)

transform(*transform_op*)

5.4 Measurement Implementations

class qubricks.wall.measurements.**AmplitudeMeasurement**(**args*, ***kwargs*)

Bases: qubricks.measurement.Measurement

Amplitude is a sample Measurement subclass that measures the amplitude of being in certain basis states as function of time throughout some state evolution. The basis states of interest should be identified using the *subspace* keyword to the measure function.

amplitudes(*state*)

init()

measure(*data*, *times*, *psi_0s*, *params={}*, *subspace=None*, ***kwargs*)

result_shape(**args*, ***kwargs*)

result_type(**args*, ***kwargs*)

class qubricks.wall.measurements.**ExpectationMeasurement**(**args*, ***kwargs*)

Bases: qubricks.measurement.Measurement

ExpectationMeasurement measures the expectation values of a particular set of operators applied to a system state. It can be initialised using:

```
>>> ExpectationMeasurement(<Operator 1>, <Operator 2>, ...)
```

expectations(*state*)

init(**ops*)

measure(*data*, *params={}*, *subspace=None*, *psi_0s=None*, *times=None*, ***kwargs*)

result_shape(**args*, ***kwargs*)

result_type(**args*, ***kwargs*)

class qubricks.wall.measurements.**LeakageMeasurement**(**args*, ***kwargs*)
Bases: qubricks.measurement.Measurement

Leakage measures the probability of a quantum system being outside of a specified subspace. The subspace of interest should be identified using the *subspace* keyword to the measure function.

init()

leakage(*state*, *subspace*, *output*, *params*)

measure(*data*, *times*, *psi_0s*, *params={}*, *subspace=None*, *input=None*, *output=None*, ***kwargs*)

result_shape(**args*, ***kwargs*)

result_type(**args*, ***kwargs*)

result_units

5.5 Basis Implementations

class qubricks.wall.bases.**SimpleBasis**(*dim=None*, *parameters=None*, ***kwargs*)
Bases: qubricks.basis.Basis

SimpleBasis is a subclass of *Basis* that allows a Basis object to be created on the fly from an *Operator*, a numpy array or a list instance. For example:

```
>>> SimpleBasis(parameters=<Parameters intance>, operator=<Operator, numpy.ndarray or list insta
```

init(*operator=None*)

operator

class qubricks.wall.bases.**SpinBasis**(*dim=None*, *parameters=None*, ***kwargs*)
Bases: qubricks.wall.bases.StandardBasis

SpinBasis is a subclass of *StandardBasis* that associates each element of the standard basis with a spin configuration. It assumes that there are n spin-1/2 particles in the system, and thus requires the dimension to be 2^n. It also implements conversion to and from string representation of the states.

init()

operator

state_fromString(*state*, *params={}*)
Convert strings representing sums of basis states of form: "<complex coefficient>|<state label>>" into a numerical vector.

e.g. "|uuu>" -> [1,0,0,0,0,0,0,0] "0.5|uuu>+0.5|ddd>" -> [0.5,0,0,0,0,0,0,0.5] etc.

state_info(*state*, *params={}*)
Return a dictionary with the total z-spin projection of the state.

e.g. |uud> -> {'spin': 0.5}

state_latex(*state*, *params={}*)
Returns the latex representation of each of the basis states. Note that this method does not deal with arbitrary states, as does SpinBasis.state_toString and SpinBasis.state_fromString .

state_toString(*state*, *params={}*)
Applies the inverse map of SpinBasis.state_fromString .

class qubricks.wall.bases.**StandardBasis**(*dim=None*, *parameters=None*, ***kwargs*)

Bases: qubricks.basis.Basis

StandardBasis is a simple subclass of *Basis* that describes the standard basis; that is, presents a basis that looks like the identity operator. An instance can be created using:

```
>>> StandardBasis(parameters=<Parameters instance>, dim=<dimension of basis>)
```

init()

operator

CHAPTER

SIX

API: ANALYSIS TOOLS

6.1 Perturbative Analysis

class qubricks.analysis.perturbation.**Perturb**(*H_0=None*, *V=None*, *subspace=None*)
Bases: object

Perturb is a class that allows one to perform degenerate perturbation theory. The perturbation theory logic is intentionally separated into a different class for clarity. Currently it only supports using *RSPT* for perturbation theory, though in the future this may be extended to *Kato* perturbation theory. The advantage of using this class as compared to directly using the *RSPT* class is that the energies and eigenstates can be computed cumulatively, as well as gaining access to shorthand constructions of effective Hamiltonians.

