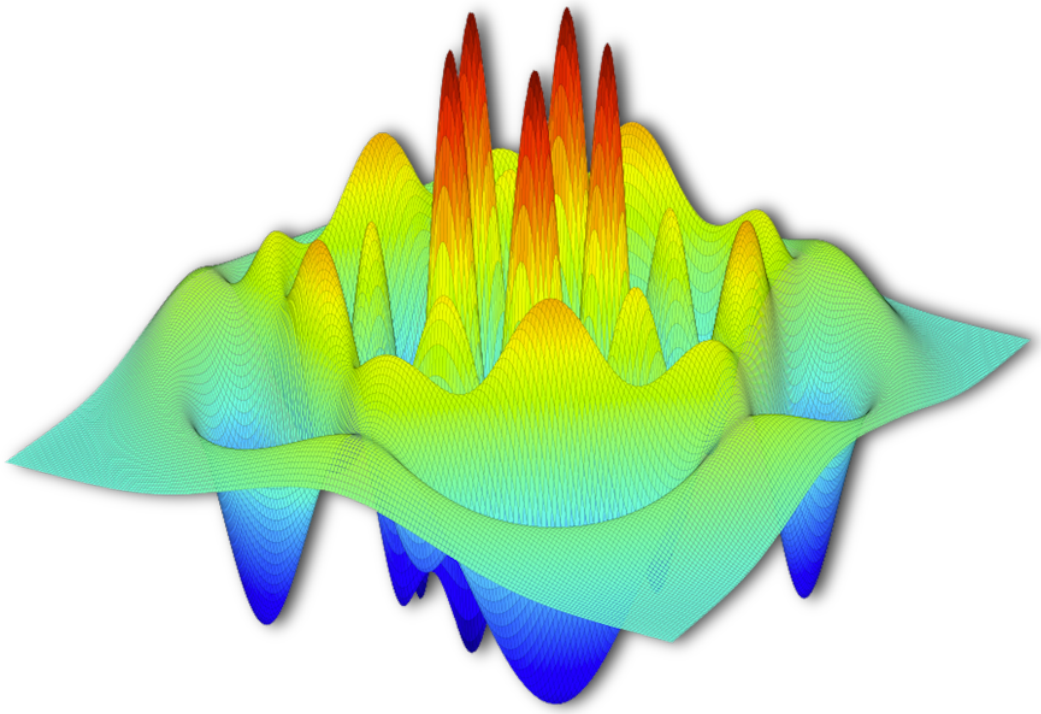


QuTiP: The Quantum Toolbox in Python

Release 2.2.0

P.D. Nation and J.R. Johansson

March 01, 2013

CONTENTS

CHAPTER

ONE

FRONTMATTER

1.1 About This Documentation

This document contains a user guide and automatically generated API documentation for QuTiP. A PDF version of this text is available at the download page.

For more information see the QuTiP project web page.

Author P.D. Nation

Address Department of Physics, Korea University, Seongbuk-gu Seoul, 136-713 South Korea

Author J.R. Johansson

Address Advanced Science Institute, RIKEN, Wako-shi Saitama, 351-0198 Japan

version 2.2

status In Development

1.2 Citing This Project

If you find this project useful, then please cite:

J. R. Johansson, P.D. Nation, and F. Nori, "QuTiP 2: A Python framework for the dynamics of open quantum systems", Comp. Phys. Comm. **184**, 1234 (2013).

or

J. R. Johansson, P.D. Nation, and F. Nori, "QuTiP: An open-source Python framework for the dynamics of open quantum systems", Comp. Phys. Comm. **183**, 1760 (2012).

which may also be download from http://arxiv.org/abs/1211.6518 or http://arxiv.org/abs/1110.0573, respectively.

1.3 Funding

The development of QuTiP has been partially supported by the Japanese Society for the Promotion of Science Foreign Postdoctoral Fellowship Program under grants P11202 (PDN) and P11501 (JRJ). Additional funding comes from RIKEN, Kakenhi grant Nos. 2301202 (PDN), 2302501 (JRJ), and Korea University.

日本学術振興会
Japan Society for the Promotion of Science

1.4 Contributing to QuTiP

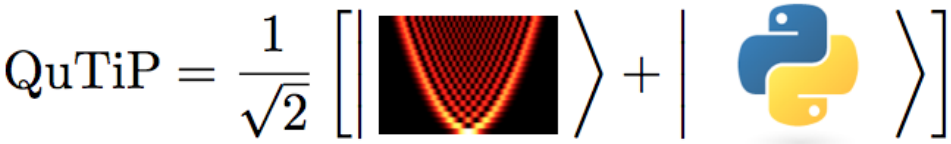
Like any open-source project, we welcome anyone who is interested in helping us make QuTiP the best package for simulating quantum optics-like systems. Anyone who contributes will be duly recognized. Even small contributions are noted. See *Contributors* for a list of people who have helped in one way or another. If you are interested, please drop us a line at the QuTiP discussion group webpage.

고려대학교
KOREA UNIVERSITY

CHAPTER

TWO

ABOUT QUTIP



2.1 Brief Description

Every quantum system encountered in the real world is an open quantum system. For although much care is taken experimentally to eliminate the unwanted influence of external interactions, there remains, if ever so slight, a coupling between the system of interest and the external world. In addition, any measurement performed on the system necessarily involves coupling to the measuring device, therefore introducing an additional source of external influence. Consequently, developing the necessary tools, both theoretical and numerical, to account for the interactions between a system and its environment is an essential step in understanding the dynamics of quantum systems.

In general, for all but the most basic of Hamiltonians, an analytical description of the system dynamics is not possible, and one must resort to numerical simulations of the equations of motion. In absence of a quantum computer, these simulations must be carried out using classical computing techniques, where the exponentially increasing dimensionality of the underlying Hilbert space severely limits the size of system that can be efficiently simulated. However, in many fields such as quantum optics, trapped ions, superconducting circuit devices, and most recently nanomechanical systems, it is possible to design systems using a small number of effective oscillator and spin components, excited by a small number of quanta, that are amenable to classical simulation in a truncated Hilbert space.

The Quantum Toolbox in Python, or QuTiP, is a fully open-source implementation of a framework written in the Python programming language designed for simulating the open quantum dynamics for systems such as those listed above. This framework distinguishes itself from the other available software solutions by providing the following advantages:

- QuTiP relies on completely open-source software. You are free to modify and use it as you wish with no licensing fees.
- QuTiP is based on the Python scripting language, providing easy to read, fast code generation without the need to compile after modification.
- The numerics underlying QuTiP are time-tested algorithms that run at C-code speeds, thanks to the Numpy and Scipy libraries, and are based on many of the same algorithms used in propriety software.
- QuTiP allows for solving the dynamics of Hamiltonians with arbitrary time-dependence, including collapse operators.
- Time-dependent problems can be automatically compiled into C-code at run-time for increased performance.
- Takes advantage of the multiple processing cores found in essentially all modern computers.

- QuTiP was designed from the start to require a minimal learning curve for those users who have experience using the popular quantum optics toolbox by Sze M. Tan.
- Includes the ability to create high-quality plots, and animations, using the excellent Matplotlib package.

2.2 Whats New in QuTiP Version 2.2

- **QuTiP is now 100% compatible with the Windows Operating System!**
- New Bloch3d class for plotting 3D Bloch spheres using Mayavi.
- Added partial transpose function.
- New Wigner colormap for highlighting negative values.
- Bloch sphere vectors now look like arrows instead of lines.
- Continous variable functions for calculating correlation and covariance matrices, the Wigner covariance matrix and the logarithmic negativity for multimode field states expressed in the Fock basis.
- The master-equation solver (mesolve) now accepts pre-constructed Liouvillian terms, which makes it possible to solve master equations that are not on the standard Lindblad form.
- Optional Fortran Monte Carlo solver (mcsolve_f90) by Arne Grimsmo.
- A module with tools for using QuTiP in IPython notebooks.
- Added more graph styles to the visualization module.
- Increased performance of the steady state solver.

2.3 Whats New in QuTiP Version 2.1

- New method for generating Wigner functions based on Laguerre polynomials.
- coherent(), coherent_dm(), and thermal_dm() can now be expressed using analytic values.
- Unittests now use nose and can be run after installation.
- Added iswap and sqrt-iswap gates.
- Functions for quantum process tomography.
- Window icons are now set for Ubuntu application launcher.

2.4 Whats New in QuTiP Version 2.0

The second version of QuTiP has seen many improvements in the performance of the original code base, as well as the addition of several new routines supporting a wide range of functionality. Some of the highlights of this release include:

- QuTiP now includes solvers for both Floquet and Bloch-Redfield master equations.
- The Lindblad master equation and Monte Carlo solvers allow for time-dependent collapse operators.
- It is possible to automatically compile time-dependent problems into c-code using Cython (if installed).
- Python functions can be used to create arbitrary time-dependent Hamiltonians and collapse operators.
- Solvers now return Odedata objects containing all simulation results and parameters, simplifying the saving of simulation results.

Important: This breaks compatibility with QuTiP version 1.x. See *The Odedata Class and Dynamical Simulation Results* for further details.

- mesolve and mcsolve can reuse Hamiltonian data when only the initial state, or time-dependent arguments, need to be changed.
- QuTiP includes functions for creating random quantum states and operators.
- The generation and manipulation of quantum objects is now more efficient.
- Quantum objects have basis transformation and matrix element calculations as built-in methods.
- The quantum object eigensolver can use sparse solvers.
- The partial-trace (ptrace) function is up to 20x faster.
- The Bloch sphere can now be used with the Matplotlib animation function, and embedded as a subplot in a figure.
- QuTiP has built-in functions for saving quantum objects and data arrays.
- The steady-state solver has been further optimized for sparse matrices, and can handle much larger system Hamiltonians.
- The steady-state solver can use the iterative bi-conjugate gradient method instead of a direct solver.
- There are three new entropy functions for concurrence, mutual information, and conditional entropy.
- Correlation functions have been combined under a single function.
- The operator norm can now be set to trace, Frobius, one, or max norm.
- Global QuTiP settings can now be modified.
- QuTiP includes a collection of unit tests for verifying the installation.
- Demos window now lets you copy and paste code from each example.

CHAPTER

THREE

INSTALLATION

3.1 General Requirements

QuTiP is based on several open-source packages designed for numerical simulations in the Python programming language. Currently. QuTiP requires the following packages to run:

Package	Version	Details
Python	2.6+	Requires multiprocessing (v2.6 and higher only).
Numpy	1.6+	Not tested on lower versions.
Scipy	0.9+	Not tested on lower versions. Use 0.9+ if possible.
Matplotlib	1.1.0+	Some plotting does not work on lower versions.
GCC Compiler	4.2+	Needed for compiling Cython files.
Qt	4.7.3+	Optional. For GUI elements only.
PySide	1.0.6+	Optional, required only for GUI elements. PyQt4 may be used instead.
PyQt4	4.8+	Optional, required only for GUI elements. PySide may be used instead (recommended).
PyObjC	2.2+	Mac only. Very optional. Needed only for a GUI Monte Carlo progress bar.
Cython	0.15+	Optional. Needed for compiling some time-dependent Hamiltonians.
BLAS library	1.2+	Optional, Linux & Mac only. Needed for installing Fortran Monte Carlo solver.
Mayavi	4.1+	Optional. Needed for using the Bloch3d class.
Python Headers	2.6+	Linux only. Needed for compiling Cython files.
LaTeX	TexLive 2009+	Optional. Needed if using LaTeX in figures.
nose	1.1.2+	Optional. For running tests.

On all platforms (Linux, Mac, Windows), QuTiP works "out-of-the-box" using the Anaconda CE. This distribution is created by the developers of Numpy, and is free for both commercial and noncommercial use.

As of version 2.2, QuTiP includes an optional Fortran-based Monte Carlo solver that has a substantial performance benefit when compared with the Python-based solver. In order to install this package you must have a Fortran compiler (for example gfortran) and BLAS development libraries. At present, these packages are only tested on the Linux and OS X platforms.

3.2 Get the software

Official releases of QuTiP are available from the download section on the project's web pages

http://code.google.com/p/qutip/downloads

and the latest source code is available in our Github repository

http://github.com/qutip

In general we recommend users to use the latest stable release of QuTiP, but if you are interested in helping us out with development or wish to submit bug fixes, then use the latest development version from the Github repository.

3.3 Installation on Ubuntu Linux

3.3.1 Using QuTiP's PPA

The easiest way to install QuTiP in Ubuntu (12.04 and later) is to use the QuTiP PPA:

```
sudo add-apt-repository ppa:jrjohansson/qutip-releases
sudo apt-get update
sudo apt-get install python-qutip
```

With this method the most important dependencies are installed automatically, and when a new version of QuTiP is released it can be upgraded through the standard package management system. In addition to the required dependencies, it is also strongly recommended that you install the `texlive-latex-extra` package:

```
sudo apt-get install texlive-latex-extra
```

3.3.2 Manual installation

First install the following dependency packages:

```
sudo apt-get install python-scipy
sudo apt-get install python-pyside
sudo apt-get install python-setuptools
sudo apt-get install python-dev
sudo apt-get install python-matplotlib
sudo apt-get install cython
sudo apt-get install python-nose         # recommended, for testing
sudo apt-get install texlive-latex-extra # recommended
sudo apt-get install libblas-dev         # optional, for Fortran Monte Carlo solver
sudo apt-get install mayavi2             # optional, for Bloch3d only
```

For a standard installation, run this command in the QuTiP source code directory:

```
sudo python setup.py install
```

To install QuTiP with the optional Fortran Monte Carlo solver use:

```
sudo python setup.py install --with-f90mc
```

However, this additionally requires a Fortran compiler to be installed. For example the GNU Fortran compiler, which can be installed using:

```
sudo apt-get install gfortran
```

Note: Ubuntu 11.04 and lower do not have Matplotlib>=1.0, but we can use the following unofficial matplotlib ppa to install a newer version of matplotlib on these Ubuntu releases:

```
sudo add-apt-repository ppa:bgamari/matplotlib-unofficial
sudo apt-get update
sudo apt-get install python-matplotlib
```

Note: On some versions of Ubuntu you might have to configure Matplotlib to use the GTKAgg or Qt4Agg backends instead of the default TkAgg backend. To do change backend, edit /etc/matplotlibrc or ~/.matplotlib/matplotlibrc, and change `backend: TkAgg` to `backend: GTKAgg` or `backend: Qt4Agg`.

3.4 Installation on Mac OS X (10.6+)

If you have not done so already, install the Apple Xcode developer tools from the Apple App Store. After installation, open Xcode and go to: Preferences -> Downloads, and install the 'Command Line Tools'.

3.4.1 Setup Using Macports [1]

On the Mac, it is recommended that you install the required libraries via MacPorts. After installation, the necessary "ports" for QuTiP may be installed via:

```
sudo port install py27-scipy
sudo port install py27-matplotlib +latex
```

and in addition:

```
sudo port install py27-pyside   # recommended
```

or:

```
sudo port install py27-pyqt4
```

Optional, but highly recommended ports include:

```
sudo port install py27-ipython +pyside+notebook+parallel+scientific  #switch to +pyqt4 if using py
sudo port install py27-cython              #used for string-based time-dependent Hamiltonians
sudo port install vtk5 +python27+qt4_mac  #used for the Bloch3d class
sudo port install py27-mayavi             #used for the Bloch3d class
```

Now, we want to tell OSX which Python and iPython we are going to use:

```
sudo port select python python27
sudo port select ipython ipython27
```

Note: The next step is optional, but is necessary if you plan to use the string (Cython) based time-dependent format. See *Solving Problems with Time-dependent Hamiltonians*.

Finally, we want to set the macports compiler to the vanilla GCC version. From the command line type:

```
port select gcc
```

which will bring up a list of installed compilers, such as:

```
Available versions for gcc:
        apple-gcc42
        gcc42
        llvm-gcc42
        mp-gcc47
        none (active)
```

We want to set the the compiler to the gcc4x compiler, where x is the highest number available, in this case `mp-gcc47` (the "mp-" does not matter). To do this type:

```
sudo port select gcc mp-gcc47
```

Running port select again should give:

[1] Installing QuTiP via Macports will take a long time as each of the QuTiP dependencies is build from source code. The advantage is that, after installation, everything is more or less guaranteed to work. However, if you have a hot date waiting for you, then we do not recommend this path. Or course, if you are reading this guide, this may not be the case.

```
Available versions for gcc:
       apple-gcc42
       gcc42
       llvm-gcc42
       mp-gcc47 (active)
       none
```

3.4.2 Setup via SciPy Superpack

A second option is to install the required Python packages using the SciPy Superpack. Further information on installing the superpack can be found on the SciPy Downloads page. Note that, if you choose this option, the GUI elements of QuTiP will not be available without further installing either the PyQt4 or PySide packages separately.

3.4.3 Anaconda CE Distribution

Finally, one can also use the Anaconda CE package to install all of the QuTiP dependencies.

3.4.4 Installing QuTiP

No matter which installation path you choose, installing a standard QuTiP installation is the same as on linux. From the QuTiP directory run:

```
sudo python setup.py install
```

In order to install the Fortran Monte Carlo solver use the following command:

```
sudo python setup.py install --with-f90mc
```

3.5 Installation on Windows

QuTiP is primarily developed for Unix-based platforms such as Linux an Mac OS X, but it can also be used on Windows. We have limited experience and ability to help troubleshoot problems on Windows, but the following installation steps have been reported to work:

1. Install the Python(X,Y) distribution (tested with version 2.7.3.1). Other Python distributions, such as Enthought Python Distribution or Anaconda CE might work too, but this has not been verified.
2. When installing Python(x,y), explicitly select to include the Cython package in the installation. This package is not selected by default.
3. Add the following content to the file *C:/Python27/Lib/distutils/distutils.cfg* (or create the file if it does not already exists):

   ```
   [build]
   compiler = mingw32

   [build_ext]
   compiler = mingw32
   ```

The directory where the distutils.cfg file should be placed might be different if you have installed the Python environment in a different location than in the example above.

4. Obtain the QuTiP source code, unpack it and run the following command in the source code directory:

   ```
   python setup.py install
   ```

3.6 Verifying the Installation

QuTiP now includes a collection of built-in test scripts to verify that the installation was indeed successful. To run the suite of tests scripts you must have the nose testing library. After installing QuTiP, exit the installation directory, run Python (or iPython), and call:

```
>>> import qutip.testing as qt
>>> qt.run()
```

If successful, these tests indicate that all of the QuTiP functions are working properly. If any errors occur, please check that your have installed all of the required modules. See the next section on how to check the installed versions of the QuTiP dependencies. If these tests still fail, then head on over to the QuTiP Discussion Board and post a message detailing your particular issue.

To further verify that all of the QuTiP components are working, you can run the collection of examples built into QuTiP as discussed in the *QuTiP Example Scripts* section of the guide.

3.7 Checking Version Information via the About Box

QuTiP includes a graphical "about" box for viewing information about QuTiP, and the important dependencies installed on your system. To view the about box type:

```
>>> from qutip import *
>>> about()
```

That will pop up a window similar to the one shown below. If instead you get command-line output, then your PyQt or PySide graphics are not installed properly or unavailable. When running the about box, QuTiP will automatically check for a newer version of itself from the QuTiP website. The about box will have an "update" link next to the QuTiP version number if your are not running the latest version of QuTiP.

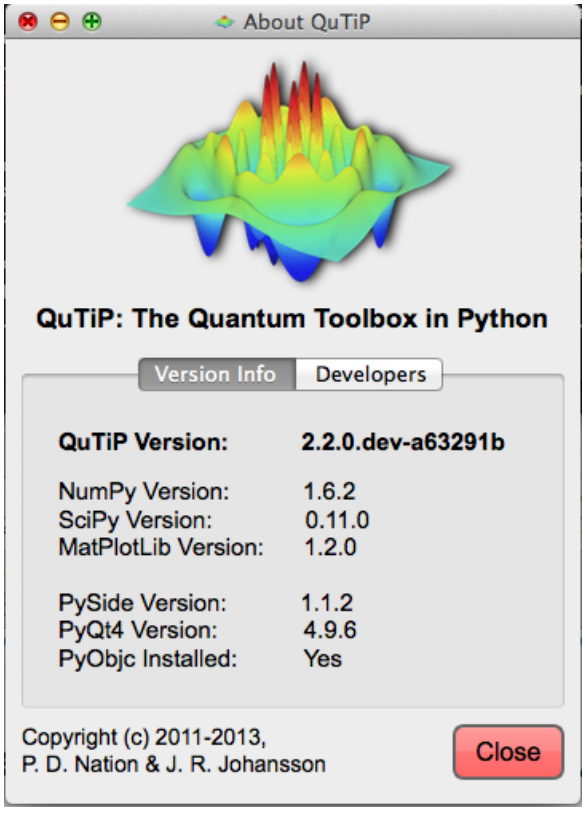


Figure 3.1: QuTiP about box window.

