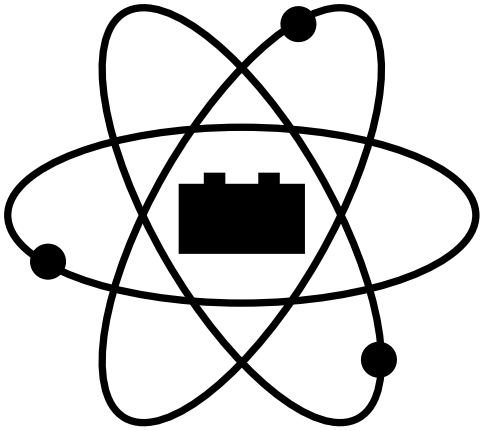


QuBricks

This documentation is a work in progress. If you are interested in using QuBricks, please get in touch with me at mister.wardrop@gmail.com; which will greatly enhance my motivation toward completing this documentation. If you need QuBricks to be extended for your work, I am more than willing to be involved.

QuBricks Documentation

February 26, 2015

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**??** 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.

1 Installation

QuBricks depends upon `python-parameters` >= 1.1.9, which is available from GitHub or pypi. Once python-parameters is installed, installing the QuBricks module is done in one line:

```
$ python2 setup.py install
```

If you run Arch Linux, you can instead install `python-qubricks` via your package manager using:

```
$ makepkg -i
```

2 Usage Overview

The main purpose of this documentation is to demonstrate how to use QuBricks, rather than to explain the mechanisms of how it works; though there is obviously some overlap. In section 2.1 we demonstrate some extremely basic use cases, before exploring more interesting cases in section 2.2.

In the following, we assume that the following has been run:

```
>>> from qubricks import Operator, QuantumSystem...
>>> import numpy as np
```

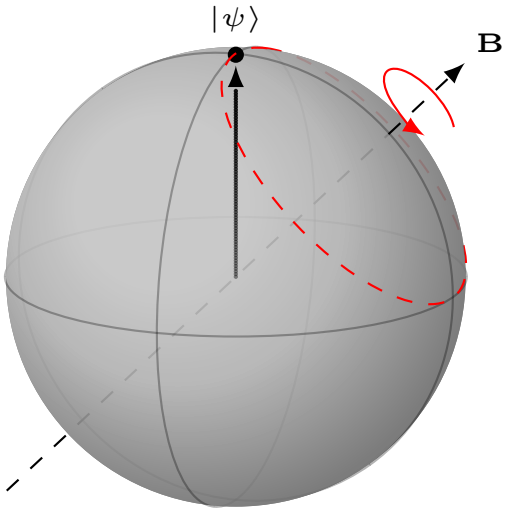


Figure 1: Dynamics of a single electron spin under a magnetic field aligned 45 degrees in the XZ plane.

2.1 Basic Usage

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 aligned with Z axis, and a noisy magnetic field along the X axis. The Hamiltonian describing the mechanics is:

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

where B_z is the magnetic field along z and $B_x = B_{x0} + \tilde{B}_x$ is a noise field along x centred on some nominal value B_{x0} with a high frequency noise component $\tilde{B}_x$. We assume the noise is white:

$$\langle (B_x - B_{x0})_t (B_x - B_{x0})_{t'} \rangle = D\delta(t - t'),$$

and model it using a Lindblad superoperator.

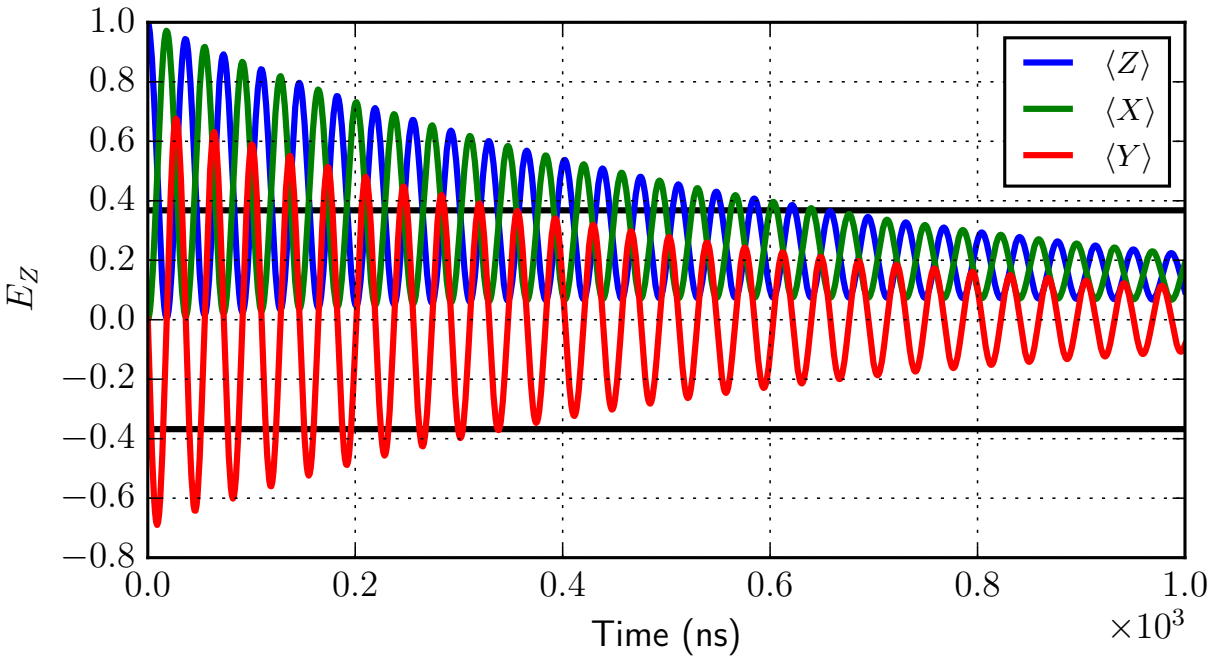
This is a simple system which can be analytically solved easily in the absence of noise. Evolution under this Hamiltonian will lead to the electron's spin gyrating around an axis between Z and X (i.e. at an angle $\theta = \tan^{-1}(B_z/J_x)$ from the x-axis) at a frequency of $2\sqrt{B_z^2 + B_x^2}$.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