Parameters

- **H_0** (*Operator, sympy matrix or numpy array*) – The unperturbed Hamiltonian to consider.
- **V** (*Operator, sympy matrix or numpy array*) – The Hamiltonian perturbation to consider.
- **subspace** (*list of int*) – The state indices to which attention should be restricted.

E(*index*, *order=0*, *cumulative=True*)
This method returns the *index* th eigenvalue correct to order *order* if *cumulative* is *True*; or the the *order* th correction otherwise.

Parameters

- **index** (*int*) – The index of the state to be considered.
- **order** (*int*) – The order of perturbation theory to apply.
- **cumulative** (*bool*) – *True* if all order corrections up to *order* should be summed (including the initial unperturbed energy).

Es(*order=0*, *cumulative=True*, *subspace=None*)
This method returns a the energies associated with the indices in *subspaces*. Internally this uses *Perturb.E*, passing through the keyword arguments *order* and *cumulative* for each index in subspace.

Parameters

- **order** (*int*) – The order of perturbation theory to apply.
- **cumulative** (*bool*) – *True* if all order corrections up to *order* should be summed (including the initial unperturbed energy).
- **subspace** (*list of int*) – The set of indices for which to return the associated energies.

H_eff(*order=0*, *cumulative=True*, *subspace=None*, *adiabatic=False*)
This method returns the effective Hamiltonian on the subspace indicated, using energies and eigenstates computed using *Perturb.E* and *Perturb.Psi*. If *adiabatic* is *True*, the effective Hamiltonian describing

the energies of the instantaneous eigenstates is returned in the basis of the instantaneous eigenstates (i.e. the Hamiltonian is diagonal with energies corresponding to the instantaneous energies). Otherwise, the Hamiltonian returned is the sum over the indices of the subspace of the perturbed energies multiplied by the outer product of the corresponding perturbed eigenstates.

Parameters

- **order** (*int*) – The order of perturbation theory to apply.
- **cumulative** (*bool*) – *True* if all order corrections up to *order* should be summed (including the initial unperturbed energies and states).
- **subspace** (*list of int*) – The set of indices for which to return the associated energies.
- **adiabatic** (*bool*) – *True* if the adiabatic effective Hamiltonian (as described above) should be returned. *False* otherwise.

Psi(*index*, *order=0*, *cumulative=True*)

This method returns the *index* th eigenstate correct to order *order* if *cumulative* is *True*; or the the *order* th correction otherwise.

Parameters

- **index** (*int*) – The index of the state to be considered.
- **order** (*int*) – The order of perturbation theory to apply.
- **cumulative** (*bool*) – *True* if all order corrections up to *order* should be summed (including the initial unperturbed state).

Psis(*order=0*, *cumulative=True*, *subspace=None*)

This method returns a the eigenstates associated with the indices in *subspaces*. Internally this uses *Perturb.Psi*, passing through the keyword arguments *order* and *cumulative* for each index in subspace.

Parameters

- **order** (*int*) – The order of perturbation theory to apply.
- **cumulative** (*bool*) – *True* if all order corrections up to *order* should be summed (including the initial unperturbed state).
- **subspace** (*list of int*) – The set of indices for which to return the associated energies.

dim

The dimension of H_0.

pt

A reference to the perturbation calculating object (e.g. RSPT).

class qubricks.analysis.perturbation.**RSPT**(*H_0=None*, *V=None*, *subspace=None*)

Bases: object

This class implements (degenerate) Rayleigh-Schroedinger Perturbation Theory. It is geared toward generating symbolic solutions, in the hope that the perturbation theory might provide insight into the quantum system at hand. For numerical solutions, you are better off simply diagonalising the evaluated Hamiltonian.

> **Warning:** This method currently only supports diagonal H_0.

Parameters

- **H_0** (*Operator, sympy matrix or numpy array*) – The unperturbed Hamiltonian to consider.
- **V** (*Operator, sympy matrix or numpy array*) – The Hamiltonian perturbation to consider.
- **subspace** (*list of int*) – The state indices to which attention should be restricted.

E (*index*, *order=0*)
This method returns the *order* th correction to the eigenvalue associated with the *index* th state using RSPT.

The algorithm: If *order* is 0, return the unperturbed energy.