CHAPTER

FOUR

QUTIP USERS GUIDE

4.1 Guide Overview

The goal of this guide is to introduce you to the basic structures and functions that make up QuTiP. If you are familiar with the Quantum Optics Toolbox, then you should have no problem getting used to QuTiP. This guide is broken up into several sections, each highlighting a specific set of functionalities. In combination with the examples, this guide should provide a more or less complete overview. In addition, *API documentation* for each function is located at the end of this guide.

4.1.1 Organization

QuTiP is designed to be a general toolbox for solving quantum optics like problems such as systems composed of two-level and harmonic oscillator components. To this end, the QuTiP framework is built from a large (and ever growing) library of functions; from `qutip.states.basis` to `qutip.wigner`. The general organization of QuTiP, highlighting the important functions available to the user, is shown in the *QuTiP tree-diagram of user accessible functions and classes.*

4.2 Basic Operations on Quantum Objects

4.2.1 First things first

Important: Do not run QuTiP from the installation directory.

To load the qutip modules, we must first call the import statement:

```
In [1]: from qutip import *
```

that will load all of the user available functions. We will also need to import the SciPy library with:

```
In [1]: from scipy import *
```

Note that, in the rest of the documentation, functions are written using *qutip.module.function()* notation which links to the corresponding function in the QuTiP API: *QuTiP Functions*. However, in calling *import* *, we have already loaded all of the QuTiP modules. Therefore, we will only need the function name and not the complete path when calling the function from the command line or a Python script.

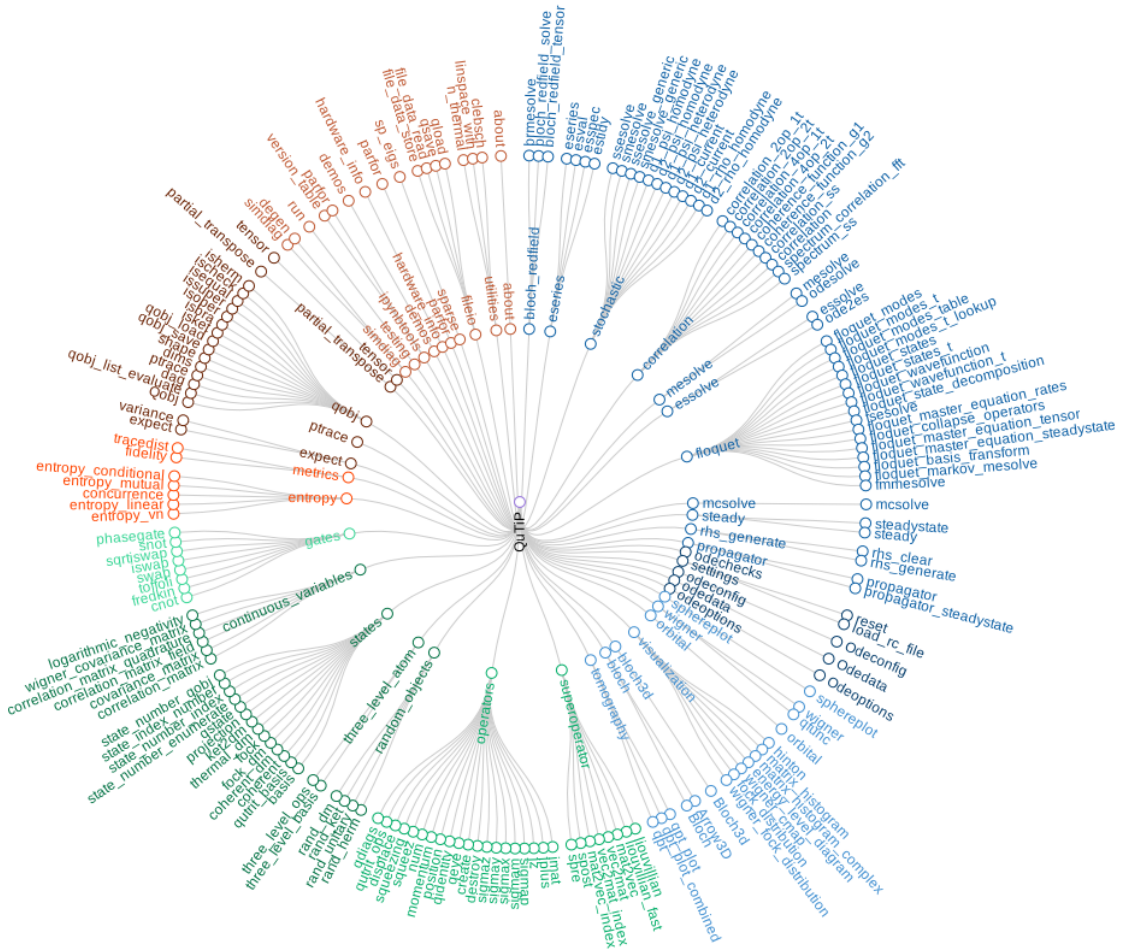


Figure 4.1: QuTiP tree-diagram of user accessible functions and classes.

4.2.2 The quantum object class

Introduction

The key difference between classical and quantum mechanics lies in the use of operators instead of numbers as variables. Moreover, we need to specify state vectors and their properties. Therefore, in computing the dynamics of quantum systems we need a data structure that is capable of encapsulating the properties of a quantum operator and ket/bra vectors. The quantum object class, `qutip.Qobj`, accomplishes this using matrix representation.

To begin, let us create a blank Qobj:

```
In [1]: Qobj()
Out[1]:
Quantum object: dims = [[1], [1]], shape = [1, 1], type = oper, isherm = True
Qobj data =
[[ 0.]]
```

where we see the blank Qobj object with dimensions, shape, and data. Here the data corresponds to a 1x1-dimensional matrix consisting of a single zero entry.

Hint: By convention, Class objects in Python such as *Qobj()* differ from functions in the use of a beginning capital letter.

We can create a Qobj with a user defined data set by passing a list or array of data into the Qobj:

```
In [1]: Qobj([1,2,3,4,5])
Out[1]:
Quantum object: dims = [[1], [5]], shape = [1, 5], type = bra
Qobj data =
[[ 1.  2.  3.  4.  5.]]

In [2]: x = array([[1],[2],[3],[4],[5]])

In [3]: Qobj(x)
Out[3]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 1.]
 [ 2.]
 [ 3.]
 [ 4.]
 [ 5.]]

In [4]: r = rand(4, 4)

In [5]: Qobj(r)
Out[5]:
Quantum object: dims = [[4], [4]], shape = [4, 4], type = oper, isherm = False
Qobj data =
[[ 0.89383496  0.28955641  0.05983687  0.68182144]
 [ 0.31716633  0.82915623  0.30300061  0.52893312]
 [ 0.79004642  0.76768826  0.62795681  0.62051515]
 [ 0.44864855  0.6967143   0.20921074  0.71941593]]
```

Notice how both the dims and shape change according to the input data. Although dims and shape appear to have the same function, the difference will become quite clear in the section on tensor products and partial traces.

Note: If you are running QuTiP from a python script you must use the `print` function to view the Qobj attributes.

States and operators

Now, unless you have lots of free time, specifying the data for each object is inefficient. Even more so when most objects correspond to commonly used types such as the ladder operators of a harmonic oscillator, the Pauli spin operators for a two-level system, or state vectors such as Fock states. Therefore, QuTiP includes predefined objects for a variety of states:

States	Command (# means optional)	Inputs
Fock state ket vector	basis(N,#m) / fock(N,#m)	N = number of levels in Hilbert space, m = level containing excitation (0 if no m given)
Fock density matrix (outer product of basis)	fock_dm(N,#p)	same as basis(N,m) / fock(N,m)
Coherent state	coherent(N,alpha)	alpha = complex number (eigenvalue) for requested coherent state
Coherent density matrix (outer product)	coher-ent_dm(N,alpha)	same as coherent(N,alpha)
Thermal density matrix (for n particles)	thermal_dm(N,n)	n = particle number expectation value

and operators:

Operators	Command (# means optional)	Inputs
Identity	qeye(N)	N = number of levels in Hilbert space.
Lowering (destruction) operator	destroy(N)	same as above
Raising (creation) operator	create(N)	same as above
Number operator	num(N)	same as above
Single-mode displacement operator	displace(N,alpha)	N=number of levels in Hilbert space, alpha = complex displacement amplitude.
Single-mode squeezing operator	squeez(N,sp)	N=number of levels in Hilbert space, sp = squeezing parameter.
Sigma-X	sigmax()	
Sigma-Y	sigmay()	
Sigma-Z	sigmaz()	
Sigma plus	sigmap()	
Sigma minus	sigmam()	
Higher spin operators	jmat(j,#s)	j = integer or half-integer representing spin, s = ‘x’, ‘y’, ‘z’, ‘+’, or ‘-‘

As an example, we give the output for a few of these functions:

```
In [1]: basis(5,3)
Out[1]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 0.]
 [ 1.]
 [ 0.]]

In [2]: coherent(5,0.5-0.5j)
Out[2]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.77880170+0.j        ]
 [ 0.38939142-0.38939142j]
 [ 0.00000000-0.27545895j]
```

```
 [-0.07898617-0.07898617j]
 [-0.04314271+0.j        ]]

In [3]: destroy(4)
Out[3]:
Quantum object: dims = [[4], [4]], shape = [4, 4], type = oper, isherm = False
Qobj data =
[[ 0.          1.          0.          0.        ]
 [ 0.          0.          1.41421356  0.        ]
 [ 0.          0.          0.          1.73205081]
 [ 0.          0.          0.          0.        ]]

In [4]: sigmaz()
Out[4]:
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[ 1.  0.]
 [ 0. -1.]]

In [5]: jmat(5/2.0,'+')
Out[5]:
Quantum object: dims = [[6], [6]], shape = [6, 6], type = oper, isherm = False
Qobj data =
[[ 0.          2.23606798  0.          0.          0.          0.        ]
 [ 0.          0.          2.82842712  0.          0.          0.        ]
 [ 0.          0.          0.          3.          0.          0.        ]
 [ 0.          0.          0.          0.          2.82842712  0.        ]
 [ 0.          0.          0.          0.          0.          2.23606798]
 [ 0.          0.          0.          0.          0.          0.        ]]
```

Qobj attributes

We have seen that a quantum object has several internal attributes, such as data, dims, and shape. These can be accessed in the following way:

```
In [1]: q = destroy(4)

In [2]: q.dims
Out[2]: [[4], [4]]

In [3]: q.shape
Out[3]: [4, 4]
```

In general, the attributes (properties) of a Qobj object (or any Python class) can be retrieved using the *Q.attribute* notation. In addition to the attributes shown with the *print* function, the Qobj class also has the following:

Property	Attribute	Description
Data	Q.data	Matrix representing state or operator
Dimensions	Q.dims	List keeping track of shapes for individual components of a multipartite system (for tensor products and partial traces).
Shape	Q.shape	Dimensions of underlying data matrix.
is Hermitian?	Q.isherm	Is the operator Hermitian or not?
Type	Q.type	Is object of type 'ket, 'bra', 'oper', or 'super'?

For the destruction operator above:

```
In [1]: q.type
Out[1]: 'oper'

In [2]: q.isherm
Out[2]: False
```

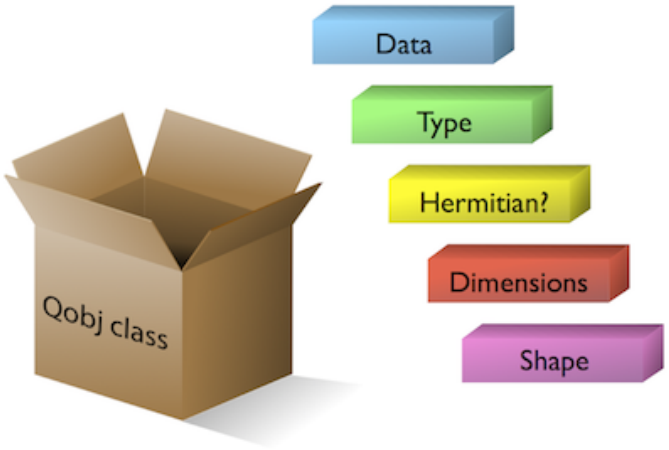


Figure 4.2: The *Qobj* Class viewed as a container for the properties need to characterize a quantum operator or state vector.

```
In [3]: q.data
Out[3]:
<4x4 sparse matrix of type '<type 'numpy.complex128'>'
        with 3 stored elements in Compressed Sparse Row format>
```

The data attribute returns a message stating that the data is a sparse matrix. All Qobjs store their data as a sparse matrix to save memory. To access the underlying matrix one needs to use the `qutip.Qobj.full` function as described in the functions section.

Qobj Math

The rules for mathematical operations on Qobj's are similar to standard matrix arithmetic:

```
In [1]: q = destroy(4)

In [2]: x = sigmax()

In [3]: q + 5
Out[3]:
Quantum object: dims = [[4], [4]], shape = [4, 4], type = oper, isherm = False
Qobj data =
[[ 5.          1.          0.          0.        ]
 [ 0.          5.          1.41421356  0.        ]
 [ 0.          0.          5.          1.73205081]
 [ 0.          0.          0.          5.        ]]

In [4]: x * x
Out[4]:
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[ 1.  0.]
 [ 0.  1.]]

In [5]: q ** 3
Out[5]:
Quantum object: dims = [[4], [4]], shape = [4, 4], type = oper, isherm = False
Qobj data =
[[ 0.          0.          0.          2.44948974]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.        ]]
```

```
In [6]: x / sqrt(2)
Out[6]:
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[ 0.          0.70710678]
 [ 0.70710678  0.        ]]
```

Of course, like matrices, multiplying two objects of incompatible shape throws an error:

```
>>> q * x
TypeError: Incompatible Qobj shapes
```

In addition, the logic operators is equal == and is not equal != are also supported.

4.2.3 Functions operating on Qobj class

Like attributes, the quantum object class has defined functions (methods) that operate on Qobj class instances. For a general quantum object Q:

Function	Command	Description
Conjugate	Q.conj()	Conjugate of quantum object.
Dagger (adjoint)	Q.dag()	Returns adjoint (dagger) of object.
Diagonal	Q.diag()	Returns the diagonal elements.
Eigenenergies	Q.eigenenergies()	Eigenenergies (values) of operator.
Eigenstates	Q.eigenstates()	Returns eigenvalues and eigenvectors.
Exponential	Q.expm()	Matrix exponential of operator.
Full	Q.full()	Returns full (not sparse) array of Q's data.
Groundstate	Q.groundstate()	Eigenval & eigket of Qobj groundstate.
Matrix Element	Q.matrix_element(bra,ket)	Matrix element <bra|Q|ket>
Norm	Q.norm()	Returns L2 norm for states, trace norm for operators.
Partial Trace	Q.ptrace(sel)	Partial trace returning components selected using 'sel' parameter.
Sqrt	Q.sqrtm()	Matrix sqrt of operator.
Tidyup	Q.tidyup()	Removes small elements from Qobj.
Trace	Q.tr()	Returns trace of quantum object.
Transform	Q.transform(inpt)	A basis transformation defined by matrix or list of kets 'inpt' .
Transpose	Q.trans()	Transpose of quantum object.
Unit	Q.unit()	Returns normalized (unit) vector Q/Q.norm().

```
In [1]: basis(5, 3)
Out[1]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 0.]
 [ 1.]
 [ 0.]]

In [2]: basis(5, 3).dag()
Out[2]:
Quantum object: dims = [[1], [5]], shape = [1, 5], type = bra
Qobj data =
[[ 0.  0.  0.  1.  0.]]

In [3]: coherent_dm(5, 1)
Out[3]:
Quantum object: dims = [[5], [5]], shape = [5, 5], type = oper, isherm = True
Qobj data =
[[ 0.36791117  0.36774407  0.26105441  0.14620658  0.08826704]
 [ 0.36774407  0.36757705  0.26093584  0.14614018  0.08822695]
```

```
 [ 0.26105441  0.26093584  0.18523331  0.10374209  0.06263061]
 [ 0.14620658  0.14614018  0.10374209  0.05810197  0.035077  ]
 [ 0.08826704  0.08822695  0.06263061  0.035077    0.0211765 ]]

In [4]: coherent_dm(5, 1).diag()
Out[4]: array([ 0.36791117,  0.36757705,  0.18523331,  0.05810197,  0.0211765 ])

In [5]: coherent_dm(5, 1).full()
Out[5]:
array([[ 0.36791117+0.j,  0.36774407+0.j,  0.26105441+0.j,  0.14620658+0.j,
         0.08826704+0.j],
       [ 0.36774407+0.j,  0.36757705+0.j,  0.26093584+0.j,  0.14614018+0.j,
         0.08822695+0.j],
       [ 0.26105441+0.j,  0.26093584+0.j,  0.18523331+0.j,  0.10374209+0.j,
         0.06263061+0.j],
       [ 0.14620658+0.j,  0.14614018+0.j,  0.10374209+0.j,  0.05810197+0.j,
         0.03507700+0.j],
       [ 0.08826704+0.j,  0.08822695+0.j,  0.06263061+0.j,  0.03507700+0.j,
         0.02117650+0.j]])

In [6]: coherent_dm(5, 1).norm()
Out[6]: 1.0

In [7]: coherent_dm(5, 1).sqrtm()
Out[7]:
Quantum object: dims = [[5], [5]], shape = [5, 5], type = oper, isherm = True
Qobj data =
[[ 0.36791118  0.36774407  0.26105441  0.14620658  0.08826704]
 [ 0.36774407  0.36757705  0.26093584  0.14614018  0.08822695]
 [ 0.26105441  0.26093584  0.18523331  0.10374209  0.06263061]
 [ 0.14620658  0.14614018  0.10374209  0.05810197  0.035077  ]
 [ 0.08826704  0.08822695  0.06263061  0.035077    0.0211765 ]]

In [8]: coherent_dm(5, 1).tr()
Out[8]: 1.0

In [9]: (basis(4, 2) + basis(4, 1)).unit()
Out[9]:
Quantum object: dims = [[4], [1]], shape = [4, 1], type = ket
Qobj data =
[[ 0.        ]
 [ 0.70710678]
 [ 0.70710678]
 [ 0.        ]]
```

4.3 Manipulating States and Operators

4.3.1 Introduction

In the previous guide section *Basic Operations on Quantum Objects*, we saw how to create operators and states, using the functions built into QuTiP. In this portion of the guide, we will look at performing basic operations with states and operators. For more detailed demonstrations on how to use and manipulate these objects, see the *QuTiP Example Scripts* section.