The dynamics of the system can be simulated using the code below, where we have chosen $B_z = B_x$, and so dynamics should evolve as shown in figure 1.

```
>>> from qubricks import Operator, QuantumSystem...
>>> import numpy as np
```



2.2 Advanced Usage

For more fine-grained control, one can subclass `QuantumSystem`, `Measurement` and `StateOperator` as necessary. For more information about which methods are available on these objects, please use the inline python help: help(python-obj). The templates are as follows:

```
from qubricks import QuantumSystem

class CustomSystem(QuantumSystem):

    def setup_environment(self, **kwargs):
        '''
        Configure any custom properties/attributes using kwargs passed
        to __init__.
        '''
        raise NotImplementedError

    def setup_parameters(self):
        '''
        After the QuantumSystem parameter initialisation routines
        are run, check that the parameters are initialised correctly.
        '''
        raise NotImplementedError

    def setup_bases(self):
        '''
        Add the bases which are going to be used by this QuantumSystem
        instance.
        '''
        raise NotImplementedError

    def setup_hamiltonian(self):
        '''
        Initialise the Hamiltonian to be used by this QuantumSystem
        instance.
        '''
```

```
        raise NotImplementedError

    def setup_states(self):
        '''
        Add the named/important states to be used by this quantum system.
        '''
        raise NotImplementedError

    def setup_measurements(self):
        '''
        Add the measurements to be used by this quantum system instance.
        '''
        raise NotImplementedError

    @property
    def default_derivative_ops(self):
        raise NotImplementedError

    # Optional: can use self.add_derivative_op instead
    def get_derivative_ops(self, components=None):
        '''
        Setup the derivative operators to be implemented on top of the
        basic quantum evolution operator.
        '''
        raise NotImplementedError
```

```
from qubricks import Measurement

class CustomMeasurement(Measurement):

    def init(self,**kwargs):
        '''
        Measurement.init should be specified by measurement subclasses.
        '''
        raise NotImplementedError("Measurement.init_measurement has not been implemented.")

    def measure(self,times,y_0s,params={},**kwargs):
        '''
        Measurement.measure is where the grunt work is done; and it should return
        a numpy array of type Measurement.result_type, with shape
        Measurement.result_shape . Otherwise, anything is permitted in this method.
        '''
        raise NotImplementedError("Measurement.measure has not been implemented.")

    def result_type(self,*args,**kwargs):
        '''
        Measurement.result_type should return an object suitable for use as the dtype
        argument in numpy. Otherwise, no restrictions; other than that it must also
        agree with the datatype returned by Measurement.measure.
        '''
        raise NotImplementedError("Measurement.result_type has not been implemented.")

    def result_shape(self,*args,**kwargs):
        '''
        Measurement.result_shape should agree with the shape of the numpy array returned
        by Measurement.measure, but otherwise has no restrictions.
        '''
        raise NotImplementedError("Measurement.result_shape has not been implemented.")
```

```
from qubricks import StateOperator

```

```
class CustomStateOperator(StateOperator):

    def init(self, **kwargs):
        '''
        StateOperator.init is called when StateOperator subclasses are
        initialised, which allows subclasses to set themselves up as appropriate.
        '''
        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_ensemble
        which kinds of states are supported.
        '''
        raise NotImplementedError("StateOperator.__call__ is not implemented.")

    def restrict(self, *indicies):
        '''
        StateOperator.restrict should return a new StateOperator restricted to the basis states
        with indicies `indicies`.
        '''
        raise NotImplementedError("StateOperator.restrict is not implemented.")

    def connected(self, *indicies, **params):
        '''
        StateOperator.connected should return the list of basis state indicies which would mix
        with the specified basis state indicies `indicies` under repeated operation of the
        StateOperator.
        '''
        raise NotImplementedError("StateOperator.connected is not implemented.")

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

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

    def transform(self, transform_op):
        '''
        StateOperator.transform should transform all operations on the state
        according to the basis transformation `transform_op`.
        '''
        raise NotImplementedError("StateOperator.transform is not implemented.")
```

```
from qubricks import Basis

class CustomBasis(Basis):

    def init(self, dim=None, **kwargs):
        '''
        Basis.init is called during the basis initialisation
```

```
        routines, allowing Basis subclasses to initialise themselves.
        '''
        pass

    @property
    def operator(self):
        '''
        Basis.operator must return an Operator object with basis states as the
        columns. The operator should use the parameters instance provided by the
        Basis subclass.
        '''
        raise NotImplementedError, "Basis operator has not been implemented."
```

3 QuBricks Internals

3.1 Operator

Perhaps the most important brick is the `Operator` object. 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(np.array([[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.

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 instead of the numpy array above. 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 section **??**), 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 **??**). 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].

In conjunction with functional dependence inherited from `python-parameters` this 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()
```

There is a subclass of the `Operator` class, `OrthogonalOperator`, which is for operators that have orthogonal eigenvectors; in which case the `inverse` operation can be greatly simplified.

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 a certain variables 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 for J) 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, in 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]])
```