If *order* is even:

$$E_n = \left\langle \Psi_{n/2} \right| V \left| \Psi_{n/2-1} \right\rangle - \sum_{k=1}^{n/2} \sum_{l=1}^{n/2-1} E_{n-k-l} \left\langle \Psi_k \middle| \Psi_l \right\rangle$$

If *order* is odd:

$$E_n = \left\langle \Psi_{(n-1)/2} \right| V \left| \Psi_{(n-1)/2} \right\rangle - \sum_{k=1}^{(n-1)/2} \sum_{l=1}^{(n-1)/2} E_{n-k-l} \left\langle \Psi_k \middle| \Psi_l \right\rangle$$

Where subscripts indicate that the subscripted symbol is correct to the indicated order in RSPT, and where *n* = *order*.

Parameters

- **index** (*int*) – The index of the state to be considered.
- **order** (*int*) – The order of perturbation theory to apply.

H_0

Psi (*index*, *order=0*)
This method returns the *order* th correction to the *index* th eigenstate using RSPT.

The algorithm: If *order* is 0, return the unperturbed eigenstate.

Otherwise, return:

$$|\Psi_n\rangle = (E_0 - H_0)^{-1} P \left(V |\Psi_{n-1}\rangle - \sum_{k=1}^{n} E_k |\Psi_{n-k}\rangle \right)$$

Where *P* is the projector off the degenerate subspace enveloping the indexed state.

Parameters

- **index** (*int*) – The index of the state to be considered.
- **order** (*int*) – The order of perturbation theory to apply.

V

dim
The dimension of H_0.

get_unperturbed_states ()
This method returns the unperturbed eigenvalues and eigenstates as a tuple of energies and state-vectors.

Note: This is the only method that does not support a non-diagonal H_0. While possible to implement, it is not currently clear that a non-diagonal H_0 is actually terribly useful.

---

inv (*index*)
This method returns: $(E_0 - H_0)^{-1} P$, for use in *Psi*, which is computed using:

$$A_{ij} = \delta_{ij} \delta_{i0} (E_0^n - E_0^i)^{-1}$$

Where *n* = *order*.

Note: In cases where a singularity would result, *0* is used instead. This works because the projector off the subspace *P* reduces support on the singularities to zero.

Parameters **index** (*int*) – The index of the state to be considered.

qubricks.analysis.perturbation.**debug**(**messages*)

6.2 Spectral Analysis

qubricks.analysis.spectrum.**energy_spectrum**(*system*, *states*, *ranges*, *input=None*, *output=None*, *hamiltonian=None*, *components=[]*, *params={}*, *hamiltonian_init=None*, *components_init=None*, *params_init=None*, *complete=False*)

This function returns a list of sequence which are the energy eigenvalues of the states which map adiabatically to those provided in *states*. Consequently, the provided states should be eigenstates of the Hamiltonian (determined by *components_init* or *hamiltonian_init*) when the parameters are set according to *params_init*. Where the initial conditions are not set, the states are assumed to be eigenstates of the Hamiltonian provided for analysis (*hamiltonian* or *components*) in the corresponding parameter context *params*.

Parameters

- **system** (*QuantumSystem*) – A QuantumSystem instance.
- **states** (*list of object*) – A list of states (assumed to be eigenstates as noted above) for which we are interested in examining the eigen-spectrum.
- **ranges** (*dict*) – A range specification for iteration (see *Parameters.range*).
- **input** (*str, Basis or None*) – The basis of the specified states.
- **output** (*str, Basis or None*) – The basis in which to perform the calculations.
- **hamiltonian** (*Operator or None*) – The Hamiltonian for which a spectrum is desired.
- **components** (*list of str*) – If *hamiltonian* is *None*, the components to use from the provided *QuantumSystem* (see *QuantumSystem.H*).
- **params** (*dict*) – The parameter context in which to perform calculations.
- **hamiltonian_init** (*Operator*) – The Hamiltonian for which provided states are eigenstates. If not provided, and *components_init* is also not provided, this defaults to the value of *hamiltonian*.
- **components_init** – The components to enable such that the provided states are eigenstates. If not provided, this defaults to the value of *components*. (see *QuantumSystem.H*)
- **params_init** (*dict*) – The parameter context to be used such that the provided states are eigenstates of the initial Hamiltonian. If not provided, defaults to the value of *params*.
- **complete** (*bool*) – If *True*, then the eigen-spectrum of the remaining states not specifically requested are appended to the returned results.

Warning: Since this method uses the ordering of the eigenvalues to detect which eigenvalues belong to which eigenstates, this method does not work in cases when the adiabatic theorem is violated (i.e. when energy levels cross).