4.3.2 State Vectors (kets or bras)

Here we begin by creating a Fock `qutip.states.basis` vacuum state vector $|0\rangle$ with in a Hilbert space with 5 number states, from 0 to 4:

```
In [1]: vec = basis(5, 0)

In [2]: print(vec)
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 1.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]]
```

and then create a lowering operator ($\hat{a}$) corresponding to 5 number states using the `qutip.operators.destroy` function:

```
In [1]: a = destroy(5)

In [2]: print(a)
Quantum object: dims = [[5], [5]], shape = [5, 5], type = oper, isherm = False
Qobj data =
[[ 0.          1.          0.          0.          0.        ]
 [ 0.          0.          1.41421356  0.          0.        ]
 [ 0.          0.          0.          1.73205081  0.        ]
 [ 0.          0.          0.          0.          2.        ]
 [ 0.          0.          0.          0.          0.        ]]
```

Now lets apply the destruction operator to our vacuum state `vec`,

```
In [1]: a * vec
Out[1]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]]
```

We see that, as expected, the vacuum is transformed to the zero vector. A more interesting example comes from using the adjoint of the lowering operator, the raising operator $\hat{a}^\dagger$:

```
In [1]: a.dag() * vec
Out[1]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.]
 [ 1.]
 [ 0.]
 [ 0.]
 [ 0.]]
```

The raising operator has in indeed raised the state *vec* from the vacuum to the $|1\rangle$ state. Instead of using the dagger `Qobj.dag()` method to raise the state, we could have also used the built in `qutip.operators.create` function to make a raising operator:

```
In [1]: c = create(5)

In [2]: c * vec
Out[2]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.]
 [ 1.]
 [ 0.]
```

```
 [ 0.]
 [ 0.]]
```

which obviously does the same thing. We can of course raise the vacuum state more than once:

```
In [1]: c * c * vec
Out[1]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.        ]
 [ 0.        ]
 [ 1.41421356]
 [ 0.        ]
 [ 0.        ]]
```

or just taking the square of the raising operator $\left(\hat{a}^\dagger\right)^2$:

```
In [1]: c**2 * vec
Out[1]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.        ]
 [ 0.        ]
 [ 1.41421356]
 [ 0.        ]
 [ 0.        ]]
```

Applying the raising operator twice gives the expected $\sqrt{n+1}$ dependence. We can use the product of $c * a$ to also apply the number operator to the state vector `vec`:

```
In [1]: c * a * vec
Out[1]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]]
```

or on the $|1\rangle$ state:

```
In [1]: c * a * (c * vec)
Out[1]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.]
 [ 1.]
 [ 0.]
 [ 0.]
 [ 0.]]
```

or the $|2\rangle$ state:

```
In [1]: c * a * (c**2 * vec)
Out[1]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.        ]
 [ 0.        ]
 [ 2.82842712]
 [ 0.        ]
 [ 0.        ]]
```

Notice how in this last example, application of the number operator does not give the expected value $n = 2$, but rather $2\sqrt{2}$. This is because this last state is not normalized to unity as $c|n\rangle = \sqrt{n+1}|n+1\rangle$. Therefore, we should normalize our vector first:

```
In [1]: c * a * (c**2 * vec).unit()
Out[1]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 2.]
 [ 0.]
 [ 0.]]
```

Since we are giving a demonstration of using states and operators, we have done a lot more work than we should have. For example, we do not need to operate on the vacuum state to generate a higher number Fock state. Instead we can use the `qutip.states.basis` (or `qutip.states.fock`) function to directly obtain the required state:

```
In [1]: vec = basis(5, 2)

In [2]: print(vec)
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 1.]
 [ 0.]
 [ 0.]]
```

Notice how it is automatically normalized. We can also use the built in `qutip.operators.num` operator:

```
In [1]: n = num(5)

In [2]: print(n)
Quantum object: dims = [[5], [5]], shape = [5, 5], type = oper, isherm = True
Qobj data =
[[ 0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.]
 [ 0.  0.  2.  0.  0.]
 [ 0.  0.  0.  3.  0.]
 [ 0.  0.  0.  0.  4.]]
```

Therefore, instead of `c * a * (c**2 * vec).unit()` we have:

```
In [1]: n * vec
Out[1]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 2.]
 [ 0.]
 [ 0.]]
```

We can also create superpositions of states:

```
In [1]: vec = (basis(5, 0) + basis(5, 1)).unit()

In [2]: print(vec)
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.70710678]
 [ 0.70710678]
```

```
 [ 0.        ]
 [ 0.        ]
 [ 0.        ]]
```

where we have used the `qutip.Qobj.unit` method to again normalize the state. Operating with the number function again:

```
In [1]: n * vec
Out[1]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.        ]
 [ 0.70710678]
 [ 0.        ]
 [ 0.        ]
 [ 0.        ]]
```

We can also create coherent states and squeezed states by applying the `qutip.operators.displace` and `qutip.operators.squeez` functions to the vacuum state:

```
In [1]: vec = basis(5, 0)

In [2]: d = displace(5, 1j)

In [3]: s = squeez(5, 0.25 + 0.25j)

In [4]: d * vec
Out[4]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.60655682+0.j        ]
 [ 0.00000000+0.60628133j]
 [-0.43038740+0.j        ]
 [ 0.00000000-0.24104351j]
 [ 0.14552147+0.j        ]]

In [1]: d * s * vec
Out[1]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.65893786+0.08139381j]
 [ 0.10779462+0.51579735j]
 [-0.37567217-0.01326853j]
 [-0.02688063-0.23828775j]
 [ 0.26352814+0.11512178j]]
```

Of course, displacing the vacuum gives a coherent state, which can also be generated using the built in `qutip.states.coherent` function.

4.3.3 Density matrices

One of the main purpose of QuTiP is to explore the dynamics of **open** quantum systems, where the most general state of a system is not longer a state vector, but rather a density matrix. Since operations on density matrices operate identically to those of vectors, we will just briefly highlight creating and using these structures.

The simplest density matrix is created by forming the outer-product $|\psi\rangle\langle\psi|$ of a ket vector:

```
In [1]: vec = basis(5, 2)

In [2]: vec * vec.dag()
Out[2]:
Quantum object: dims = [[5], [5]], shape = [5, 5], type = oper, isherm = True
```

```
Qobj data =
[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]
```

A similar task can also be accomplished via the `qutip.states.fock_dm` or `qutip.states.ket2dm` functions:

```
In [1]: fock_dm(5, 2)
Out[1]:
Quantum object: dims = [[5], [5]], shape = [5, 5], type = oper, isherm = True
Qobj data =
[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]

In [1]: ket2dm(vec)
Out[1]:
Quantum object: dims = [[5], [5]], shape = [5, 5], type = oper, isherm = True
Qobj data =
[[ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]
```

If we want to create a density matrix with equal classical probability of being found in the $|2\rangle$ or $|4\rangle$ number states we can do the following:

```
In [1]: 0.5 * ket2dm(basis(5, 4)) + 0.5 * ket2dm(basis(5, 2))
Out[1]:
Quantum object: dims = [[5], [5]], shape = [5, 5], type = oper, isherm = True
Qobj data =
[[ 0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0. ]
 [ 0.   0.   0.5  0.   0. ]
 [ 0.   0.   0.   0.   0. ]
 [ 0.   0.   0.   0.   0.5]]
```

or use `0.5 * fock_dm(5, 2) + 0.5 * fock_dm(5, 4)`. There are also several other built-in functions for creating predefined density matrices, for example `qutip.states.coherent_dm` and `qutip.states.thermal_dm` which create coherent state and thermal state density matrices, respectively.

```
In [1]: coherent_dm(5, 1.25)
Out[1]:
Quantum object: dims = [[5], [5]], shape = [5, 5], type = oper, isherm = True
Qobj data =
[[ 0.20980701  0.26141096  0.23509686  0.15572585  0.13390765]
 [ 0.26141096  0.32570738  0.29292109  0.19402805  0.16684347]
 [ 0.23509686  0.29292109  0.26343512  0.17449684  0.1500487 ]
 [ 0.15572585  0.19402805  0.17449684  0.11558499  0.09939079]
 [ 0.13390765  0.16684347  0.1500487   0.09939079  0.0854655 ]]

In [1]: thermal_dm(5, 1.25)
Out[1]:
Quantum object: dims = [[5], [5]], shape = [5, 5], type = oper, isherm = True
Qobj data =
[[ 0.46927974  0.          0.          0.          0.        ]
 [ 0.          0.26071096  0.          0.          0.        ]
```

```
 [ 0.          0.          0.14483942  0.          0.        ]
 [ 0.          0.          0.          0.08046635  0.        ]
 [ 0.          0.          0.          0.          0.04470353]]
```

QuTiP also provides a set of distance metrics for determining how close two density matrix distributions are to each other. Included are the trace distance `qutip.metrics.tracedist` and the fidelity `qutip.metrics.fidelity`.

```
In [1]: x = coherent_dm(5, 1.25)

In [2]: y = coherent_dm(5, 1.25j)  # <-- note the 'j'

In [3]: z = thermal_dm(5, 0.125)

In [4]: fidelity(x, x)
Out[4]: 1.0000000288041344

In [5]: tracedist(y, y)
Out[5]: 0.0
```

We also know that for two pure states, the trace distance (T) and the fidelity (F) are related by $T = \sqrt{1 - F^2}$.

```
In [1]: tracedist(y, x)
Out[1]: 0.9771565856691513

In [1]: sqrt(1 - fidelity(y, x) ** 2)
Out[1]: 0.97715656913682691
```

For a pure state and a mixed state, $1 - F^2 \leq T$ which can also be verified:

```
In [1]: 1 - fidelity(x,z) ** 2
Out[1]: 0.7782890444194828

In [1]: tracedist(x, z)
Out[1]: 0.8559028328862588
```

4.3.4 Qubit (two-level) systems

Having spent a fair amount of time on basis states that represent harmonic oscillator states, we now move on to qubit, or two-level quantum systems (for example a spin-1/2). To create a state vector corresponding to a qubit system, we use the same `qutip.states.basis`, or `qutip.states.fock`, function with only two levels:

```
In [1]: spin = basis(2, 0)
```

Now at this point one may ask how this state is different than that of a harmonic oscillator in the vacuum state truncated to two energy levels?

```
In [1]: vec = basis(2, 0)
```

At this stage, there is no difference. This should not be surprising as we called the exact same function twice. The difference between the two comes from the action of the spin operators `qutip.operators.sigmax`, `qutip.operators.sigmay`, `qutip.operators.sigmaz`, `qutip.operators.sigmap`, and `qutip.operators.sigmam` on these two-level states. For example, if `vec` corresponds to the vacuum state of a harmonic oscillator, then, as we have already seen, we can use the raising operator to get the $|1\rangle$ state:

```
In [1]: vec
Out[1]:
Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[ 1.]
 [ 0.]]
```

```
In [1]: c = create(2)

In [2]: c * vec
Out[2]:
Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[ 0.]
 [ 1.]]
```

For a spin system, the operator analogous to the raising operator is the sigma-plus operator `qutip.operators.sigmap`. Operating on the `spin` state gives:

```
In [1]: spin
Out[1]:
Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[ 1.]
 [ 0.]]

In [2]: sigmap() * spin
Out[2]:
Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[ 0.]
 [ 0.]]
```

Now we see the difference! The `qutip.operators.sigmap` operator acting on the `spin` state returns the zero vector. Why is this? To see what happened, let us use the `qutip.operators.sigmaz` operator:

```
In [1]: sigmaz()
Out[1]:
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[ 1.  0.]
 [ 0. -1.]]

In [2]: sigmaz() * spin
Out[2]:
Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[ 1.]
 [ 0.]]

In [3]: spin2 = basis(2, 1)

In [4]: spin2
Out[4]:
Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[ 0.]
 [ 1.]]

In [5]: sigmaz() * spin2
Out[5]:
Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[ 0.]
 [-1.]]
```

The answer is now apparent. Since the QuTiP `qutip.operators.sigmaz` function uses the standard z-basis representation of the sigma-z spin operator, the `spin` state corresponds to the $|\mathrm{up}\rangle$ state of a two-level spin system while `spin2` gives the $|\mathrm{down}\rangle$ state. Therefore, in our previous example `sigmap()*spin`, we raised the qubit state out of the truncated two-level Hilbert space resulting in the zero state.

While at first glance this convention might seem somewhat odd, it is in fact quite handy. For one, the spin operators remain in the conventional form. Second, when the spin system is in the $|\mathrm{up}\rangle$ state:

```
In [1]: sigmaz() * spin
Out[1]:
Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[ 1.]
 [ 0.]]
```

the non-zero component is the zeroth-element of the underlying matrix (remember that python uses c-indexing, and matrices start with the zeroth element). The $|\mathrm{down}\rangle$ state therefore has a non-zero entry in the first index position. This corresponds nicely with the quantum information definitions of qubit states, where the excited $|\mathrm{up}\rangle$ state is label as $|0\rangle$, and the $|\mathrm{down}\rangle$ state by $|1\rangle$.

If one wants to create spin operators for higher spin systems, then the `qutip.operators.jmat` function comes in handy.

4.3.5 Expectation values

Some of the most important information about quantum systems comes from calculating the expectation value of operators, both Hermitian and non-Hermitian, as the state or density matrix of the system varies in time. Therefore, in this section we demonstrate the use of the `qutip.expect` function. To begin:

```
In [1]: vac = basis(5, 0)

In [2]: one = basis(5, 1)

In [3]: c = create(5)

In [4]: N = num(5)

In [5]: expect(N, vac)
Out[5]: 0.0

In [6]: expect(N, one)
Out[6]: 1.0

In [1]: coh = coherent_dm(5, 1.0j)

In [2]: expect(N, coh)
Out[2]: 0.9970555745806599

In [1]: cat = (basis(5, 4) + 1.0j * basis(5, 3)).unit()

In [2]: expect(c, cat)
Out[2]: 0.9999999999999978j
```

The `qutip.expect` function also accepts lists or arrays of state vectors or density matrices for the second input:

```
In [1]: states = [(c**k * vac).unit() for k in range(5)]  # must normalize

In [2]: expect(N, states)
Out[2]: array([ 0.,  1.,  2.,  3.,  4.])

In [1]: cat_list = [(basis(5, 4) + x * basis(5, 3)).unit() for x in [0, 1.0j, -1.0, -1.0j]]

In [2]: expect(c, cat_list)
Out[2]: array([ 0.+0.j,  0.+1.j, -1.+0.j,  0.-1.j])
```

Notice how in this last example, all of the return values are complex numbers. This is because the `qutip.expect` function looks to see whether the operator is Hermitian or not. If the operator is Hermitian,

than the output will always be real. In the case of non-Hermitian operators, the return values may be complex. Therefore, the `qutip.expect` function will return an array of complex values for non-Hermitian operators when the input is a list/array of states or density matrices.

Of course, the `qutip.expect` function works for spin states and operators:

```
In [1]: up = basis(2, 0)

In [2]: down = basis(2, 1)

In [3]: expect(sigmaz(), up)
Out[3]: 1.0

In [4]: expect(sigmaz(), down)
Out[4]: -1.0
```

as well as the composite objects discussed in the next section *Using Tensor Products and Partial Traces*:

```
In [1]: spin1 = basis(2, 0)

In [2]: spin2 = basis(2, 1)

In [3]: two_spins = tensor(spin1, spin2)

In [4]: sz1 = tensor(sigmaz(), qeye(2))

In [5]: sz2 = tensor(qeye(2), sigmaz())

In [6]: expect(sz1, two_spins)
Out[6]: 1.0

In [7]: expect(sz2, two_spins)
Out[7]: -1.0
```

4.4 Using Tensor Products and Partial Traces

4.4.1 Tensor products

To describe the states of multipartite quantum systems - such as two coupled qubits, a qubit coupled to an oscillator, etc. - we need to expand the Hilbert space by taking the tensor product of the state vectors for each of the system components. Similarly, the operators acting on the state vectors in the combined Hilbert space (describing the coupled system) are formed by taking the tensor product of the individual operators.

In QuTiP the function `qutip.tensor.tensor` is used to accomplish this task. This function takes as argument a collection:

```
>>> tensor(op1, op2, op3)
```

or a `list`:

```
>>> tensor([op1, op2, op3])
```

of state vectors *or* operators and returns a composite quantum object for the combined Hilbert space. The function accepts an arbitray number of states or operators as argument. The type returned quantum object is the same as that of the input(s).

For example, the state vector describing two qubits in their ground states is formed by taking the tensor product of the two single-qubit ground state vectors:

```
In [1]: tensor(basis(2, 0), basis(2, 0))
Out[1]:
Quantum object: dims = [[2, 2], [1, 1]], shape = [4, 1], type = ket
```

```
Qobj data =
[[ 1.]
 [ 0.]
 [ 0.]
 [ 0.]]
```

or equivalently using the `list` format:

```
In [1]: tensor([basis(2, 0), basis(2, 0)])
Out[1]:
Quantum object: dims = [[2, 2], [1, 1]], shape = [4, 1], type = ket
Qobj data =
[[ 1.]
 [ 0.]
 [ 0.]
 [ 0.]]
```

This is straightforward to generalize to more qubits by adding more component state vectors in the argument list to the `qutip.tensor.tensor` function, as illustrated in the following example:

```
In [1]: tensor((basis(2, 0) + basis(2, 1)).unit(), (basis(2, 0) + basis(2, 1)).unit(), basis(2, 0
Out[1]:
Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = [8, 1], type = ket
Qobj data =
[[ 0.5]
 [ 0. ]
 [ 0.5]
 [ 0. ]
 [ 0.5]
 [ 0. ]
 [ 0.5]
 [ 0. ]]
```

This state is slightly more complicated, describing two qubits in a superposition between the up and down states, while the third qubit is in its ground state.

To construct operators that act on an extended Hilbert space of a combined system, we similarly pass a list of operators for each component system to the `qutip.tensor.tensor` function. For example, to form the operator that represents the simultaneous action of the σ_x operator on two qubits:

```
In [1]: tensor(sigmax(), sigmax())
Out[1]:
Quantum object: dims = [[2, 2], [2, 2]], shape = [4, 4], type = oper, isherm = True
Qobj data =
[[ 0.  0.  0.  1.]
 [ 0.  0.  1.  0.]
 [ 0.  1.  0.  0.]
 [ 1.  0.  0.  0.]]
```

To create operators in a combined Hilbert space that only act only on a single component, we take the tensor product of the operator acting on the subspace of interest, with the identity operators corresponding to the components that are to be unchanged. For example, the operator that represents σ_z on the first qubit in a two-qubit system, while leaving the second qubit unaffected:

```
In [1]: tensor(sigmaz(), qeye(2))
Out[1]:
Quantum object: dims = [[2, 2], [2, 2]], shape = [4, 4], type = oper, isherm = True
Qobj data =
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0. -1.  0.]
 [ 0.  0.  0. -1.]]
```

4.4.2 Example: Constructing composite Hamiltonians

The `qutip.tensor.tensor` function is extensively used when constructing Hamiltonians for composite systems. Here we'll look at some simple examples.

Two coupled qubits

First, let's consider a system of two coupled qubits. Assume that both qubit has equal energy splitting, and that the qubits are coupled through a $\sigma_x \otimes \sigma_x$ interaction with strength g = 0.05 (in units where the bare qubit energy splitting is unity). The Hamiltonian describing this system is:

```
In [1]: H = tensor(sigmaz(), qeye(2)) + tensor(qeye(2), sigmaz()) + 0.05 * tensor(sigmax(), sigmax
```

```
In [2]: H
Out[2]:
Quantum object: dims = [[2, 2], [2, 2]], shape = [4, 4], type = oper, isherm = True
Qobj data =
[[ 2.    0.    0.    0.05]
 [ 0.    0.    0.05  0.  ]
 [ 0.    0.05  0.    0.  ]
 [ 0.05  0.    0.   -2.  ]]
```

Three coupled qubits

The two-qubit example is easily generalized to three coupled qubits:

```
In [1]: H = tensor(sigmaz(), qeye(2), qeye(2)) + tensor(qeye(2), sigmaz(), qeye(2)) + tensor(qeye
```

```
In [2]: H
Out[2]:
Quantum object: dims = [[2, 2, 2], [2, 2, 2]], shape = [8, 8], type = oper, isherm = True
Qobj data =
[[ 3.    0.    0.    0.25  0.    0.    0.5   0.  ]
 [ 0.    1.    0.25  0.    0.    0.    0.    0.5 ]
 [ 0.    0.25  1.    0.    0.5   0.    0.    0.  ]
 [ 0.25  0.    0.   -1.    0.    0.5   0.    0.  ]
 [ 0.    0.    0.5   0.    1.    0.    0.    0.25]
 [ 0.    0.    0.    0.5   0.   -1.    0.25  0.  ]
 [ 0.5   0.    0.    0.    0.    0.25 -1.    0.  ]
 [ 0.    0.5   0.    0.    0.25  0.    0.   -3.  ]]
```

A two-level system coupled to a cavity: The Jaynes-Cummings model

The simplest possible quantum mechanical description for light-matter interaction is encapsulated in the Jaynes-Cummings model, which describes the coupling between a two-level atom and a single-mode electromagnetic field (a cavity mode). Denoting the energy splitting of the atom and cavity `omega_a` and `omega_c`, respectively, and the atom-cavity interaction strength `g`, the Jaynes-Cumming Hamiltonian can be constructed as:

```
>>> N = 10
>>> omega_a = 1.0
>>> omega_c = 1.25
>>> g = 0.05
>>> a = tensor(qeye(2), destroy(N))
>>> sm = tensor(destroy(2), qeye(N))
>>> sz = tensor(sigmaz(), qeye(N))
>>> H = 0.5 * omega_a * sz + omega_c * a.dag() * a + g * (a.dag() * sm + a * sm.dag())
```

Here `N` is the number of Fock states included in the cavity mode.

