

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` $\geq 1.1.9$, which is available from GitHub or pypi. Once `python-parameters` is installed, installing the QuBricks module is done in one line:

```
1 $ python2 setup.py install
```

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

```
1 $ 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 ?? we demonstrate some extremely basic use cases, before exploring more interesting cases in section ??.

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

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

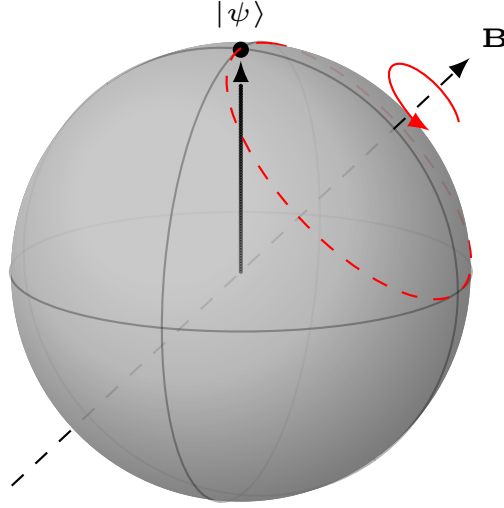


Figure 1: Dynamics of a single electron spin under a magnetic field aligned 45 degrees in the XZ plane.

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: $\langle \tilde{B}_x^2 \rangle_{\text{avg}} = D/t$, by taylor expanding we find: $\langle p \rangle = 1 - Dt$. A more careful analysis finds that:

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

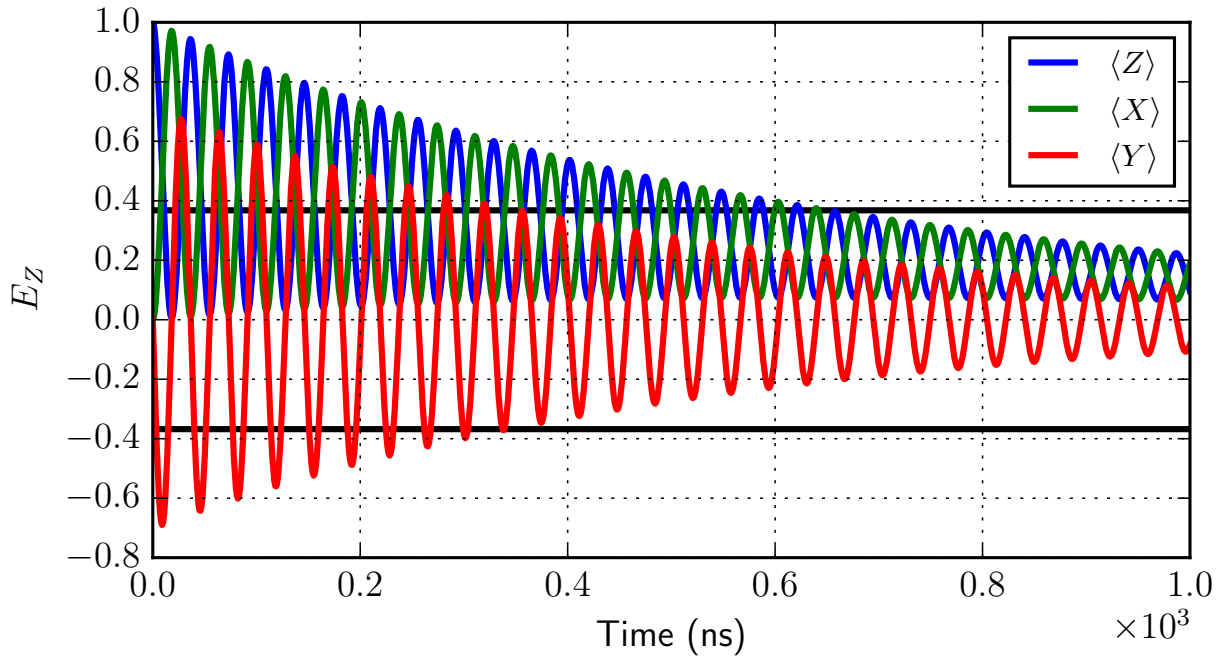
It is possible to find approximations for a general θ , but we leave that as an exercise.

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...
2 >>> 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:

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

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

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

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

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:

```

1 >>> op = Operator(np.array([[1,2],[3,4]]))
2 >>> op
3 <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.

```

1 >>> op() # Numeric representation as a NumPy array
2 array([[ 1.+0.j,  2.+0.j],
3        [ 3.+0.j,  4.+0.j]])
4 >>> op.symbolic() # Symbolic representation as a SymPy matrix
5 Matrix([
6  [1, 2],
7  [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:

```

1 >>> op = Operator({'B':[[1,0],[0,-1]], 'J':[[0,1],[1,0]]})
2 >>> op.symbolic()
3 Matrix([
4  [B,  J],
5  [J, -B]])
6 >>> op.symbolic(J=10)
7 Matrix([
8  [ B, 10],
9  [10, -B]])
10 >>> op()
11 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 should 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.

```

1 >>> from parameters import Parameters
2 >>> p = Parameters()
3 >>> p(B=2,J=1)
4 <Parameters with 2 definitions >
5 >>> op = Operator({'B':[[1,0],[0,-1]], 'J':[[0,1],[1,0]]},parameters=p)
6 >>> op()
7 array([[ 2.+0.j,  1.+0.j],
8        [ 1.+0.j, -2.+0.j]])
9 >>> op(J=10,B=1)
10 array([[ 1.+0.j, 10.+0.j],
11         [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:

```
1 >>> 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:

```
1 >>> 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.

```
1 >>> 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.

```
1 >>> op = Operator({'B':[[1,0],[0,-1]], 'J':[[0,1],[1,0]]},parameters=p)
2 >>> op.connected(0)
3 {0,1}
4 >>> op.connected(0,J=0)
5 {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.

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