6.3 Utilities

class qubricks.analysis.model.**ModelAnalysis**(**args*, ***kwargs*)

Bases: object

ModelAnalysis is a helper class that can simplify the routine of running simulations, processing the results, and then plotting (or otherwise outputting) them. One simply need subclass *ModelAnalysis*, and implement the following methods:

- prepare(self, *args, **kwargs)
- simulate(self, *args, **kwargs)
- process(self, *args, **kwargs)
- plot(self, *args, **kwargs)

Each of these methods is guaranteed to be called in the order specified above, with the return values of the previous method being fed forward to the next. Calling *process* (with no arguments), for example, will also call *prepare* and *simulate* in order, with the return values of *prepare* being passed to *simulate*, and the return values of *simulate* being passed to *process*. If a method is called directly with input values, then this chaining does not occur, and the method simply returns what it should.

It is necessary to be a little bit careful about what one returns in these methods. In particular, this is the way in which return values are processed:

- If a tuple is returned of length 2, and the first element is a

tuple and the second a dict, then it is assumed that these are respectively the *args* and *kwargs* to be fed forward. - If a tuple is returned of any other length, or any of the above conditions fail, then these are assumed to be the *args* to be fed forward. - If a dictionary is returned, then these are assumed to be the *kwargs* to be fed forward. - Otherwise, the result is fed forward as the first non-keyword argument.

Note: It is not necessary to return values at these steps, if it is unnecessary or if you prefer to save your results as attributes.

Note: Return values of all of these methods will be cached, so each method will only be run once.

---

plot(**args*, ***kwargs*)

This method should perform whatever plotting/output is desired based upon return values of *process*.

prepare(**args*, ***kwargs*)

This method should prepare the *ModelAnalysis* instance for calling the rest of the methods. It is invoked on class initialisation, with the arguments passed to the constructor. Any return values will be passed onto *simulate* if it is ever called with no arguments.

process(**args*, ***kwargs*)

This method should perform whatever processing is interesting on return values of *simulate*. Any values returned will be passed onto *plot* if it is ever called with no arguments.

simulate(**args*, ***kwargs*)

This method should perform whichever simulations are required. Any values returned will be passed onto *process* if it is ever called with no arguments.

q

qubricks.wall.bases, 58
qubricks.wall.measurements, 57
qubricks.wall.operators, 55
qubricks.wall.stateoperators, 56
qubricks.wall.systems, 55

A

add_basis() (qubricks.system.QuantumSystem method), 22
add_derivative_op() (qubricks.system.QuantumSystem method), 22
add_measurement() (qubricks.system.QuantumSystem method), 22
add_measurements() (qubricks.measurement.MeasurementWrapper method), 41
add_operator() (qubricks.integrator.Integrator method), 50
add_state() (qubricks.system.QuantumSystem method), 23
add_subspace() (qubricks.system.QuantumSystem method), 23
add_time_op() (qubricks.integrator.Integrator method), 50
AmplitudeMeasurement (class in qubricks.wall.measurements), 57
amplitudes() (qubricks.wall.measurements.AmplitudeMeasurement method), 57
apply() (qubricks.operator.Operator method), 30
apply() (qubricks.operator.OperatorSet method), 33

B

bases (qubricks.system.QuantumSystem attribute), 23
Basis (class in qubricks.basis), 44
basis (qubricks.operator.Operator attribute), 30
basis (qubricks.stateoperator.StateOperator attribute), 35
basis() (qubricks.system.QuantumSystem method), 23

C

change_basis() (qubricks.operator.Operator method), 30
change_basis() (qubricks.stateoperator.StateOperator method), 35
clean() (qubricks.operator.Operator method), 31
collapse() (qubricks.operator.Operator method), 31
collapse() (qubricks.stateoperator.StateOperator method), 35
collapse() (qubricks.wall.stateoperators.DummyStateOperator method), 56
collapse() (qubricks.wall.stateoperators.LindbladStateOperator method), 56
collapse() (qubricks.wall.stateoperators.SchrodingerStateOperator method), 57
connected() (qubricks.operator.Operator method), 31
connected() (qubricks.stateoperator.StateOperator method), 35
connected() (qubricks.wall.stateoperators.DummyStateOperator method), 56
connected() (qubricks.wall.stateoperators.LindbladStateOperator method), 56
connected() (qubricks.wall.stateoperators.SchrodingerStateOperator method), 57
continue_mask() (qubricks.measurement.MeasurementIterationResults method), 39