4.4.3 Partial trace

The partial trace is an operation that reduces the dimension of a Hilbert space by eliminating some degrees of freedom by averaging (tracing). In this sense it is therefore the converse of the tensor product. It is useful when one is interested in only a part of a coupled quantum system. For open quantum systems, this typically involves tracing over the environment leaving only the system of interest. In QuTiP the class method `qutip.Qobj.ptrace` is used to take partial traces. `qutip.Qobj.ptrace` acts on the `qutip.Qobj` instance for which it is called, and it takes one argument `sel`, which is a `list` of integers that mark the component systems that should be **kept**. All other components are traced out.

For example, the density matrix describing a single qubit obtained from a coupled two-qubit system is obtained via:

```
In [1]: psi = tensor(basis(2, 0), basis(2, 1))

In [2]: psi.ptrace(0)
Out[2]:
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[ 1.  0.]
 [ 0.  0.]]

In [3]: psi.ptrace(1)
Out[3]:
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[ 0.  0.]
 [ 0.  1.]]
```

Note that the partial trace always results in a density matrix (mixed state), regardless of whether the composite system is a pure state (described by a state vector) or a mixed state (described by a density matrix):

```
In [1]: psi = tensor((basis(2, 0) + basis(2, 1)).unit(), basis(2, 0))

In [2]: psi
Out[2]:
Quantum object: dims = [[2, 2], [1, 1]], shape = [4, 1], type = ket
Qobj data =
[[ 0.70710678]
 [ 0.        ]
 [ 0.70710678]
 [ 0.        ]]

In [3]: psi.ptrace(0)
Out[3]:
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[ 0.5  0.5]
 [ 0.5  0.5]]

In [4]: rho = tensor(ket2dm((basis(2, 0) + basis(2, 1)).unit()), fock_dm(2, 0))

In [5]: rho
Out[5]:
Quantum object: dims = [[2, 2], [2, 2]], shape = [4, 4], type = oper, isherm = True
Qobj data =
[[ 0.5  0.   0.5  0. ]
 [ 0.   0.   0.   0. ]
 [ 0.5  0.   0.5  0. ]
 [ 0.   0.   0.   0. ]]

In [6]: rho.ptrace(0)
Out[6]:
```

```
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[ 0.5  0.5]
 [ 0.5  0.5]]
```

4.5 Time Evolution and Quantum System Dynamics

4.5.1 The Odedata Class and Dynamical Simulation Results

Important: In QuTiP 2, the results from all of the dynamics solvers are returned as Odedata objects. This significantly simplifies the storage and saving of simulation data. However, this change also results in the loss of backward compatibility with QuTiP version 1.x. Therefore, please read this section to avoid running into any issues.

The Odedata Class

Before embarking on simulating the dynamics of quantum systems, we will first look at the data structure used for returning the simulation results to the user. This object is a `qutip.Odedata` class that stores all the crucial data needed for analyzing and plotting the results of a simulation. Like the `qutip.Qobj` class, the `Odedata` class has a collection of properties for storing information. However, in contrast to the Qobj class, this structure contains no methods, and is therefore nothing but a container object. A generic Odedata object `odedata` contains the following properties for storing simulation data:

Property	Description
odedata.solver	String indicating which solver was used to generate the data.
odedata.times	List/array of times at which simulation data is calculated.
odedata.expect	List/array of expectation values, if requested.
odedata.states	List/array of state vectors/density matrices calculated at `times`, if requested.
odedata.num_expect	The number of expectation value operators in the simulation.
odedata.num_collapse	The number of collapse operators in the simulation.
odedata.ntraj	Number of Monte Carlo trajectories run.
odedata.col_times	Times at which state collapse occurred. Only for Monte Carlo solver.
odedata.col_which	Which collapse operator was responsible for each collapse in in `col_times`. Only used by Monte Carlo solver.

Accessing Odedata Data

To understand how to access the data in a Odedata object we will use the *Driven Cavity+Qubit Monte Carlo* example as a guide, although we do not worry about the simulation details at this stage. Like all solvers, the Monte Carlo solver used in this example returns an Odedata object, here called simply `data`. To see what is contained inside `data` we can use the print function:

```
>>> print(data)
    Odedata object with mcsolve data.
    ---------------------------------
    expect = True
    num_expect = 2, num_collapse = 2, ntraj = 500
```

The first line tells us that this data object was generated from the Monte Carlo solver `mcsolve` (discussed in *Quantum Dynamics via the Monte Carlo Solver*). The next line (not the `---` line of course) indicates that this object contains expectation value data. Finally, the last line gives the number of expectation value and collapse operators used in the simulation, along with the number of Monte Carlo trajectories run. Note that the number of trajectories `ntraj` is only displayed when using the Monte Carlo solver.

Now we have all the information needed to analyze the simulation results. To access the data for the two expectation values one can do:

```
>>> expt0 = data.expect[0]
>>> expt1 = data.expect[1]
```

Recall that Python uses C-style indexing that begins with zero (i.e. [0] => 1st collapse operator data). Together with the array of times at which these expectation values are calculated:

```
>>> times = data.times
```

we can plot the resulting expectation values:

```
>>> plot(times, expt0, times, expt1)
>>> show()
```

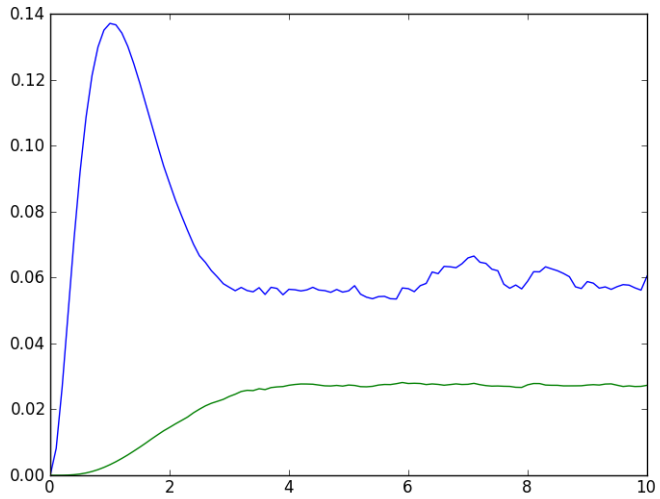


Figure 4.3: Data for expectation values extracted from the `data` Odedata object.

State vectors, or density matrices, as well as `col_times` and `col_which`, are accessed in a similar manner, although typically one does not need an index (i.e [0]) since there is only one list for each of these components. The one exception to this rule is if you choose to output state vectors from the Monte Carlo solver, in which case there are `ntraj` number of state vector arrays.

Saving and Loading Odedata Objects

The main advantage in using the Odedata class as a data storage object comes from the simplicity in which simulation data can be stored and later retrieved. The `qutip.fileio.qsave` and `qutip.fileio.qload` functions are designed for this task. To begin, let us save the `data` object from the previous section into a file called "cavity+qubit-data" in the current working directory by calling:

```
>>> qsave(data, 'cavity+qubit-data')
```

All of the data results are then stored in a single file of the same name with a ".qu" extension. Therefore, everything needed to later this data is stored in a single file. Loading the file is just as easy as saving:

```
>>> chicken = qload('cavity+qubit-data')
    Loaded Odedata object:
    Odedata object with mcsolve data.
    ---------------------------------
    expect = True
    num_expect = 2, num_collapse = 2, ntraj = 500
```

where `chicken` is the new name of the Odedata object. We can then extract the data and plot in the same manner as before:

```
expt0 = chicken.expect[0]
expt1 = chicken.expect[1]
times = chicken.times
plot(times, expt0, times, expt1)
show()
```

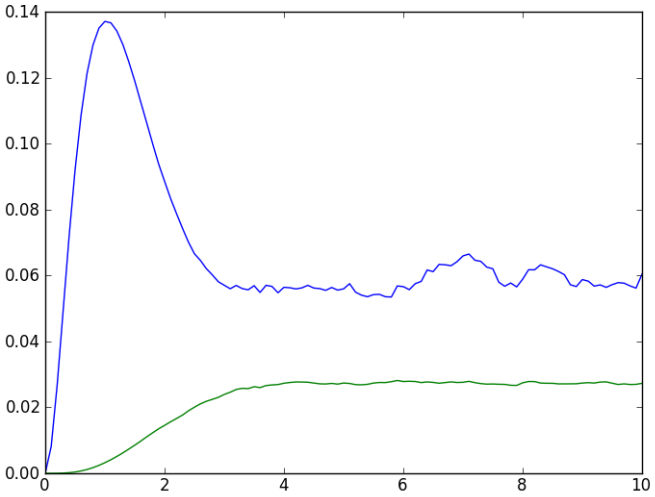


Figure 4.4: Data for expectation values from the `chicken` object loaded from the `data` object stored with `qutip.fileio.qsave`

Also see *Saving QuTiP Objects and Data Sets* for more information on saving quantum objects, as well as arrays for use in other programs.

4.5.2 Lindblad Master Equation Solver

Unitary evolution

The dynamics of a closed (pure) quantum system is governed by the Schrödinger equation

$$i\hbar\frac{\partial}{\partial t}\Psi = \hat{H}\Psi, \tag{4.1}$$

where Ψ is the wave function, $\hat{H}$ the Hamiltonian, and $\hbar$ is Planck's constant. In general, the Schrödinger equation is a partial differential equation (PDE) where both Ψ and $\hat{H}$ are functions of space and time. For computational purposes it is useful to expand the PDE in a set of basis functions that span the Hilbert space of the Hamiltonian, and to write the equation in matrix and vector form

$$i\hbar\frac{d}{dt}\left|\psi\right\rangle = H\left|\psi\right\rangle$$

where $\left|\psi\right\rangle$ is the state vector and H is the matrix representation of the Hamiltonian. This matrix equation can, in principle, be solved by diagonalizing the Hamiltonian matrix H. In practice, however, it is difficult to perform this diagonalization unless the size of the Hilbert space (dimension of the matrix H) is small. Analytically, it is a formidable task to calculate the dynamics for systems with more than two states. If, in addition, we consider dissipation due to the inevitable interaction with a surrounding environment, the computational complexity grows even larger, and we have to resort to numerical calculations in all realistic situations. This illustrates the importance of numerical calculations in describing the dynamics of open quantum systems, and the need for efficient and accessible tools for this task.

The Schrödinger equation, which governs the time-evolution of closed quantum systems, is defined by its Hamiltonian and state vector. In the previous section, *Using Tensor Products and Partial Traces*, we showed how Hamiltonians and state vectors are constructed in QuTiP. Given a Hamiltonian, we can calculate the unitary (non-dissipative) time-evolution of an arbitrary state vector $|\psi_0\rangle$ (`psi0`) using the QuTiP function `qutip.mesolve`. It evolves the state vector and evaluates the expectation values for a set of operators `expt_op_list` at the points in time in the list `tlist`, using an ordinary differential equation solver. Alternatively, we can use the function `qutip.essolve`, which uses the exponential-series technique to calculate the time evolution of a system. The `qutip.mesolve` and `qutip.essolve` functions take the same arguments and it is therefore easy switch between the two solvers.

For example, the time evolution of a quantum spin-1/2 system with tunneling rate 0.1 that initially is in the up state is calculated, and the expectation values of the σ_z operator evaluated, with the following code:

```
>>> H = 2 * pi * 0.1 * sigmax()
>>> psi0 = basis(2, 0)
>>> tlist = linspace(0.0, 10.0, 20.0)
>>> result = mesolve(H, psi0, tlist, [], [sigmaz()])
>>> result
Odedata object with mesolve data.
---------------------------------
expect = True
num_expect = 1, num_collapse = 0
>>> result.expect[0]
array([ 1.00000000+0.j,  0.78914229+0.j,  0.24548596+0.j, -0.40169696+0.j,
       -0.87947669+0.j, -0.98636356+0.j, -0.67728166+0.j, -0.08257676+0.j,
        0.54695235+0.j,  0.94582040+0.j,  0.94581706+0.j,  0.54694422+0.j,
       -0.08258520+0.j, -0.67728673+0.j, -0.98636329+0.j, -0.87947111+0.j,
       -0.40168898+0.j,  0.24549302+0.j,  0.78914528+0.j,  0.99999927+0.j])
```

The brackets in the fourth argument is an empty list of collapse operators, since we consider unitary evolution in this example. See the next section for examples on how dissipation is included by defining a list of collapse operators.

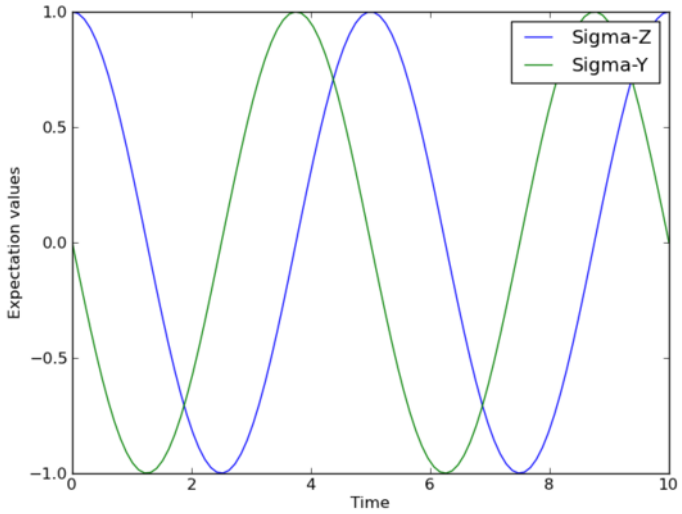
The function returns an instance of `qutip.Odedata`, as described in the previous section *The Odedata Class and Dynamical Simulation Results*. The attribute `expect` in `result` is a list of expectation values for the operators that are included in the list in the fifth argument. Adding operators to this list results in a larger output list returned by the function (one array of numbers, corresponding to the times in tlist, for each operator):

```
>>> result = mesolve(H, psi0, tlist, [], [sigmaz(), sigmay()])
>>> result.expect
[array([  1.00000000e+00+0.j,   7.89142292e-01+0.j,   2.45485961e-01+0.j,
         -4.01696962e-01+0.j,  -8.79476686e-01+0.j,  -9.86363558e-01+0.j,
         -6.77281655e-01+0.j,  -8.25767574e-02+0.j,   5.46952346e-01+0.j,
          9.45820404e-01+0.j,   9.45817056e-01+0.j,   5.46944216e-01+0.j,
         -8.25852032e-02+0.j,  -6.77286734e-01+0.j,  -9.86363287e-01+0.j,
         -8.79471112e-01+0.j,  -4.01688979e-01+0.j,   2.45493023e-01+0.j,
          7.89145284e-01+0.j,   9.99999271e-01+0.j]),
 array([  0.00000000e+00+0.j,  -6.14214010e-01+0.j,  -9.69403055e-01+0.j,
         -9.15775807e-01+0.j,  -4.75947716e-01+0.j,   1.64596791e-01+0.j,
          7.35726839e-01+0.j,   9.96586861e-01+0.j,   8.37166184e-01+0.j,
          3.24695883e-01+0.j,  -3.24704840e-01+0.j,  -8.37170685e-01+0.j,
         -9.96585195e-01+0.j,  -7.35720619e-01+0.j,  -1.64588257e-01+0.j,
          4.75953748e-01+0.j,   9.15776736e-01+0.j,   9.69398541e-01+0.j,
          6.14206262e-01+0.j,  -8.13905967e-06+0.j])]
```

The resulting list of expectation values can easily be visualized using matplotlib's plotting functions:

```
>>> tlist = linspace(0.0, 10.0, 100)
>>> result = mesolve(H, psi0, tlist, [], [sigmaz(), sigmay()])
>>>
>>> from pylab import *
>>> plot(result.times, result.expect[0])
>>> plot(result.times, result.expect[1])
>>> xlabel('Time')
```

```
>>> ylabel('Expectation values')
>>> legend(("Sigma-Z", "Sigma-Y"))
>>> show()
```



If an empty list of operators is passed as fifth parameter, the `qutip.mesolve` function returns a `qutip.Odedata` instance that contains a list of state vectors for the times specified in `tlist`:

```
>>> tlist = [0.0, 1.0]
>>> result = mesolve(H, psi0, tlist, [], [])
>>> result.states
[
Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[ 1.+0.j]
 [ 0.+0.j]]
, Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[ 0.80901765+0.j        ]
 [ 0.00000000-0.58778584j]]
, Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[ 0.3090168+0.j        ]
 [ 0.0000000-0.95105751j]]
, Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[-0.30901806+0.j        ]
 [ 0.00000000-0.95105684j]]
]
```

Non-unitary evolution

While the evolution of the state vector in a closed quantum system is deterministic, open quantum systems are stochastic in nature. The effect of an environment on the system of interest is to induce stochastic transitions between energy levels, and to introduce uncertainty in the phase difference between states of the system. The state of an open quantum system is therefore described in terms of ensemble averaged states using the density matrix formalism. A density matrix ρ describes a probability distribution of quantum states $|\psi_n\rangle$, in a matrix representation $\rho = \sum_n p_n |\psi_n\rangle \langle\psi_n|$, where p_n is the classical probability that the system is in the quantum state $|\psi_n\rangle$. The time evolution of a density matrix ρ is the topic of the remaining portions of this section.

The Lindblad Master equation

The standard approach for deriving the equations of motion for a system interacting with its environment is to expand the scope of the system to include the environment. The combined quantum system is then closed, and its evolution is governed by the von Neumann equation

$$\dot{\rho}_{\rm tot}(t) = -\frac{i}{\hbar}[H_{\rm tot}, \rho_{\rm tot}(t)], \tag{4.2}$$

the equivalent of the Schrödinger equation (4.1) in the density matrix formalism. Here, the total Hamiltonian

$$H_{\rm tot} = H_{\rm sys} + H_{\rm env} + H_{\rm int},$$

includes the original system Hamiltonian $H_{\rm sys}$, the Hamiltonian for the environment $H_{\rm env}$, and a term representing the interaction between the system and its environment $H_{\rm int}$. Since we are only interested in the dynamics of the system, we can at this point perform a partial trace over the environmental degrees of freedom in Eq. (4.2), and thereby obtain a master equation for the motion of the original system density matrix. The most general trace-preserving and completely positive form of this evolution is the Lindblad master equation for the reduced density matrix $\rho = {\rm Tr}_{\rm env}[\rho_{\rm tot}]$

$$\dot{\rho}(t) = -\frac{i}{\hbar}[H(t), \rho(t)] + \sum_n \frac{1}{2}\left[2C_n\rho(t)C_n^+ - \rho(t)C_n^+C_n - C_n^+C_n\rho(t)\right] \tag{4.3}$$

where the $C_n = \sqrt{\gamma_n}A_n$ are collapse operators, and A_n are the operators through which the environment couples to the system in $H_{\rm int}$, and γ_n are the corresponding rates. The derivation of Eq. (4.3) may be found in several sources, and will not be reproduced here. Instead, we emphasize the approximations that are required to arrive at the master equation in the form of Eq. (4.3) from physical arguments, and hence perform a calculation in QuTiP:

- **Separability:** At $t = 0$ there are no correlations between the system and its environment such that the total density matrix can be written as a tensor product $\rho^I_{\rm tot}(0) = \rho^I(0) \otimes \rho^I_{\rm env}(0)$.
- **Born approximation:** Requires: (1) that the state of the environment does not significantly change as a result of the interaction with the system; (2) The system and the environment remain separable throughout the evolution. These assumptions are justified if the interaction is weak, and if the environment is much larger than the system. In summary, $\rho_{\rm tot}(t) \approx \rho(t) \otimes \rho_{\rm env}$.
- **Markov approximation** The time-scale of decay for the environment $\tau_{\rm env}$ is much shorter than the smallest time-scale of the system dynamics $\tau_{\rm sys} \gg \tau_{\rm env}$. This approximation is often deemed a "short-memory environment" as it requires that environmental correlation functions decay on a time-scale fast compared to those of the system.
- **Secular approximation** Stipulates that elements in the master equation corresponding to transition frequencies satisfy $|\omega_{ab} - \omega_{cd}| \ll 1/\tau_{\rm sys}$, i.e., all fast rotating terms in the interaction picture can be neglected. It also ignores terms that lead to a small renormalization of the system energy levels. This approximation is not strictly necessary for all master-equation formalisms (e.g., the Block-Redfield master equation), but it is required for arriving at the Lindblad form (4.3) which is used in `qutip.mesolve`.

For systems with environments satisfying the conditions outlined above, the Lindblad master equation (4.3) governs the time-evolution of the system density matrix, giving an ensemble average of the system dynamics. In order to ensure that these approximations are not violated, it is important that the decay rates γ_n be smaller than the minimum energy splitting in the system Hamiltonian. Situations that demand special attention therefore include, for example, systems strongly coupled to their environment, and systems with degenerate or nearly degenerate energy levels.

For non-unitary evolution of a quantum systems, i.e., evolution that includes incoherent processes such as relaxation and dephasing, it is common to use master equations. In QuTiP, the same function (`qutip.mesolve`) is used for evolution both according to the Schrödinger equation and to the master equation, even though these two equations of motion are very different. The `qutip.mesolve` function automatically determines if it is sufficient to use the Schrödinger equation (if no collapse operators were given) or if it has to use the master equation (if collapse operators were given). Note that to calculate the time evolution according to the Schrödinger equation is easier and much faster (for large systems) than using the master equation, so if possible the solver will fall back on using the Schrödinger equation.

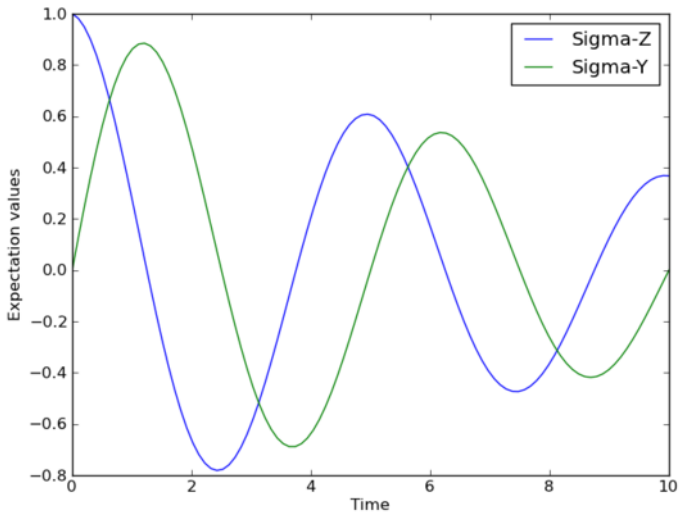
What is new in the master equation compared to the Schrödinger equation are processes that describe dissipation in the quantum system due to its interaction with an environment. These environmental interactions are defined by the operators through which the system couples to the environment, and rates that describe the strength of the processes.

In QuTiP, the product of the square root of the rate and the operator that describe the dissipation process is called a collapse operator. A list of collapse operators (`c_op_list`) is passed as the fourth argument to the `qutip.mesolve` function in order to define the dissipation processes in the master equation. When the `c_op_list` isn't empty, the `qutip.mesolve` function will use the master equation instead of the unitary Schrödinger equation.

Using the example with the spin dynamics from the previous section, we can easily add a relaxation process (describing the dissipation of energy from the spin to its environment), by adding `sqrt(0.05) * sigmax()` to the previously empty list in the fourth parameter to the `qutip.mesolve` function:

```
>>> tlist = linspace(0.0, 10.0, 100)
>>> result = mesolve(H, psi0, tlist, [sqrt(0.05) * sigmax()], [sigmaz(), sigmay()])
>>> from pylab import *
>>> plot(tlist, result.expect[0])
>>> plot(tlist, result.expect[1])
>>> xlabel('Time')
>>> ylabel('Expectation values')
>>> legend(("Sigma-Z", "Sigma-Y"))
>>> show()
```

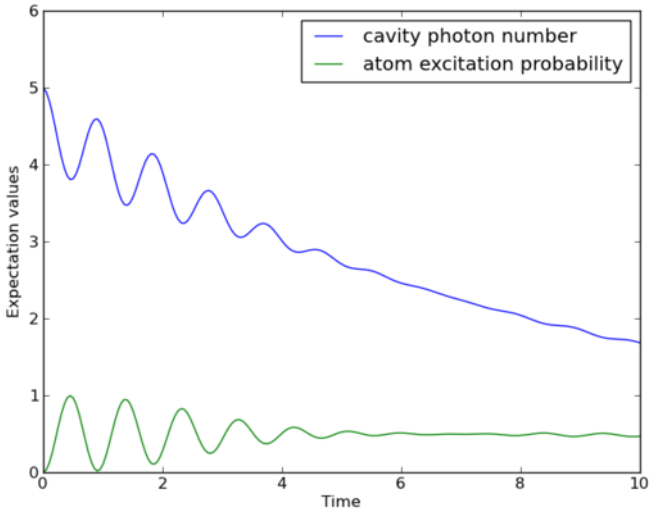
Here, 0.05 is the rate and the operator σ_x (`qutip.operators.sigmax`) describes the dissipation process.



Now a slightly more complex example: Consider a two-level atom coupled to a leaky single-mode cavity through a dipole-type interaction, which supports a coherent exchange of quanta between the two systems. If the atom initially is in its groundstate and the cavity in a 5-photon Fock state, the dynamics is calculated with the lines following code:

```
>>> tlist = linspace(0.0, 10.0, 200)
>>> psi0 = tensor(fock(2,0), fock(10, 5))
>>> a  = tensor(qeye(2), destroy(10))
>>> sm = tensor(destroy(2), qeye(10))
>>> H = 2 * pi * a.dag() * a + 2 * pi * sm.dag() * sm + 2 * pi * 0.25 * (sm * a.dag() + sm.dag()
>>> result = mesolve(H, psi0, tlist, ntraj, [sqrt(0.1)*a], [a.dag()*a, sm.dag()*sm])
>>> from pylab import *
>>> plot(tlist, result.expect[0])
>>> plot(tlist, result.expect[1])
>>> xlabel('Time')
```

```
>>> ylabel('Expectation values')
>>> legend(("cavity photon number", "atom excitation probability"))
>>> show()
```



4.5.3 Quantum Dynamics via the Monte Carlo Solver

Introduction

Where as the density matrix formalism describes the ensemble average over many identical realizations of a quantum system, the Monte Carlo (MC), or quantum-jump approach to wave function evolution, allows for simulating an individual realization of the system dynamics. Here, the environment is continuously monitored, resulting in a series of quantum jumps in the system wave function, conditioned on the increase in information gained about the state of the system via the environmental measurements. In general, this evolution is governed by the Schrödinger equation with a **non-Hermitian** effective Hamiltonian

$$H_{\rm eff} = H_{\rm sys} - \frac{i\hbar}{2}\sum_i C_n^+ C_n, \tag{4.4}$$

where again, the C_n are collapse operators, each corresponding to a separate irreversible process with rate γ_n. Here, the strictly negative non-Hermitian portion of Eq. (4.4) gives rise to a reduction in the norm of the wave function, that to first-order in a small time δt, is given by $\langle\psi(t+\delta t)|\psi(t+\delta t)\rangle = 1 - \delta p$ where

$$\delta p = \delta t \sum_n \left\langle \psi(t)|C_n^+ C_n|\psi(t)\right\rangle, \tag{4.5}$$

and δt is such that $\delta p \ll 1$. With a probability of remaining in the state $|\psi(t+\delta t)\rangle$ given by $1-\delta p$, the corresponding quantum jump probability is thus Eq. (4.5). If the environmental measurements register a quantum jump, say via the emission of a photon into the environment, or a change in the spin of a quantum dot, the wave function undergoes a jump into a state defined by projecting $|\psi(t)\rangle$ using the collapse operator C_n corresponding to the measurement

$$|\psi(t+\delta t)\rangle = C_n|\psi(t)\rangle \,/\, \left\langle\psi(t)|C_n^+ C_n|\psi(t)\right\rangle^{1/2}. \tag{4.6}$$

If more than a single collapse operator is present in Eq. (4.4), the probability of collapse due to the ith-operator C_i is given by

$$P_i(t) = \left\langle\psi(t)|C_i^+ C_i|\psi(t)\right\rangle / \delta p. \tag{4.7}$$

Evaluating the MC evolution to first-order in time is quite tedious. Instead, QuTiP uses the following algorithm to simulate a single realization of a quantum system. Starting from a pure state $|\psi(0)\rangle$:

- **I:** Choose a random number r between zero and one, representing the probability that a quantum jump occurs.
- **II:** Integrate the Schrödinger equation, using the effective Hamiltonian (4.4) until a time τ such that the norm of the wave function satisfies $\langle\psi(\tau)\,|\psi(\tau)\rangle = r$, at which point a jump occurs.
- **III:** The resultant jump projects the system at time τ into one of the renormalized states given by Eq. (4.6). The corresponding collapse operator C_n is chosen such that n is the smallest integer satisfying:

$$\sum_{i=1}^{n} P_n(\tau) \geq r \tag{4.8}$$

where the individual P_n are given by Eq. (4.7). Note that the left hand side of Eq. (4.8) is, by definition, normalized to unity.

- **IV:** Using the renormalized state from step III as the new initial condition at time τ, draw a new random number, and repeat the above procedure until the final simulation time is reached.

Monte Carlo in QuTiP

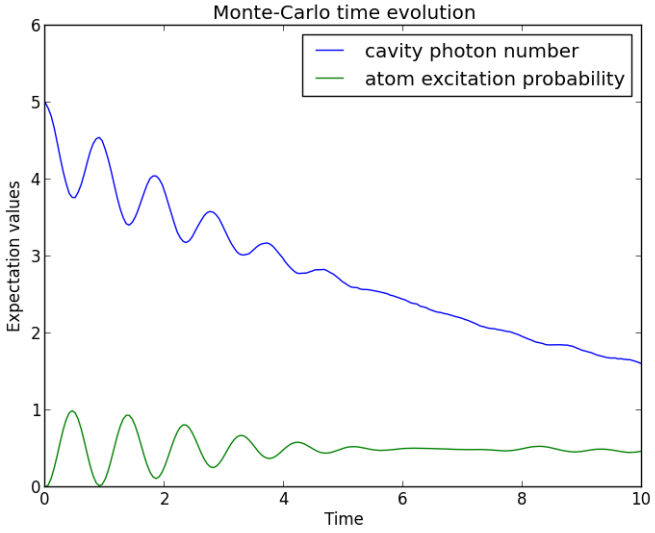
In QuTiP, Monte Carlo evolution is implemented with the `qutip.mcsolve` function. It takes nearly the same arguments as the `qutip.mesolve` function for master-equation evolution, except that the initial state must be a ket vector, as oppose to a density matrix, and there is an optional keyword parameter `ntraj` that defines the number of stochastic trajectories to be simulated. By default, `ntraj=500` indicating that 500 Monte Carlo trajectories will be performed.

To illustrate the use of the Monte Carlo evolution of quantum systems in QuTiP, let's again consider the case of a two-level atom coupled to a leaky cavity. The only differences to the master-equation treatment is that in this case we invoke the `qutip.mcsolve` function instead of `qutip.mesolve`:

```
from qutip import *
from pylab import *

tlist = linspace(0.0, 10.0, 200)
psi0 = tensor(fock(2, 0), fock(10, 5))
a  = tensor(qeye(2), destroy(10))
sm = tensor(destroy(2), qeye(10))
H = 2 * pi * a.dag() * a + 2 * pi * sm.dag() * sm + 2 * pi * 0.25 * (sm * a.dag() + sm.dag() * a)
# run Monte Carlo solver
data = mcsolve(H, psi0, tlist, [sqrt(0.1) * a], [a.dag() * a, sm.dag() * sm])
plot(tlist, data.expect[0], tlist, data.expect[1])
title('Monte Carlo time evolution')
xlabel('Time')
ylabel('Expectation values')
legend(("cavity photon number", "atom excitation probability"))
show()
```

The advantage of the Monte Carlo method over the master equation approach is that only the state vector is required to be kept in the computers memory, as opposed to the entire density matrix. For large quantum system this becomes a significant advantage, and the Monte Carlo solver is therefore generally recommended for such systems. For example, simulating a Heisenberg spin-chain consisting of 10 spins with random parameters and initial states takes almost 7 times longer using the master equation rather than Monte Carlo approach with the default number of trajectories running on a quad-CPU machine. Furthermore, it takes about 7 times the memory as well. However, for small systems, the added overhead of averaging a large number of stochastic trajectories to obtain the open system dynamics, as well as starting the multiprocessing functionality, outweighs the benefit of the minor (in this case) memory saving. Master equation methods are therefore generally more efficient when Hilbert space sizes are on the order of a couple of hundred states or smaller.

Like the master equation solver `qutip.mesolve`, the Monte Carlo solver returns a Odedata object consisting of expectation values, if the user has defined expectation value operators in the 5th argument to `mcsolve`, or state vectors if no expectation value operators are given. If state vectors are returned, then the `qutip.Odedata`



returned by `qutip.mcsolve` will be an array of length `ntraj`, with each element containing an array of ket-type qobjs with the same number of elements as `tlist`. Furthermore, the output Odedata object will also contain a list of times at which collapse occurred, and which collapse operators did the collapse, in the `col_times` and `col_which` properties, respectively. See example *Visualizing Monte Carlo Collapse Times and Operators* for an example using these properties.

Changing the Number of Trajectories

As mentioned earlier, by default, the `mcsolve` function runs 500 trajectories. This value was chosen because it gives good accuracy, Monte Carlo errors scale as $1/n$ where n is the number of trajectories, and simultaneously does not take an excessive amount of time to run. However, like many other options in QuTiP you are free to change the number of trajectories to fit your needs. If we want to run 1000 trajectories in the above example, we can simply modify the call to `mcsolve` like:

```
>>> data = mcsolve(H, psi0, tlist, [sqrt(0.1) * a], [a.dag() * a, sm.dag() * sm], ntraj=1000)
```

where we have added the keyword argument `ntraj=1000` at the end of the inputs. Now, the Monte Carlo solver will calculate expectation values for both operators, `a.dag() * a, sm.dag() * sm` averaging over 1000 trajectories. Sometimes one is also interested in seeing how the Monte Carlo trajectories converge to the master equation solution by calculating expectation values over a range of trajectory numbers. If, for example, we want to average over 1, 10, 100, and 1000 trajectories, then we can input this into the solver using:

```
>>> ntraj = [1, 10, 100, 1000]
```

Keep in mind that the input list must be in ascending order since the total number of trajectories run by `mcsolve` will be calculated using the last element of `ntraj`. In this case, we need to use an extra index when getting the expectation values from the Odedata object returned by `mcsolve`. In the above example using:

```
>>> data = mcsolve(H, psi0, tlist, [sqrt(0.1) * a], [a.dag() * a, sm.dag() * sm], ntraj=[1, 10, 1
```

we can extract the relevant expectation values using:

```
expt1 = data.expect[0]      # <- expectation values for 1 trajectory
expt10 = data.expect[1]     # <- expectation values avg. over 10 trajectories
expt100 = data.expect[2]    # <- expectation  values avg. over 100 trajectories
expt1000 = data.expect[3]   # <- expectation values avg. over 1000 trajectories
```

An example is given in *Averaging of Monte Carlo Trajectories to Master Equation Solution*.

The Monte Carlo solver also has many available options that can be set using the `qutip.Odeoptions` class as discussed in *Setting Options for the Dynamics ODE Solvers*.

Reusing Hamiltonian Data

Note: This section covers a specialized topic and may be skipped if you are new to QuTiP.

In order to solve a given simulation as fast as possible, the solvers in QuTiP take the given input operators and break them down into simpler components before passing them on to the ODE solvers. Although these operations are reasonably fast, the time spent organizing data can become appreciable when repeatedly solving a system over, for example, many different initial conditions. In cases such as this, the Hamiltonian and other operators may be reused after the initial configuration, thus speeding up calculations. Note that, unless you are planning to reuse the data many times, this functionality will not be very useful.

To turn on the "reuse" functionality we must set the `rhs_reuse=True` flag in the `qutip.Odeoptions`:

```
>>> options = Odeoptions(rhs_reuse=True)
```

A full account of this feature is given in *Setting Options for the Dynamics ODE Solvers*. Using the previous example, we will calculate the dynamics for two different initial states, with the Hamiltonian data being reused on the second call:

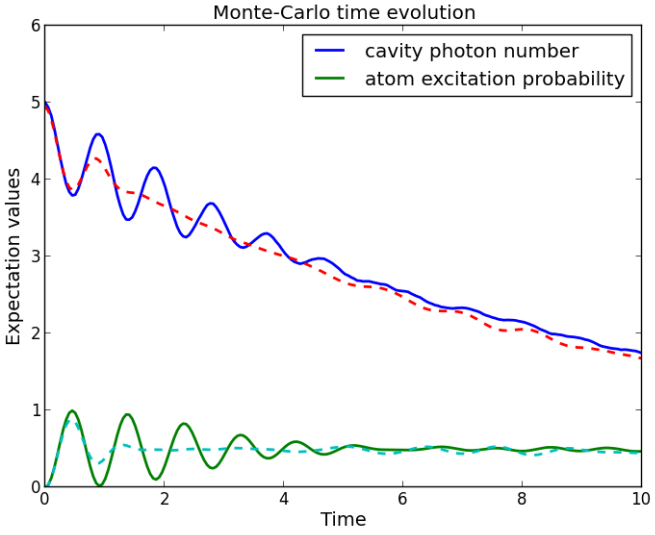
```
from qutip import *
from pylab import *

tlist = linspace(0.0, 10.0, 200)
psi0 = tensor(fock(2, 0), fock(10, 5))
a  = tensor(qeye(2), destroy(10))
sm = tensor(destroy(2), qeye(10))
H = 2 * pi * a.dag() * a + 2 * pi * sm.dag() * sm + 2 * pi * 0.25 * (sm * a.dag() + sm.dag() * a)

# first run
data1 = mcsolve(H, psi0, tlist, [sqrt(0.1) * a], [a.dag() * a, sm.dag() * sm])

# change initial state
psi1 = tensor(fock(2, 0), coherent(10, 2 - 1j))

# run again, reusing data
opts = Odeoptions(rhs_reuse=True)
data2 = mcsolve(H, psi1, tlist, [sqrt(0.1) * a], [a.dag() * a, sm.dag() * sm], options=opts)

# plot both results
plot(tlist, data1.expect[0], tlist, data1.expect[1], lw=2)
plot(tlist, data2.expect[0], '--', tlist, data2.expect[1], '--', lw=2)
title('Monte Carlo time evolution')
xlabel('Time', fontsize=14)
ylabel('Expectation values', fontsize=14)
legend(("cavity photon number", "atom excitation probability"))
show()
```

In addition to the initial state, one may reuse the Hamiltonian data when changing the number of trajectories `ntraj` or simulation times `tlist`. The reusing of Hamiltonian data is also supported for time-dependent Hamiltonians. See *Solving Problems with Time-dependent Hamiltonians* for further details.

Fortran Based Monte Carlo Solver

Note: In order to use the Fortran Monte Carlo solver, you must have the blas development libraries, and installed QuTiP using the flag: `--with-f90mc`.