D

debug() (in module qubricks.analysis.perturbation), 64
default_assumptions (qubricks.basis.QubricksBasis attribute), 47
default_assumptions (qubricks.basis.QubricksBra attribute), 48
default_assumptions (qubricks.basis.QubricksKet attribute), 49
default_derivative_ops (qubricks.system.QuantumSystem attribute), 23
derivative_ops() (qubricks.system.QuantumSystem method), 23
dim (qubricks.analysis.perturbation.Perturb attribute), 62
dim (qubricks.analysis.perturbation.RSPT attribute), 63
dim (qubricks.basis.Basis attribute), 44
dim (qubricks.system.QuantumSystem attribute), 24
dual_class() (qubricks.basis.QubricksBra class method), 48
dual_class() (qubricks.basis.QubricksKet class method), 49
DummyStateOperator (class in qubricks.wall.stateoperators), 56

E

E() (qubricks.analysis.perturbation.Perturb method), 61
E() (qubricks.analysis.perturbation.RSPT method), 62

energy_spectrum() (in module qubricks.analysis.spectrum), 64
ensemble() (qubricks.system.QuantumSystem method), 24
ensembles (qubricks.system.QuantumSystem attribute), 24
error_abs (qubricks.integrator.Integrator attribute), 50
error_rel (qubricks.integrator.Integrator attribute), 51
Es() (qubricks.analysis.perturbation.Perturb method), 61
exact (qubricks.operator.Operator attribute), 31
ExpectationMeasurement (class in qubricks.wall.measurements), 57
expectations() (qubricks.wall.measurements.ExpectationMeasurement method), 57
extend() (qubricks.integrator.Integrator method), 51

F

for_ensemble (qubricks.stateoperator.StateOperator attribute), 35
for_ensemble (qubricks.wall.stateoperators.DummyStateOperator attribute), 56
for_ensemble (qubricks.wall.stateoperators.LindbladStateOperator attribute), 56
for_ensemble (qubricks.wall.stateoperators.SchrodingerStateOperator attribute), 57
for_state (qubricks.stateoperator.StateOperator attribute), 35
for_state (qubricks.wall.stateoperators.DummyStateOperator attribute), 56
for_state (qubricks.wall.stateoperators.LindbladStateOperator attribute), 56
for_state (qubricks.wall.stateoperators.SchrodingerStateOperator attribute), 57

G

get_derivative_ops() (qubricks.system.QuantumSystem method), 24
get_integrator() (qubricks.system.QuantumSystem method), 24
get_operators() (qubricks.integrator.Integrator method), 51
get_progress_callback() (qubricks.integrator.Integrator method), 51
get_time_ops() (qubricks.integrator.Integrator method), 51
get_unperturbed_states() (qubricks.analysis.perturbation.RSPT method), 63

H

H() (qubricks.system.QuantumSystem method), 22
H_0 (qubricks.analysis.perturbation.RSPT attribute), 63
H_eff() (qubricks.analysis.perturbation.Perturb method), 61
hamiltonian (qubricks.system.QuantumSystem attribute), 25

I

init() (qubricks.basis.Basis method), 44
init() (qubricks.measurement.Measurement method), 36
init() (qubricks.operator.OperatorSet method), 33
init() (qubricks.stateoperator.StateOperator method), 35
init() (qubricks.system.QuantumSystem method), 25
init() (qubricks.wall.bases.SimpleBasis method), 58
init() (qubricks.wall.bases.SpinBasis method), 58
init() (qubricks.wall.bases.StandardBasis method), 59
init() (qubricks.wall.measurements.AmplitudeMeasurement method), 57
init() (qubricks.wall.measurements.ExpectationMeasurement method), 57
init() (qubricks.wall.measurements.LeakageMeasurement method), 58
init() (qubricks.wall.operators.PauliOperatorSet method), 56
init() (qubricks.wall.stateoperators.DummyStateOperator method), 56
init() (qubricks.wall.stateoperators.LindbladStateOperator method), 56
init() (qubricks.wall.stateoperators.SchrodingerStateOperator method), 57
init() (qubricks.wall.systems.SimpleQuantumSystem method), 55
init_bases() (qubricks.system.QuantumSystem method), 25
init_bases() (qubricks.wall.systems.SimpleQuantumSystem method), 55
init_derivative_ops() (qubricks.system.QuantumSystem method), 25
init_derivative_ops() (qubricks.wall.systems.SimpleQuantumSystem method), 55
init_hamiltonian() (qubricks.system.QuantumSystem method), 25
init_hamiltonian() (qubricks.wall.systems.SimpleQuantumSystem method), 55
init_measurements() (qubricks.system.QuantumSystem method), 25
init_measurements() (qubricks.wall.systems.SimpleQuantumSystem method), 55
init_parameters() (qubricks.system.QuantumSystem method), 25
init_states() (qubricks.system.QuantumSystem method), 26
init_states() (qubricks.wall.systems.SimpleQuantumSystem method), 55
initial (qubricks.integrator.Integrator attribute), 51
int_kwargs (qubricks.integrator.Integrator attribute), 51
integrate() (qubricks.measurement.MeasurementWrapper method), 41