(New in QuTiP 2.2)



In performing time-independent Monte Carlo simulations with QuTiP, systems with small Hilbert spaces suffer from poor performance as the ODE solver must exit the ODE solver at each time step and check for the state vector norm. To correct this, QuTiP now includes an optional Fortran based Monte Carlo solver that has markedly enhanced performance for smaller systems. Using the Fortran based solver is extremely simple; one just needs to replace `mcsolve` with `mcsolve_f90`. For example, from our previous demonstation:

```
data1 = mcsolve_f90(H, psi0, tlist, [sqrt(0.1) * a], [a.dag() * a, sm.dag() * sm])
```

In using the Fortran solver, there are a few limitations that must be kept in mind. First, this solver only works for time-independent systems. Second, you can not pass a list of trajectories to `ntraj`. Finally, at present, this solver does not give any output has to how far along the simulation has progressed, nor how much time is remaining. These issues will be fixed in a later QuTiP release.

4.5.4 Bloch-Redfield master equation

Introduction

The Lindblad master equation introduced earlier is constructed so that it describes a physical evolution of the density matrix (i.e., trace and positivity preserving), but it does not provide a connection to any underlaying microscopic physical model. The Lindblad operators (collapse operators) describe phenomenological processes, such as for example dephasing and spin flips, and the rates of these processes are arbitrary parameters in the model. In many situations the collapse operators and their corresponding rates have clear physical interpretation, such as dephasing and relaxation rates, and in those cases the Lindblad master equation is usually the method of choice.

However, in some cases, for example systems with varying energy biases and eigenstates and that couple to an environment in some well-defined manner (through a physically motivated system-environment interaction operator), it is often desirable to derive the master equation from more fundamental physical principles, and relate it to for example the noise-power spectrum of the environment.

The Bloch-Redfield formalism is one such approach to derive a master equation from a microscopic system. It starts from a combined system-environment perspective, and derives a perturbative master equation for the system alone, under the assumption of weak system-environment coupling. One advantage of this approach is that the dissipation processes and rates are obtained directly from the properties of the environment. On the downside, it does not intrinsically guarantee that the resulting master equation unconditionally preserves the physical properties of the density matrix (because it is a perturbative method). The Bloch-Redfield master equation must therefore be used with care, and the assumptions made in the derivation must be honored. (The Lindblad master equation is in a sense more robust – it always results in a physical density matrix – although some collapse operators might not be physically justified). For a full derivation of the Bloch Redfield master equation, see e.g. Atom-Physics Interactions by Cohen-Tannoudji *et al.* (Wiley, 1992), or Theory of open quantum systems by Breuer

and Petruccione (Oxford, 2002). Here we present only a brief version of the derivation, with the intention of introducing the notation and how it relates to the implementation in QuTiP.

Brief Derivation and Definitions

The starting point of the Bloch-Redfield formalism is the total Hamiltonian for the system and the environment (bath): $H = H_{\rm S} + H_{\rm B} + H_{\rm I}$, where H is the total system+bath Hamiltonian, $H_{\rm S}$ and $H_{\rm B}$ are the system and bath Hamiltonians, respectively, and $H_{\rm I}$ is the interaction Hamiltonian.

The most general form of a master equation for the system dynamics is obtained by tracing out the bath from the von-Neumann equation of motion for the combined system ($\dot{\rho} = -i\hbar^{-1}[H, \rho]$). In the interaction picture the result is

$$\frac{d}{dt}\rho_S(t) = -\hbar^{-2}\int_0^t d\tau\ {\rm Tr}_B[H_I(t), [H_I(\tau), \rho_S(\tau)\otimes\rho_B]], \tag{4.9}$$

where the additional assumption that the total system-bath density matrix can be factorized as $\rho(t) \approx \rho_S(t)\otimes\rho_B$. This assumption is known as the Born approximation, and it implies that there never is any entanglement between the system and the bath, neither in the initial state nor at any time during the evolution. *It is justified for weak system-bath interaction.*

The master equation (4.9) is non-Markovian, i.e., the change in the density matrix at a time t depends on states at all times $\tau < t$, making it intractable to solve both theoretically and numerically. To make progress towards a manageable master equation, we now introduce the Markovian approximation, in which $\rho(s)$ is replaced by $\rho(t)$ in Eq. (4.9). The result is the Redfield equation

$$\frac{d}{dt}\rho_S(t) = -\hbar^{-2}\int_0^t d\tau\ {\rm Tr}_B[H_I(t), [H_I(\tau), \rho_S(t)\otimes\rho_B]], \tag{4.10}$$

which is local in time with respect the density matrix, but still not Markovian since it contains an implicit dependence on the initial state. By extending the integration to infinity and substituting $\tau \to t - \tau$, a fully Markovian master equation is obtained:

$$\frac{d}{dt}\rho_S(t) = -\hbar^{-2}\int_0^\infty d\tau\ {\rm Tr}_B[H_I(t), [H_I(t-\tau), \rho_S(t)\otimes\rho_B]]. \tag{4.11}$$

The two Markovian approximations introduced above are valid if the time-scale with which the system dynamics changes is large compared to the time-scale with which correlations in the bath decays (corresponding to a "short-memory" bath, which results in Markovian system dynamics).

The master equation (4.11) is still on a too general form to be suitable for numerical implementation. We therefore assume that the system-bath interaction takes the form $H_I = \sum_\alpha A_\alpha \otimes B_\alpha$ and where A_α are system operators and B_α are bath operators. This allows us to write master equation in terms of system operators and bath correlation functions:

$$\begin{aligned}\frac{d}{dt}\rho_S(t) = -\hbar^{-2}\sum_{\alpha\beta}\int_0^\infty d\tau\ \{&g_{\alpha\beta}(\tau)\left[A_\alpha(t)A_\beta(t-\tau)\rho_S(t) - A_\alpha(t-\tau)\rho_S(t)A_\beta(t)\right] \\ &g_{\alpha\beta}(-\tau)\left[\rho_S(t)A_\alpha(t-\tau)A_\beta(t) - A_\alpha(t)\rho_S(t)A_\beta(t-\tau)\right]\},\end{aligned}$$

where $g_{\alpha\beta}(\tau) = {\rm Tr}_B\left[B_\alpha(t)B_\beta(t-\tau)\rho_B\right] = \langle B_\alpha(\tau)B_\beta(0)\rangle$, since the bath state ρ_B is a steady state.

In the eigenbasis of the system Hamiltonian, where $A_{mn}(t) = A_{mn}e^{i\omega_{mn}t}$, $\omega_{mn} = \omega_m - \omega_n$ and ω_m are the eigenfrequencies corresponding the eigenstate $|m\rangle$, we obtain in matrix form in the Schrödinger picture

$$\begin{aligned}\frac{d}{dt}\rho_{ab}(t) = -i\omega_{ab}\rho_{ab}(t) - \hbar^{-2}\sum_{\alpha,\beta}\sum_{c,d}^{\rm sec}\int_0^\infty d\tau\ \Bigg\{&g_{\alpha\beta}(\tau)\left[\delta_{bd}\sum_n A^\alpha_{an}A^\beta_{nc}e^{i\omega_{cn}\tau} - A^\alpha_{ac}A^\beta_{db}e^{i\omega_{ca}\tau}\right] \\ &+ g_{\alpha\beta}(-\tau)\left[\delta_{ac}\sum_n A^\alpha_{dn}A^\beta_{nb}e^{i\omega_{nd}\tau} - A^\alpha_{ac}A^\beta_{db}e^{i\omega_{bd}\tau}\right]\Bigg\}\rho_{cd}(t),\end{aligned}$$

where the "sec" above the summation symbol indicate summation of the secular terms which satisfy $|\omega_{ab} - \omega_{cd}| \ll \tau_{\rm decay}$. This is an almost-useful form of the master equation. The final step before arriving at the form of the Bloch-Redfield master equation that is implemented in QuTiP, involves rewriting the bath correlation function $g(\tau)$ in terms of the noise-power spectrum of the environment $S(\omega) = \int_{-\infty}^{\infty} d\tau e^{i\omega\tau} g(\tau)$:

$$\int_0^\infty d\tau\ g_{\alpha\beta}(\tau) e^{i\omega\tau} = \frac{1}{2} S_{\alpha\beta}(\omega) + i\lambda_{\alpha\beta}(\omega), \tag{4.12}$$

where $\lambda_{ab}(\omega)$ is an energy shift that is neglected here. The final form of the Bloch-Redfield master equation is

$$\frac{d}{dt}\rho_{ab}(t) = -i\omega_{ab}\rho_{ab}(t) + \sum_{c,d}^{\rm sec} R_{abcd}\rho_{cd}(t), \tag{4.13}$$

where

$$R_{abcd} = -\frac{\hbar^{-2}}{2} \sum_{\alpha,\beta} \left\{ \delta_{bd} \sum_n A^\alpha_{an} A^\beta_{nc} S_{\alpha\beta}(\omega_{cn}) - A^\alpha_{ac} A^\beta_{db} S_{\alpha\beta}(\omega_{ca}) + \delta_{ac} \sum_n A^\alpha_{dn} A^\beta_{nb} S_{\alpha\beta}(\omega_{dn}) - A^\alpha_{ac} A^\beta_{db} S_{\alpha\beta}(\omega_{db}) \right\},$$

is the Bloch-Redfield tensor.

The Bloch-Redfield master equation in the form Eq. (4.13) is suitable for numerical implementation. The input parameters are the system Hamiltonian H, the system operators through which the environment couples to the system A_α, and the noise-power spectrum $S_{\alpha\beta}(\omega)$ associated with each system-environment interaction term.

To simplify the numerical implementation we assume that A_α are Hermitian and that cross-correlations between different environment operators vanish, so that the final expression for the Bloch-Redfield tensor that is implemented in QuTiP is

$$R_{abcd} = -\frac{\hbar^{-2}}{2} \sum_{\alpha} \left\{ \delta_{bd} \sum_n A^\alpha_{an} A^\alpha_{nc} S_{\alpha}(\omega_{cn}) - A^\alpha_{ac} A^\alpha_{db} S_{\alpha}(\omega_{ca}) + \delta_{ac} \sum_n A^\alpha_{dn} A^\alpha_{nb} S_{\alpha}(\omega_{dn}) - A^\alpha_{ac} A^\alpha_{db} S_{\alpha}(\omega_{db}) \right\}.$$

Bloch-Redfield master equation in QuTiP

In QuTiP, the Bloch-Redfield tensor Eq. (4.5.4) can be calculated using the function `qutip.bloch_redfield.bloch_redfield_tensor`. It takes three mandatory arguments: The system Hamiltonian H, a list of operators through which to the bath A_α, and a list of corresponding spectral density functions $S_\alpha(\omega)$. The spectral density functions are callback functions that takes the (angular) frequency as a single argument.

To illustrate how to calculate the Bloch-Redfield tensor, let's consider a two-level atom

$$H = -\frac{1}{2}\Delta\sigma_x - \frac{1}{2}\epsilon_0\sigma_z \tag{4.14}$$

that couples to an Ohmic bath through the σ_x operator. The corresponding Bloch-Redfield tensor can be calculated in QuTiP using the following code:

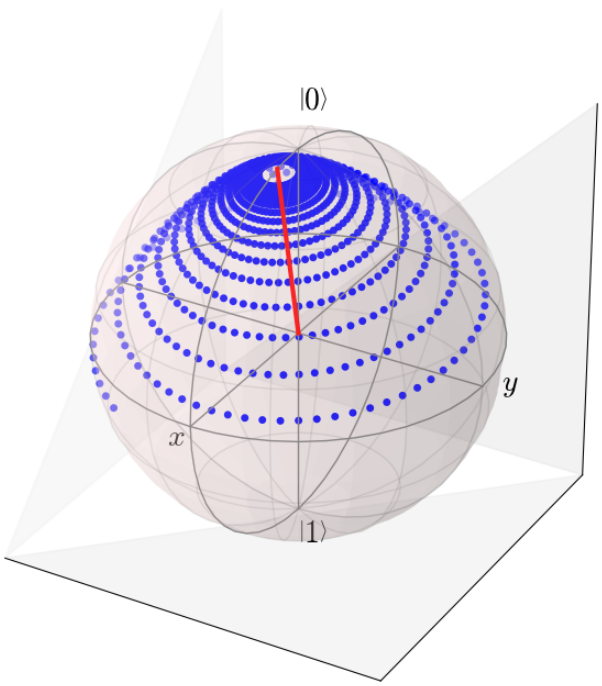
```
>>> delta = 0.2 * 2*pi; eps0 = 1.0 * 2*pi; gamma1 = 0.5
>>> H = - delta/2.0 * sigmax() - eps0/2.0 * sigmaz()
>>> def ohmic_spectrum(w):
>>>     if w == 0.0: # dephasing inducing noise
>>>         return gamma1
>>>     else: # relaxation inducing noise
>>>         return gamma1 / 2 * (w / (2 * pi)) * (w > 0.0)
>>>
```

```
>>> R, ekets = bloch_redfield_tensor(H, [sigmax()], [ohmic_spectrum])
>>> real(R.full())
array([[ 0.        ,  0.        ,  0.        ,  0.04902903],
       [ 0.        , -0.03220682,  0.        ,  0.        ],
       [ 0.        ,  0.        , -0.03220682,  0.        ],
       [ 0.        ,  0.        ,  0.        , -0.04902903]])
```

For convenience, the function `qutip.bloch_redfield.bloch_redfield_tensor` also returns a list of eigenkets *ekets*, since they are calculated in the process of calculating the Bloch-Redfield tensor *R*, and the *ekets* are usually needed again later when transforming operators between the computational basis and the eigenbasis.

The evolution of a wavefunction or density matrix, according to the Bloch-Redfield master equation (4.13), can be calculated using the QuTiP function `qutip.bloch_redfield.bloch_redfield_solve`. It takes five mandatory arguments: the Bloch-Redfield tensor `R`, the list of eigenkets `ekets`, the initial state `psi0` (as a ket or density matrix), a list of times `tlist` for which to evaluate the expectation values, and a list of operators `e_ops` for which to evaluate the expectation values at each time step defined by *tlist*. For example, to evaluate the expectation values of the σ_x, σ_y, and σ_z operators for the example above, we can use the following code:

```
>>> tlist = linspace(0, 15.0, 1000)
>>> psi0 = rand_ket(2)
>>> e_ops = [sigmax(), sigmay(), sigmaz()]
>>> expt_list = bloch_redfield_solve(R, ekets, psi0, tlist, e_ops)
>>>
>>> sphere = Bloch()
>>> sphere.add_points([expt_list[0], expt_list[1], expt_list[2]])
>>> sphere.vector_color = ['r']
>>> # Hamiltonian axis
>>> sphere.add_vectors(array([delta, 0, eps0]) / sqrt(delta ** 2 + eps0 ** 2))
>>> sphere.make_sphere()
>>> show()
```



The two steps of calculating the Bloch-Redfield tensor and evolve the corresponding master equation can be combined into one by using the function `qutip.bloch_redfield.brmesolve`, which takes same arguments as `qutip.mesolve` and `qutip.mcsolve`, expect for the additional list of spectral callback functions.

```
>>> output = brmesolve(H, psi0, tlist, [sigmax()], e_ops, [ohmic_spectrum])
```

where the resulting *output* is an instance of the class `qutip.Odedata`.

4.5.5 Solving Problems with Time-dependent Hamiltonians

Methods for Writing Time-Dependent Operators

In the previous examples of quantum evolution, we assumed that the systems under consideration were described by time-independent Hamiltonians. However, many systems have explicit time dependence in either the Hamiltonian, or the collapse operators describing coupling to the environment, and sometimes both components might depend on time. The two main evolution solvers in QuTiP, `qutip.mesolve` and `qutip.mcsolve`, discussed in *Lindblad Master Equation Solver* and *Quantum Dynamics via the Monte Carlo Solver* respectively, are capable of handling time-dependent Hamiltonians and collapse terms. There are, in general, three different ways to implement time-dependent problems in QuTiP 2:

1. **Function based**: Hamiltonian / collapse operators expressed using [qobj, func] pairs, where the time-dependent coefficients of the Hamiltonian (or collapse operators) are expressed in the Python functions.
2. **String (Cython) based**: The Hamiltonian and/or collapse operators are expressed as a list of [qobj, string] pairs, where the time-dependent coefficients are represented as strings. The resulting Hamiltonian is then compiled into C code using Cython and executed.
3. **Hamiltonian function (outdated)**: The Hamiltonian is itself a Python function with time-dependence. Collapse operators must be time independent using this input format.

Give the multiple choices of input style, the first question that arrises is which option to choose? In short, the function based method (option #1) is the most general, allowing for essentially arbitrary coefficients expressed via user defined functions. However, by automatically compiling your system into C code, the second option (string based) tends to be more efficient and will run faster. Of course, for small system sizes and evolution times, the difference will be minor. Although this method does not support all time-dependent coefficients that one can think of, it does support essentially all problems that one would typically encounter. If you can write you time-dependent coefficients using any of the following functions, or combinations thereof (including constants) then you may use this method:

```
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p',
'modf', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc'
```

Finally option #3, expressing the Hamiltonian as a Python function, is the original method for time dependence in QuTiP 1.x. However, this method is somewhat less efficient then the previously mentioned methods, and does not allow for time-dependent collapse operators. However, in contrast to options #1 and #2, this method can be used in implementing time-dependent Hamiltonians that cannot be expressed as a function of constant operators with time-dependent coefficients.

A collection of examples demonstrating the simulation of time-dependent problems can be found here: *Dynamics of time-dependent systems*.

Function Based Time Dependence

A very general way to write a time-dependent Hamiltonian or collapse operator is by using Python functions as the time-dependent coefficients. To accomplish this, we need to write a Python function that returns the time-dependent coefficient. Additionally, we need to tell QuTiP that a given Hamiltonian or collapse operator should be associated with a given Python function. To do this, one needs to specify operator-function pairs in list format: `[Op, py_coeff]`, where `Op` is a given Hamiltonian or collapse operator and `py_coeff` is the name of the Python function representing the coefficient. With this format, the form of the Hamiltonian for both `mesolve` and `mcsolve` is:

```
>>> H = [H0, [H1, py_coeff1], [H2, py_coeff2], ...]
```

where `H0` is a time-independent Hamiltonian, while `H1`,''H2'', are time dependent. The same format can be used for collapse operators:

```
>>> c_op_list = [[C0, py_coeff0], C1, [C2, py_coeff2], ...]
```

Here we have demonstrated that the ordering of time-dependent and time-independent terms does not matter. In addition, any or all of the collapse operators may be time dependent.

Note: While, in general, you can arrange time-dependent and time-independent terms in any order you like, it is best to place all time-independent terms first.

As an example, we will look at *Single photon source based on a three level atom strongly coupled to a cavity* that has a time-dependent Hamiltonian of the form $H = H_0 - f(t)H_1$ where $f(t)$ is the time-dependent driving strength given as $f(t) = A\exp\left[-(t/\sigma)^2\right]$. The follow code sets up the problem:

```
from qutip import *
from scipy import *
# Define atomic states. Use ordering from paper
ustate = basis(3, 0)
excited = basis(3, 1)
ground = basis(3, 2)

# Set where to truncate Fock state for cavity
N = 2

# Create the atomic operators needed for the Hamiltonian
sigma_ge = tensor(qeye(N), ground * excited.dag())  # |g><e|
sigma_ue = tensor(qeye(N), ustate * excited.dag())  # |u><e|

# Create the photon operator
a = tensor(destroy(N), qeye(3))
ada = tensor(num(N), qeye(3))

# Define collapse operators
c_op_list = []
# Cavity decay rate
kappa = 1.5
c_op_list.append(sqrt(kappa) * a)

# Atomic decay rate
gamma = 6  # decay rate
# Use Rb branching ratio of 5/9 e->u, 4/9 e->g
c_op_list.append(sqrt(5*gamma/9) * sigma_ue)
c_op_list.append(sqrt(4*gamma/9) * sigma_ge)

# Define time vector
t = linspace(-15, 15, 100)

# Define initial state
psi0 = tensor(basis(N, 0), ustate)

# Define states onto which to project
state_GG = tensor(basis(N, 1), ground)
sigma_GG = state_GG * state_GG.dag()
state_UU = tensor(basis(N, 0), ustate)
sigma_UU = state_UU * state_UU.dag()

# Set up the time varying Hamiltonian
g = 5  # coupling strength
```

```
H0 = -g * (sigma_ge.dag() * a + a.dag() * sigma_ge)  # time-independent term
H1 = (sigma_ue.dag() + sigma_ue)  # time-dependent term
```

Given that we have a single time-dependent Hamiltonian term, and constant collapse terms, we need to specify a single Python function for the coefficient $f(t)$. In this case, one can simply do:

```
def H1_coeff(t, args):
        return 9 * exp(-(t / 5.) ** 2)
```

In this case, the return value dependents only on time. However, when specifying Python functions for coefficients, **the function must have (t,args) as the input variables, in that order**. Having specified our coefficient function, we can now specify the Hamiltonian in list format and call the solver (in this case `qutip.mesolve`):

```
H=[H0,[H1,H1_coeff]]
output = mesolve(H, psi0, t, c_op_list,[ada, sigma_UU, sigma_GG])
```

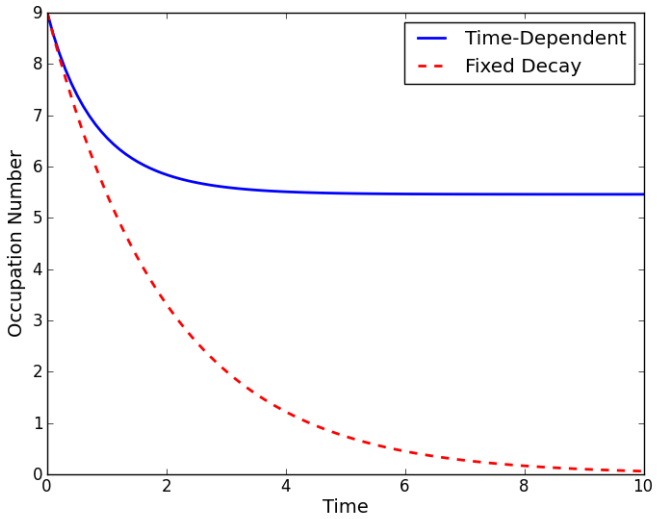
We can call the Monte Carlo solver in the exact same way (if using the default `ntraj=500`):

```
>>> output = mcsolve(H, psi0, t, c_op_list,[ada, sigma_UU, sigma_GG])
```

The output from the master equation solver is identical to that shown in the examples, the Monte Carlo however will be noticeably off, suggesting we should increase the number of trajectories for this example. In addition, we can also consider the decay of a simple Harmonic oscillator with time-varying decay rate:

```
from qutip import *
kappa = 0.5
def col_coeff(t, args):  # coefficient function
    return sqrt(kappa * exp(-t))
N = 10  # number of basis states
a = destroy(N)
H = a.dag() * a  # simple HO
psi0 = basis(N, 9)  # initial state
c_op_list = [[a, col_coeff]]  # time-dependent collapse term
tlist = linspace(0, 10, 100)
output = mesolve(H, psi0, tlist, c_op_list, [a.dag() * a])
```

A comparison of this time-dependent damping, with that of a constant decay term is presented below.



Using the args variable

In the previous example we hardcoded all of the variables, driving amplitude A and width σ, with their numerical values. This is fine for problems that are specialized, or that we only want to run once. However, in many cases, we would like to change the parameters of the problem in only one location (usually at the top of the script), and not have to worry about manually changing the values on each run. QuTiP allows you to accomplish this using the keyword `args` as an input to the solvers. For instance, instead of explicitly writing 9 for the amplitude and 5 for the width of the gaussian driving term, we can make us of the args variable:

```
def H1_coeff(t, args):
    return args['A'] * exp(-(t/args['sigma'])**2)
```

or equivalently:

```
def H1_coeff(t, args):
    A = args['A']
    sig = args['sigma']
    return A * exp(-(t / sig) ** 2)
```

where args is a Python dictionary of `key: value` pairs `args = {'A': a, 'sigma': b}` where `a` and `b` are the two parameters for the amplitude and width, respectively. Of course, we can always hardcode the values in the dictionary as well `args = {'A': 9, 'sigma': 5}`, but there is much more flexibility by using variables in `args`. To let the solvers know that we have a set of args to pass we append the `args` to the end of the solver input:

```
>>> output = mesolve(H, psi0, tlist, c_op_list, [a.dag() * a], args={'A': 9, 'sigma': 5})
```

or to keep things looking pretty:

```
args = {'A': 9, 'sigma': 5}
output = mesolve(H, psi0, tlist, c_op_list, [a.dag() * a], args=args)
```

Once again, the Monte Carlo solver `qutip.mcsolve` works in an identical manner.

String Format Method

Note: You must have Cython installed on your computer to use this format. See *Installation* for instructions on installing Cython.

The string-based time-dependent format works in a similar manner as the previously discussed Python function method. That being said, the underlying code does something completely different. When using this format, the strings used to represent the time-dependent coefficients, as well as Hamiltonian and collapse operators, are rewritten as Cython code using a code generator class and then compiled into C code. The details of this meta-programming will be published in due course. however, in short, this can lead to a substantial reduction in time for complex time-dependent problems, or when simulating over long intervals. We remind the reader that the types of functions that can be used with this method is limited to:

```
['acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil'
, 'copysign', 'cos', 'cosh', 'degrees', 'erf', 'erfc', 'exp', 'expm1'
, 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma'
, 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p'
, 'modf', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Like the previous method, the string-based format uses a list pair format `[Op, str]` where `str` is now a string representing the time-dependent coefficient. For our first example, this string would be `'9 * exp(-(t / 5.) ** 2)'`. The Hamiltonian in this format would take the form:

```
>>> H = [H0, [H1, '9 * exp(-(t / 5.) ** 2)']]
```

Notice that this is a valid Hamiltonian for the string-based format as `exp` is included in the above list of suitable functions. Calling the solvers is the same as before:

```
>>> output = mesolve(H, psi0, tlist, c_op_list, [a.dag() * a])
```

We can also use the `args` variable in the same manner as before, however we must rewrite our string term to read: `'A * exp(-(t / sig) ** 2)'`:

```
H = [H0, [H1, 'A * exp(-(t / sig) ** 2)']]
args = {'A': 9, 'sig': 5}
output = mesolve(H, psi0, tlist, c_op_list, [a.dag()*a], args=args)
```

Important: Naming your `args` variables `e` or `pi` will mess things up when using the string-based format.

Collapse operators are handled in the exact same way.

Function Based Hamiltonian

In the previous version of QuTiP, the simulation of time-dependent problems required writing the Hamiltonian itself as a Python function. This is in fact the method used in our example *Single photon source based on a three level atom strongly coupled to a cavity*. However, this method does not allow for time-dependent collapse operators, and is therefore more restrictive. Furthermore, it is less efficient than the other methods for all but the most basic of Hamiltonians (see the next section for a comparison of times.). In this format, the entire Hamiltonian is written as a Python function:

```
def Hfunc(t, args):
    H0 = args[0]
    H1 = args[1]
    w = 9 * exp(-(t/5.)**2)
    return H0 - w * H1
```

where the `args` variable **must always be given**, and is now a `list` of Hamiltonian terms: `args=[H0, H1]`. In this format, our call to the master equation is now:

```
>>> output = mesolve(Hfunc, psi0, tlist, c_op_list, [a.dag() * a], args=[H0, H1])
```

We cannot evaluate time-dependent collapse operators in this format, so we can not simulate the previous harmonic oscillator decay example.

A Quick Comparison of Simulation Times

Here we give a table of simulation times for the single-photon example using the different time-dependent formats and both the master equation and Monte Carlo solver.

Format	Master Equation	Monte Carlo
Python Function	2.1 sec	27 sec
Cython String	1.4 sec	9 sec
Hamiltonian Function	1.0 sec	238 sec

For the current example, the table indicates that the Hamiltonian function method is in fact the fastest when using the master equation solver. This is because the simulation is quite small. In contrast, the Hamiltonian function is over 26x slower than the compiled string version when using the Monte Carlo solver. In this case, the 500 trajectories needed in the simulation highlights the inefficient nature of the Python function calls.

Reusing Time-Dependent Hamiltonian Data

Note: This section covers a specialized topic and may be skipped if you are new to QuTiP.

When repeatedly simulating a system where only the time-dependent variables, or initial state change, it is possible to reuse the Hamiltonian data stored in QuTiP and there by avoid spending time needlessly preparing the Hamiltonian and collapse terms for simulation. To turn on the the reuse features, we must pass a `qutip.Odeoptions` object with the `rhs_reuse` flag turned on. Instructions on setting flags are found in *Setting Options for the Dynamics ODE Solvers*. For example, we can do:

```
H = [H0, [H1, 'A * exp(-(t / sig) ** 2)']]
args = {'A': 9, 'sig': 5}
output = mcsolve(H, psi0, tlist, c_op_list, [a.dag()*a], args=args)
opts = Odeoptions(rhs_reuse=True)
args = {'A': 10, 'sig': 3}
output = mcsolve(H, psi0, tlist, c_op_list, [a.dag()*a], args=args, options=opts)
```

In this case, the second call to `qutip.mcsolve` takes 3 seconds less than the first. Of course our parameters are different, but this also shows how much time one can save by not reorganizing the data, and in the case of the string format, not recompiling the code. If you need to call the solvers many times for different parameters, this savings will obviously start to add up.

Running String-Based Time-Dependent Problems using Parfor

Note: This section covers a specialized topic and may be skipped if you are new to QuTiP.

In this section we discuss running string-based time-dependent problems using the `qutip.parfor` function. As the `qutip.mcsolve` function is already parallelized, running string-based time dependent problems inside of parfor loops should be restricted to the `qutip.mesolve` function only. When using the string-based format, the system Hamiltonian and collapse operators are converted into C code with a specific file name that is automatically genrated, or supplied by the user via the `rhs_filename` property of the `qutip.Odeoptions` class. Because the `qutip.parfor` function uses the built-in Python multiprocessing functionality, in calling the solver inside a parfor loop, each thread will try to generate compiled code with the same file name, leading to a crash. To get around this problem you can call the `qutip.rhs_generate` function to compile simulation into C code before calling parfor. You **must** then set the `qutip.Odedata` object `rhs_reuse=True` for all solver calls inside the parfor loop that indicates that a valid C code file already exists and a new one should not be generated. As an example, we will look at the Landau-Zener-Stuckelberg interferometry example that can be found in the *Advanced topics and examples* section.

To set up the problem, we run the following code:

```
from qutip import *

# set up the parameters and start calculation
delta    = 0.1  * 2 * pi  # qubit sigma_x coefficient
w        = 2.0  * 2 * pi  # driving frequency
T        = 2 * pi / w     # driving period
gamma1   = 0.00001        # relaxation rate
gamma2   = 0.005          # dephasing  rate
eps_list = linspace(-10.0, 10.0, 501) * 2 * pi  # epsilon
A_list   = linspace(0.0, 20.0, 501) * 2 * pi    # Amplitude

# pre-calculate the necessary operators
sx = sigmax(); sz = sigmaz(); sm = destroy(2); sn = num(2)
# collapse operators
c_op_list = [sqrt(gamma1) * sm, sqrt(gamma2) * sz]  # relaxation and dephasing

# setup time-dependent Hamiltonian (list-string format)
H0 = -delta / 2.0 * sx
H1 = [sz, '-eps / 2.0 + A / 2.0 * sin(w * t)']
H_td = [H0, H1]
Hargs = {'w': w, 'eps': eps_list[0], 'A': A_list[0]}
```

where the last code block sets up the problem using a string-based Hamiltonian, and `Hargs` is a dictionary of

arguments to be passed into the Hamiltonian. In this example, we are going to use the `qutip.propagator` and `qutip.propagator.propagator_steadystate` to find expectation values for different values of ϵ and A in the Hamiltonian $H = -\frac{1}{2}\Delta\sigma_x - \frac{1}{2}\epsilon\sigma_z - \frac{1}{2}A\sin(\omega t)$.

We must now tell the `qutip.mesolve` function, that is called by `qutip.propagator` to reuse a pre-generated Hamiltonian constructed using the `qutip.rhs_generate` command:

```
# ODE settings (for reusing list-str format Hamiltonian)
opts = Odeoptions(rhs_reuse=True)
# pre-generate RHS so we can use parfor
rhs_generate(H_td, c_op_list, Hargs, name='lz_func')
```

Here, we have given the generated file a custom name `lz_func`, however this is not necessary as a generic name will automatically be given. Now we define the function `task` that is called by parfor:

```
# a task function for the for-loop parallelization:
# the m-index is parallelized in loop over the elements of p_mat[m,n]
def task(args):
    m, eps = args
    p_mat_m = zeros(len(A_list))
    for n, A in enumerate(A_list):
        # change args sent to solver, w is really a constant though.
        Hargs = {'w': w, 'eps': eps,'A': A}
        U = propagator(H_td, T, c_op_list, Hargs, opts) #<- IMPORTANT LINE
        rho_ss = propagator_steadystate(U)
        p_mat_m[n] = expect(sn, rho_ss)
    return [m, p_mat_m]
```

Notice the Odeoptions `opts` in the call to the `qutip.propagator` function. This is tells the `qutip.mesolve` function used in the propagator to call the pre-generated file `lz_func`. If this were missing then the routine would fail.

4.5.6 Floquet Formalism

Introduction

Many time-dependent problems of interest are periodic. The dynamics of such systems can be solved for directly by numerical integration of the Schrödinger or Master equation, using the time-dependent Hamiltonian. But they can also be transformed into time-independent problems using the Floquet formalism. Time-independent problems can be solve much more efficiently, so such a transformation is often very desirable.

In the standard derivations of the Lindblad and Bloch-Redfield master equations the Hamiltonian describing the system under consideration is assumed to be time independent. Thus, strictly speaking, the standard forms of these master equation formalisms should not blindly be applied to system with time-dependent Hamiltonians. However, in many relevant cases, in particular for weak driving, the standard master equations still turns out to be useful for many time-dependent problems. But a more rigorous approach would be to rederive the master equation taking the time-dependent nature of the Hamiltonian into account from the start. The Floquet-Markov Master equation is one such a formalism, with important applications for strongly driven systems [see e.g., Grifoni et al., Physics Reports 304, 299 (1998)].

Here we give an overview of how the Floquet and Floquet-Markov formalisms can be used for solving time-dependent problems in QuTiP. To introduce the terminology and naming conventions used in QuTiP we first give a brief summary of quantum Floquet theory.

Floquet theory for unitary evolution

The Schrödinger equation with a time-dependent Hamiltonian $H(t)$ is

$$H(t)\Psi(t) = i\hbar\frac{\partial}{\partial t}\Psi(t), \tag{4.15}$$

where $\Psi(t)$ is the wave function solution. Here we are interested in problems with periodic time-dependence, i.e., the Hamiltonian satisfies $H(t) = H(t+T)$ where T is the period. According to the Floquet theorem, there exist solutions to (4.15) on the form

$$\Psi_\alpha(t) = \exp(-i\epsilon_\alpha t/\hbar)\Phi_\alpha(t), \tag{4.16}$$

where $\Psi_\alpha(t)$ are the *Floquet states* (i.e., the set of wave function solutions to the Schrödinger equation), $\Phi_\alpha(t) = \Phi_\alpha(t+T)$ are the periodic *Floquet modes*, and ϵ_α are the *quasienergy levels*. The quasienergy levels are constants in time, but only uniquely defined up to multiples of $2\pi/T$ (i.e., unique value in the interval $[0, 2\pi/T]$).

If we know the Floquet modes (for $t \in [0, T]$) and the quasienergies for a particular $H(t)$, we can easily decompose any initial wavefunction $\Psi(t=0)$ in the Floquet states and immediately obtain the solution for arbitrary t

$$\Psi(t) = \sum_\alpha c_\alpha \Psi_\alpha(t) = \sum_\alpha c_\alpha \exp(-i\epsilon_\alpha t/\hbar)\Phi_\alpha(t), \tag{4.17}$$

where the coefficients c_α are determined by the initial wavefunction $\Psi(0) = \sum_\alpha c_\alpha \Psi_\alpha(0)$.

This formalism is useful for finding $\Psi(t)$ for a given $H(t)$ only if we can obtain the Floquet modes $\Phi_a(t)$ and quasienergies ϵ_α more easily than directly solving (4.15). By substituting (4.16) into the Schrödinger equation (4.15) we obtain an eigenvalue equation for the Floquet modes and quasienergies

$$\mathcal{H}(t)\Phi_\alpha(t) = \epsilon_\alpha \Phi_\alpha(t), \tag{4.18}$$

where $\mathcal{H}(t) = H(t) - i\hbar\partial_t$. This eigenvalue problem could be solved analytically or numerically, but in QuTiP we use an alternative approach for numerically finding the Floquet states and quasienergies [see e.g. Creffield et al., Phys. Rev. B 67, 165301 (2003)]. Consider the propagator for the time-dependent Schrödinger equation (4.15), which by definition satisfies

$$U(T+t, t)\Psi(t) = \Psi(T+t).$$

Inserting the Floquet states from (4.16) into this expression results in

$$U(T+t, t)\exp(-i\epsilon_\alpha t/\hbar)\Phi_\alpha(t) = \exp(-i\epsilon_\alpha(T+t)/\hbar)\Phi_\alpha(T+t),$$

or, since $\Phi_\alpha(T+t) = \Phi_\alpha(t)$,

$$U(T+t, t)\Phi_\alpha(t) = \exp(-i\epsilon_\alpha T/\hbar)\Phi_\alpha(t) = \eta_\alpha \Phi_\alpha(t),$$

which shows that the Floquet modes are eigenstates of the one-period propagator. We can therefore find the Floquet modes and quasienergies $\epsilon_\alpha = -\hbar \arg(\eta_\alpha)/T$ by numerically calculating $U(T+t, t)$ and diagonalizing it. In particular this method is useful to find $\Phi_\alpha(0)$ by calculating and diagonalize $U(T, 0)$.

The Floquet modes at arbitrary time t can then be found by propagating $\Phi_\alpha(0)$ to $\Phi_\alpha(t)$ using the wave function propagator $U(t, 0)\Psi_\alpha(0) = \Psi_\alpha(t)$, which for the Floquet modes yields

$$U(t, 0)\Phi_\alpha(0) = \exp(-i\epsilon_\alpha t/\hbar)\Phi_\alpha(t),$$

so that $\Phi_\alpha(t) = \exp(i\epsilon_\alpha t/\hbar)U(t, 0)\Phi_\alpha(0)$. Since $\Phi_\alpha(t)$ is periodic we only need to evaluate it for $t \in [0, T]$, and from $\Phi_\alpha(t \in [0, T])$ we can directly evaluate $\Phi_\alpha(t)$, $\Psi_\alpha(t)$ and $\Psi(t)$ for arbitrary large t.

Floquet formalism in QuTiP

QuTiP provides a family of functions to calculate the Floquet modes and quasi energies, Floquet state decomposition, etc., given a time-dependent Hamiltonian on the *callback format*, *list-string format* and *list-callback format* (see, e.g., `qutip.mesolve` for details).

Consider for example the case of a strongly driven two-level atom, described by the Hamiltonian

$$H(t) = -\frac{1}{2}\Delta\sigma_x - \frac{1}{2}\epsilon_0\sigma_z + \frac{1}{2}A\sin(\omega t)\sigma_z. \tag{4.19}$$

In QuTiP we can define this Hamiltonian as follows

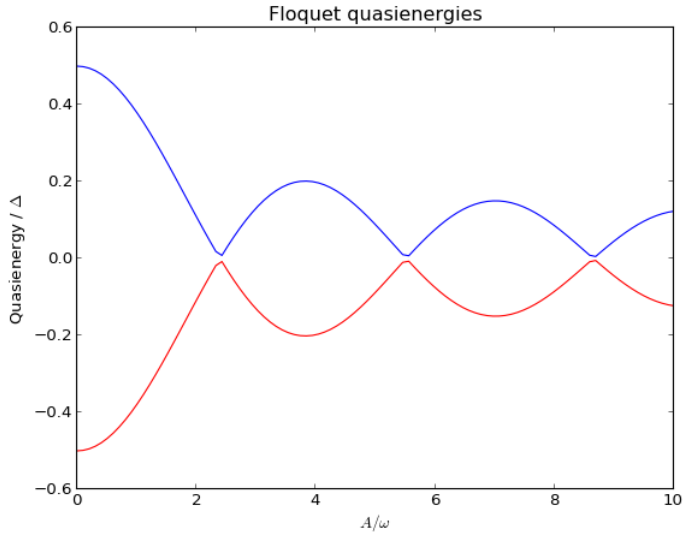
```
>>> delta = 0.2 * 2*pi; eps0 = 1.0 * 2*pi; A = 2.5 * 2*pi; omega = 1.0 * 2*pi
>>> H0 = - delta/2.0 * sigmax() - eps0/2.0 * sigmaz()
>>> H1 = A/2.0 * sigmaz()
>>> args = {'w': omega}
>>> H = [H0, [H1, 'sin(w * t)']]
```

The $t = 0$ Floquet modes corresponding to the Hamiltonian (4.19) can then be calculated using the `qutip.floquet.floquet_modes` function, which returns lists containing the Floquet modes and the quasienergies

```
>>> T = 2*pi / omega
>>> f_modes, f_energies = floquet_modes(H, T, args)
>>> f_energies
array([ 2.83131211, -2.83131211])
>>> f_modes0
[Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[ 0.39993745+0.554682j]
 [ 0.72964232+0.j      ]],
 Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[ 0.72964232+0.j      ]
 [-0.39993745+0.554682j]]]
```

For some problems interesting observations can be draw from the quasienergy levels alone. Consider for example the quasienergies for the driven two-level system introduced above as a function of the driving amplitude, calculated and plotted in the following example. For certain driving amplitudes the quasienergy levels cross. Since the the quasienergies can be associated with the time-scale of the long-term dynamics due that the driving, degenerate quasienergies indicates a "freezing" of the dynamics (sometimes known as coherent destruction of tunneling).