integrate() (qubricks.system.QuantumSystem method), 26
Integrator (class in qubricks.integrator), 49
IntegratorCallback (class in qubricks.integrator), 53
inv() (qubricks.analysis.perturbation.RSPT method), 63
inverse() (qubricks.operator.Operator method), 31
inverse() (qubricks.operator.OrthogonalOperator method), 33
is_algebraic (qubricks.basis.QubricksBasis attribute), 47
is_algebraic (qubricks.basis.QubricksBra attribute), 48
is_algebraic (qubricks.basis.QubricksKet attribute), 49
is_commutative (qubricks.basis.QubricksBasis attribute), 47
is_commutative (qubricks.basis.QubricksBra attribute), 48
is_commutative (qubricks.basis.QubricksKet attribute), 49
is_complete() (qubricks.measurement.MeasurementIterationResults method), 39
is_complex (qubricks.basis.QubricksBasis attribute), 47
is_complex (qubricks.basis.QubricksBra attribute), 48
is_complex (qubricks.basis.QubricksKet attribute), 49
is_composite (qubricks.basis.QubricksBasis attribute), 47
is_composite (qubricks.basis.QubricksBra attribute), 48
is_composite (qubricks.basis.QubricksKet attribute), 49
is_even (qubricks.basis.QubricksBasis attribute), 47
is_even (qubricks.basis.QubricksBra attribute), 48
is_even (qubricks.basis.QubricksKet attribute), 49
is_imaginary (qubricks.basis.QubricksBasis attribute), 47
is_imaginary (qubricks.basis.QubricksBra attribute), 48
is_imaginary (qubricks.basis.QubricksKet attribute), 49
is_independent (qubricks.measurement.Measurement attribute), 36
is_integer (qubricks.basis.QubricksBasis attribute), 47
is_integer (qubricks.basis.QubricksBra attribute), 48
is_integer (qubricks.basis.QubricksKet attribute), 49
is_irrational (qubricks.basis.QubricksBasis attribute), 47
is_irrational (qubricks.basis.QubricksBra attribute), 48
is_irrational (qubricks.basis.QubricksKet attribute), 49
is_negative (qubricks.basis.QubricksBasis attribute), 47
is_negative (qubricks.basis.QubricksBra attribute), 48
is_negative (qubricks.basis.QubricksKet attribute), 49
is_noninteger (qubricks.basis.QubricksBasis attribute), 47
is_noninteger (qubricks.basis.QubricksBra attribute), 48
is_noninteger (qubricks.basis.QubricksKet attribute), 49
is_nonnegative (qubricks.basis.QubricksBasis attribute), 47
is_nonnegative (qubricks.basis.QubricksBra attribute), 48
is_nonnegative (qubricks.basis.QubricksKet attribute), 49
is_nonpositive (qubricks.basis.QubricksBasis attribute), 47
is_nonpositive (qubricks.basis.QubricksBra attribute), 48
is_nonpositive (qubricks.basis.QubricksKet attribute), 49
is_nonzero (qubricks.basis.QubricksBasis attribute), 47
is_nonzero (qubricks.basis.QubricksBra attribute), 48
is_nonzero (qubricks.basis.QubricksKet attribute), 49
is_odd (qubricks.basis.QubricksBasis attribute), 47
is_odd (qubricks.basis.QubricksBra attribute), 48
is_odd (qubricks.basis.QubricksKet attribute), 49
is_positive (qubricks.basis.QubricksBasis attribute), 47
is_positive (qubricks.basis.QubricksBra attribute), 48
is_positive (qubricks.basis.QubricksKet attribute), 49
is_prime (qubricks.basis.QubricksBasis attribute), 47
is_prime (qubricks.basis.QubricksBra attribute), 48
is_prime (qubricks.basis.QubricksKet attribute), 49
is_rational (qubricks.basis.QubricksBasis attribute), 47
is_rational (qubricks.basis.QubricksBra attribute), 48
is_rational (qubricks.basis.QubricksKet attribute), 49
is_real (qubricks.basis.QubricksBasis attribute), 47
is_real (qubricks.basis.QubricksBra attribute), 48
is_real (qubricks.basis.QubricksKet attribute), 49
is_transcendental (qubricks.basis.QubricksBasis attribute), 47
is_transcendental (qubricks.basis.QubricksBra attribute), 48
is_transcendental (qubricks.basis.QubricksKet attribute), 49
is_zero (qubricks.basis.QubricksBasis attribute), 47
is_zero (qubricks.basis.QubricksBra attribute), 48
is_zero (qubricks.basis.QubricksKet attribute), 49
iterate() (qubricks.measurement.MeasurementWrapper method), 41
iterate_continue_mask() (qubricks.measurement.Measurement method), 37
iterate_is_complete() (qubricks.measurement.Measurement method), 37
iterate_results_add() (qubricks.measurement.Measurement method), 37
iterate_results_init() (qubricks.measurement.Measurement method), 37
iterate_to_file() (qubricks.measurement.MeasurementWrapper method), 41
iterate_yielder() (qubricks.measurement.MeasurementWrapper method), 42

L

leakage() (qubricks.wall.measurements.LeakageMeasurement method), 58
LeakageMeasurement (class in qubricks.wall.measurements), 57
LindbladStateOperator (class in qubricks.wall.stateoperators), 56
load() (qubricks.measurement.MeasurementIterationResults class method), 39

M

measure() (qubricks.measurement.Measurement

method), 37
measure() (qubricks.wall.measurements.AmplitudeMeasurement method), 57
measure() (qubricks.wall.measurements.ExpectationMeasurement method), 57
measure() (qubricks.wall.measurements.LeakageMeasurement method), 58
Measurement (class in qubricks.measurement), 36
MeasurementIterationResults (class in qubricks.measurement), 38
Measurements (class in qubricks.measurement), 43
measurements (qubricks.system.QuantumSystem attribute), 26
MeasurementWrapper (class in qubricks.measurement), 40
ModelAnalysis (class in qubricks.analysis.model), 65

O

on() (qubricks.measurement.MeasurementWrapper method), 42
onComplete() (qubricks.integrator.IntegratorCallback method), 53
onComplete() (qubricks.integrator.ProgressBarCallback method), 53
onProgress() (qubricks.integrator.IntegratorCallback method), 53
onProgress() (qubricks.integrator.ProgressBarCallback method), 53
onStart() (qubricks.integrator.IntegratorCallback method), 53
onStart() (qubricks.integrator.ProgressBarCallback method), 53
Operator (class in qubricks.operator), 28
operator (qubricks.basis.Basis attribute), 44
operator (qubricks.wall.bases.SimpleBasis attribute), 58
operator (qubricks.wall.bases.SpinBasis attribute), 58
operator (qubricks.wall.bases.StandardBasis attribute), 59
Operator() (qubricks.basis.Basis method), 44
Operator() (qubricks.stateoperator.StateOperator method), 34
Operator() (qubricks.system.QuantumSystem method), 22
operators (qubricks.integrator.Integrator attribute), 51
OperatorSet (class in qubricks.operator), 32
OperatorSet() (qubricks.system.QuantumSystem method), 22
optimise_coefficient() (qubricks.wall.stateoperators.LindbladStateOperator method), 56
OrthogonalOperator (class in qubricks.operator), 33

P

p (qubricks.basis.Basis attribute), 45
p (qubricks.integrator.Integrator attribute), 52
p (qubricks.operator.Operator attribute), 31
p (qubricks.stateoperator.StateOperator attribute), 35
p (qubricks.system.QuantumSystem attribute), 26
params (qubricks.integrator.Integrator attribute), 52
PauliOperatorSet (class in qubricks.wall.operators), 55
Perturb (class in qubricks.analysis.perturbation), 61
plot() (qubricks.analysis.model.ModelAnalysis method), 65
prepare() (qubricks.analysis.model.ModelAnalysis method), 65
process() (qubricks.analysis.model.ModelAnalysis method), 65
Progress (class in qubricks.integrator), 53
progress_callback (qubricks.integrator.Integrator attribute), 52
ProgressBarCallback (class in qubricks.integrator), 53
Psi() (qubricks.analysis.perturbation.Perturb method), 62
Psi() (qubricks.analysis.perturbation.RSPT method), 63
Psis() (qubricks.analysis.perturbation.Perturb method), 62
pt (qubricks.analysis.perturbation.Perturb attribute), 62

Q

QuantumSystem (class in qubricks.system), 21
qubricks.analysis.model (module), 65
qubricks.analysis.perturbation (module), 61
qubricks.analysis.spectrum (module), 64
qubricks.basis (module), 44
qubricks.integrator (module), 49
qubricks.measurement (module), 36
qubricks.operator (module), 28
qubricks.stateoperator (module), 33
qubricks.system (module), 21
qubricks.wall.bases (module), 58
qubricks.wall.measurements (module), 57
qubricks.wall.operators (module), 55
qubricks.wall.stateoperators (module), 56
qubricks.wall.systems (module), 55
QubricksBasis (class in qubricks.basis), 47
QubricksBra (class in qubricks.basis), 47
QubricksKet (class in qubricks.basis), 48

R

reset() (qubricks.integrator.Integrator method), 52
restrict() (qubricks.operator.Operator method), 31
restrict() (qubricks.stateoperator.StateOperator method), 35
restrict() (qubricks.wall.stateoperators.DummyStateOperator method), 56
restrict() (qubricks.wall.stateoperators.LindbladStateOperator method), 57
restrict() (qubricks.wall.stateoperators.SchrodingerStateOperator method), 57
result_shape() (qubricks.measurement.Measurement method), 38

result_shape() (qubricks.wall.measurements.AmplitudeMeasurement method), 57
result_shape() (qubricks.wall.measurements.ExpectationMeasurement method), 57
result_shape() (qubricks.wall.measurements.LeakageMeasurement method), 58
result_type() (qubricks.measurement.Measurement method), 38
result_type() (qubricks.wall.measurements.AmplitudeMeasurement method), 57
result_type() (qubricks.wall.measurements.ExpectationMeasurement method), 57
result_type() (qubricks.wall.measurements.LeakageMeasurement method), 58
result_units (qubricks.wall.measurements.LeakageMeasurement attribute), 58
results (qubricks.integrator.Integrator attribute), 52
RSPT (class in qubricks.analysis.perturbation), 62

S

save() (qubricks.measurement.MeasurementIterationResults method), 40
SchrodingerStateOperator (class in qubricks.wall.stateoperators), 57
shape (qubricks.operator.Operator attribute), 32
show() (qubricks.system.QuantumSystem method), 26
SimpleBasis (class in qubricks.wall.bases), 58
SimpleQuantumSystem (class in qubricks.wall.systems), 55
simulate() (qubricks.analysis.model.ModelAnalysis method), 65
SpinBasis (class in qubricks.wall.bases), 58
StandardBasis (class in qubricks.wall.bases), 58
start() (qubricks.integrator.Integrator method), 52
state() (qubricks.system.QuantumSystem method), 26
state_fromString() (qubricks.basis.Basis method), 45
state_fromString() (qubricks.system.QuantumSystem method), 27
state_fromString() (qubricks.wall.bases.SpinBasis method), 58
state_fromSymbolic() (qubricks.basis.Basis method), 45
state_info() (qubricks.basis.Basis method), 45
state_info() (qubricks.wall.bases.SpinBasis method), 58
state_latex() (qubricks.basis.Basis method), 45
state_latex() (qubricks.wall.bases.SpinBasis method), 58
state_projector() (qubricks.system.QuantumSystem method), 27
state_toString() (qubricks.basis.Basis method), 45
state_toString() (qubricks.system.QuantumSystem method), 27
state_toString() (qubricks.wall.bases.SpinBasis method), 58
state_toSymbolic() (qubricks.basis.Basis method), 45
StateOperator (class in qubricks.stateoperator), 33
states (qubricks.system.QuantumSystem attribute), 27
states() (qubricks.basis.Basis method), 45
str_label (qubricks.basis.QubricksKet attribute), 49
subspace() (qubricks.system.QuantumSystem method), 27
subspace_projector() (qubricks.system.QuantumSystem method), 28
subspaces (qubricks.system.QuantumSystem attribute), 28
symbolic() (qubricks.operator.Operator method), 32
system (qubricks.measurement.Measurement attribute), 38

T

t_offset (qubricks.integrator.Integrator attribute), 52
tensor() (qubricks.operator.Operator method), 32
time_ops (qubricks.integrator.Integrator attribute), 52
transform() (qubricks.basis.Basis method), 46
transform() (qubricks.operator.Operator method), 32
transform() (qubricks.stateoperator.StateOperator method), 35
transform() (qubricks.wall.stateoperators.DummyStateOperator method), 56
transform() (qubricks.wall.stateoperators.LindbladStateOperator method), 57
transform() (qubricks.wall.stateoperators.SchrodingerStateOperator method), 57
transform_from() (qubricks.basis.Basis method), 46
transform_op() (qubricks.basis.Basis method), 46
transform_to() (qubricks.basis.Basis method), 47

U

update() (qubricks.measurement.MeasurementIterationResults method), 40
use_ensemble() (qubricks.system.QuantumSystem method), 28

V

V (qubricks.analysis.perturbation.RSPT attribute), 63