```
from qutip import *
from scipy import *

delta = 0.2 * 2*pi; eps0  = 0.0 * 2*pi
omega = 1.0 * 2*pi; A_vec = linspace(0, 10, 100) * omega;
T     = (2*pi)/omega
tlist  = linspace(0.0, 10 * T, 101)
psi0   = basis(2,0)

q_energies = zeros((len(A_vec), 2))

H0 = delta/2.0 * sigmaz() - eps0/2.0 * sigmax()
args = omega
for idx, A in enumerate(A_vec):
    H1 = A/2.0 * sigmax()
    H = [H0, [H1, lambda t, w: sin(w*t)]]
    f_modes,f_energies = floquet_modes(H, T, args, True)
    q_energies[idx,:] = f_energies

# plot the results
from pylab import *
plot(A_vec/omega, real(q_energies[:,0]) / delta, 'b', \
     A_vec/omega, real(q_energies[:,1]) / delta, 'r')
xlabel(r'$A/\omega$')
ylabel(r'Quasienergy / $\Delta$')
title(r'Floquet quasienergies')
show()
```



Given the Floquet modes at $t = 0$, we obtain the Floquet mode at some later time t using the function `qutip.floquet.floquet_mode_t`:

```
>>> f_modes_t = floquet_modes_t(f_modes_0, f_energies, 2.5, H, T, args)
>>> f_modes_t
[Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[-0.03189259+0.6830849j ]
 [-0.61110159+0.39866357j]],
 Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[-0.61110159-0.39866357j]
 [ 0.03189259+0.6830849j ]]]
```

The purpose of calculating the Floquet modes is to find the wavefunction solution to the original problem (4.19) given some initial state $|\psi_0\rangle$. To do that, we first need to decompose the initial state in the Floquet states, using the function `qutip.floquet.floquet_state_decomposition`

```
>>> psi0 = rand_ket(2)
>>> f_coeff = floquet_state_decomposition(f_modes_0, f_energies, psi0)
[(0.81334464307183041-0.15802444453870021j),
 (-0.17549465805005662-0.53169576969399113j)]
```

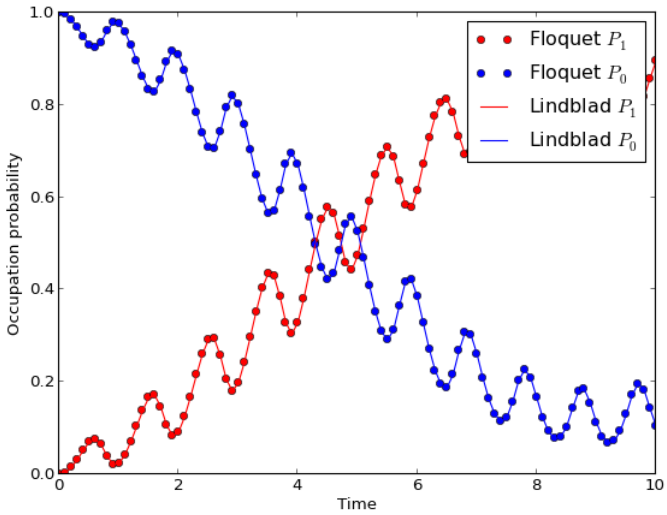
and given this decomposition of the initial state in the Floquet states we can easily evaluate the wavefunction that is the solution to (4.19) at an arbitrary time t using the function `qutip.floquet.floquet_wavefunction_t`

```
>>> t = 10 * rand()
>>> psi_t = floquet_wavefunction_t(f_modes_0, f_energies, f_coeff, t, H, T, args)
>>> psi_t
Quantum object: dims = [[2], [1]], shape = [2, 1], type = ket
Qobj data =
[[-0.29352582+0.84431304j]
 [ 0.30515868+0.32841589j]]
```

The following example illustrates how to use the functions introduced above to calculate and plot the time-evolution of (4.19).

```
from qutip import *
from scipy import *

delta = 0.2 * 2*pi; eps0  = 1.0 * 2*pi
A     = 0.5 * 2*pi; omega = 1.0 * 2*pi
```

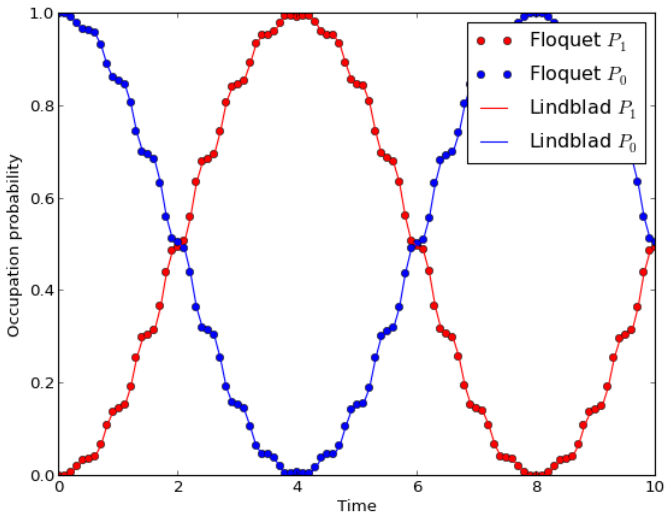
```
T      = (2*pi)/omega
tlist  = linspace(0.0, 10 * T, 101)
psi0   = basis(2,0)

H0 = - delta/2.0 * sigmax() - eps0/2.0 * sigmaz()
H1 = A/2.0 * sigmaz()
args = {'w': omega}
H = [H0, [H1, lambda t,args: sin(args['w'] * t)]]

# find the floquet modes for the time-dependent hamiltonian
f_modes_0,f_energies = floquet_modes(H, T, args)

# decompose the inital state in the floquet modes
f_coeff = floquet_state_decomposition(f_modes_0, f_energies, psi0)

# calculate the wavefunctions using the from the floquet modes
p_ex = zeros(len(tlist))
for n, t in enumerate(tlist):
    psi_t = floquet_wavefunction_t(f_modes_0, f_energies, f_coeff, t, H, T, args)
    p_ex[n] = expect(num(2), psi_t)

# For reference: calculate the same thing with mesolve
p_ex_ref = mesolve(H, psi0, tlist, [], [num(2)], args).expect[0]

# plot the results
from pylab import *
plot(tlist, real(p_ex),     'ro', tlist, 1-real(p_ex),     'bo')
plot(tlist, real(p_ex_ref), 'r',  tlist, 1-real(p_ex_ref), 'b')
xlabel('Time')
ylabel('Occupation probability')
legend(("Floquet $P_1$", "Floquet $P_0$", "Lindblad $P_1$", "Lindblad $P_0$"))
show()
```



Pre-computing the Floquet modes for one period

When evaluating the Floquet states or the wavefunction at many points in time it is useful to pre-compute the Floquet modes for the first period of the driving with the required resolution. In QuTiP the function `qutip.floquet.floquet_modes_table` calculates a table of Floquet modes which later can be used together with the function `qutip.floquet.floquet_modes_t_lookup` to efficiently lookup the Floquet mode at an arbitrary time. The following example illustrates how the example from the previous section can be solved more efficiently using these functions for pre-computing the Floquet modes.

```
from qutip import *
from scipy import *

delta = 0.0  * 2*pi; eps0  = 1.0 * 2*pi
A     = 0.25 * 2*pi; omega = 1.0 * 2*pi
T      = (2*pi)/omega
tlist  = linspace(0.0, 10 * T, 101)
psi0   = basis(2,0)

H0 = - delta/2.0 * sigmax() - eps0/2.0 * sigmaz()
H1 = A/2.0 * sigmax()
args = {'w': omega}
H = [H0, [H1, lambda t,args: sin(args['w'] * t)]]

# find the floquet modes for the time-dependent hamiltonian
f_modes_0,f_energies = floquet_modes(H, T, args)

# decompose the inital state in the floquet modes
f_coeff = floquet_state_decomposition(f_modes_0, f_energies, psi0)

# calculate the wavefunctions using the from the floquet modes
f_modes_table_t = floquet_modes_table(f_modes_0, f_energies, tlist, H, T, args)
p_ex = zeros(len(tlist))
for n, t in enumerate(tlist):
    f_modes_t = floquet_modes_t_lookup(f_modes_table_t, t, T)
    psi_t     = floquet_wavefunction(f_modes_t, f_energies, f_coeff, t)
    p_ex[n] = expect(num(2), psi_t)

# For reference: calculate the same thing with mesolve
p_ex_ref = mesolve(H, psi0, tlist, [], [num(2)], args).expect[0]

# plot the results
from pylab import *
plot(tlist, real(p_ex),     'ro', tlist, 1-real(p_ex),     'bo')
plot(tlist, real(p_ex_ref), 'r',  tlist, 1-real(p_ex_ref), 'b')
xlabel('Time')
ylabel('Occupation probability')
legend(("Floquet $P_1$", "Floquet $P_0$", "Lindblad $P_1$", "Lindblad $P_0$"))
show()
```



Note that the parameters and the Hamiltonian used in this example is not the same as in the previous section, and hence the different appearance of the resulting figure.

For convenience, all the steps described above for calculating the evolution of a quantum system using the Floquet formalisms are encapsulated in the function `qutip.floquet.fsesolve`. Using this function, we could have achieved the same results as in the examples above using:

```
output = fsesolve(H, psi0, tlist, [num(2)], args)
p_ex = output.expect[0]
```

Floquet theory for dissipative evolution

A driven system that is interacting with its environment is not necessarily well described by the standard Lindblad master equation, since its dissipation process could be time-dependent due to the driving. In such cases a rigorious approach would be to take the driving into account when deriving the master equation. This can be done in many different ways, but one way common approach is to derive the master equation in the Floquet basis. That approach results in the so-called Floquet-Markov master equation, see Grifoni et al., Physics Reports 304, 299 (1998) for details.

The Floquet-Markov master equation in QuTiP

The QuTiP function `qutip.floquet.fmmesolve` implements the Floquet-Markov master equation. It calculates the dynamics of a system given its initial state, a time-dependent hamiltonian, a list of operators through which the system couples to its environment and a list of corresponding spectral-density functions that describes the environment. In contrast to the `qutip.mesolve` and `qutip.mcsolve`, and the `qutip.floquet.fmmesolve` does characterize the environment with dissipation rates, but extract the strength of the coupling to the environment from the noise spectral-density functions and the instantaneous Hamiltonian parameters (similar to the Bloch-Redfield master equation solver `qutip.bloch_redfield.brmesolve`).

Note: Currently the `qutip.floquet.fmmesolve` can only accept a single environment coupling operator and spectral-density function.

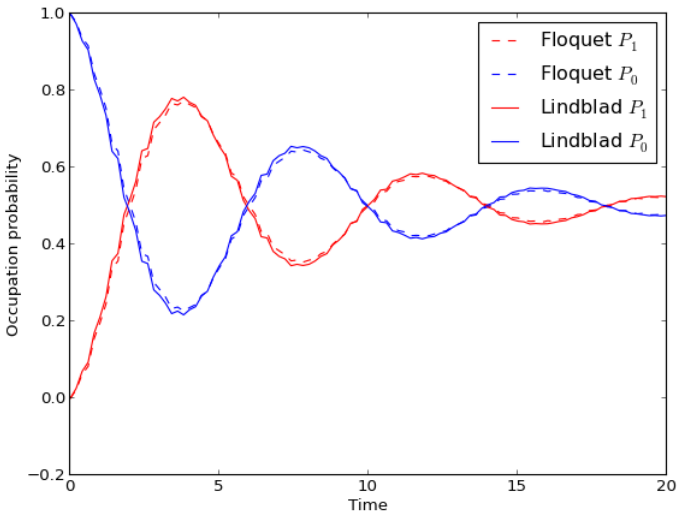
The noise spectral-density function of the environment is implemented as a Python callback function that is passed to the solver. For example:

```
>>> gamma1 = 0.1
>>> def noise_spectrum(omega):
>>>     return 0.5 * gamma1 * omega/(2*pi)
```

The other parameters are similar to the `qutip.mesolve` and `qutip.mcsolve`, and the same format for the return value is used `qutip.Odedata`. The following example extends the example studied above, and uses `qutip.floquet.fmmesolve` to introduce dissipation into the calculation

```
from qutip import *
from scipy import *

delta = 0.0  * 2*pi; eps0  = 1.0 * 2*pi
A     = 0.25 * 2*pi; omega = 1.0 * 2*pi
T      = (2*pi)/omega
tlist  = linspace(0.0, 20 * T, 101)
psi0   = basis(2,0)

H0 = - delta/2.0 * sigmax() - eps0/2.0 * sigmaz()
H1 = A/2.0 * sigmax()
args = {'w': omega}
H = [H0, [H1, lambda t,args: sin(args['w'] * t)]]

# noise power spectrum
gamma1 = 0.1
def noise_spectrum(omega):
```

```
    return 0.5 * gamma1 * omega/(2*pi)

# find the floquet modes for the time-dependent hamiltonian
f_modes_0, f_energies = floquet_modes(H, T, args)

# precalculate mode table
f_modes_table_t = floquet_modes_table(f_modes_0, f_energies,
                                      linspace(0, T, 500 + 1), H, T, args)

# solve the floquet-markov master equation
output = fmmesolve(H, psi0, tlist, [sigmax()], [], [noise_spectrum], T, args)

# calculate expectation values in the computational basis
p_ex = zeros(shape(tlist), dtype=complex)
for idx, t in enumerate(tlist):
    f_modes_t = floquet_modes_t_lookup(f_modes_table_t, t, T)
    p_ex[idx] = expect(num(2), output.states[idx].transform(f_modes_t, False))

# For reference: calculate the same thing with mesolve
output = mesolve(H, psi0, tlist, [sqrt(gamma1) * sigmax()], [num(2)], args)
p_ex_ref = output.expect[0]

# plot the results
from pylab import *
plot(tlist, real(p_ex), 'r--', tlist, 1-real(p_ex), 'b--')
plot(tlist, real(p_ex_ref), 'r', tlist, 1-real(p_ex_ref), 'b')
xlabel('Time')
ylabel('Occupation probability')
legend(("Floquet $P_1$", "Floquet $P_0$", "Lindblad $P_1$", "Lindblad $P_0$"))
show()
```



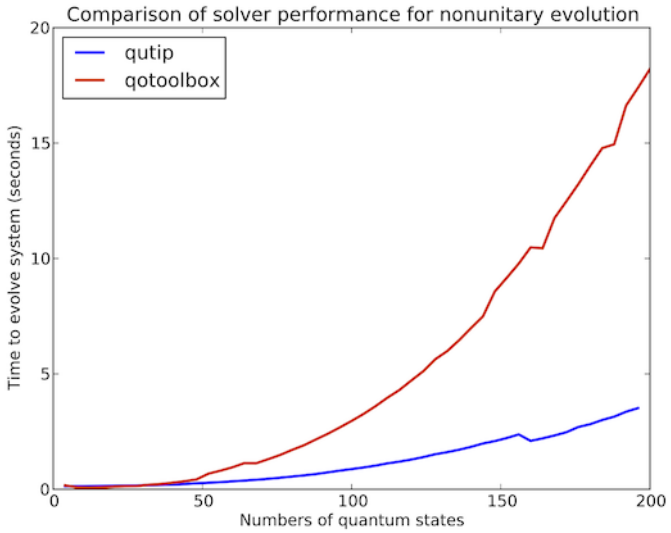
Alternatively, we can let the `qutip.floquet.fmmesolve` function transform the density matrix at each time step back to the computational basis, and calculating the expectation values for us, but using:

```
output = fmmesolve(H, psi0, tlist, [sigmax()], [num(2)], [noise_spectrum], T, args)
p_ex = output.expect[0]
```

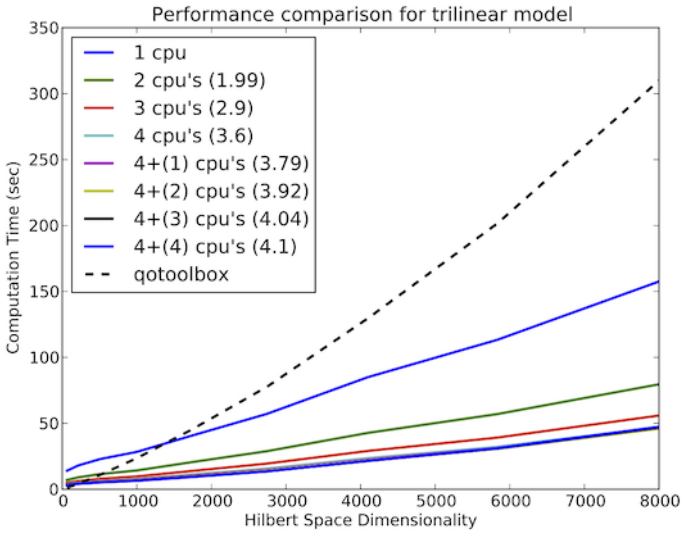
4.5.7 Performance (QuTiP vs. qotoolbox)

Here we compare the performance of the master-equation and Monte Carlo solvers to their quantum optics toolbox counterparts.

In this example, we calculate the time-evolution of the density matrix for a coupled oscillator system using the `qutip.mesolve` function, and compare it to the quantum optics toolbox (qotoolbox). Here, we see that the QuTiP solver out performs its qotoolbox counterpart by a substantial margin as the system size increases.



To test the Monte Carlo solvers, here we simulate a trilinear Hamiltonian over a range of Hilbert space sizes. Since QuTiP uses multiprocessing, we can measure the performance gain when using several CPU's. In contrast, the qotoolbox is limited to a single processor only. In the legend, we show the speed-up factor in the parenthesis, which should ideally be equal to the number of processors. Finally, we have included the results using hyperthreading, written here as 4+(x) where x is the number of hyperthreads, found in some newer Intel processors. We see however that the performance benefits from hyperthreading are marginal at best.



4.5.8 Setting Options for the Dynamics ODE Solvers

Occasionally it is necessary to change the built in parameters of the ODE solvers used by both the `qutip.mesolve` and `qutip.mcsolve` functions. The ODE options for either of these functions may be changed by calling the Odeoptions class `qutip.Odeoptions`

```
>>> opts = Odeoptions()
```

the properties and default values of this class can be view via the *print* function:

```
>>> print(opts)
    Odeoptions properties:
    ----------------------
    atol:          1e-08
    rtol:          1e-06
    method:        adams
    order:         12
    nsteps:        1000
    first_step:    0
    min_step:      0
    max_step:      0
    tidy:          True
    num_cpus:      8
    rhs_filename:  None
    rhs_reuse:     False
    gui:           True
    mc_avg:            True
```

These properties are detailed in the following table. Assuming `opts = Odeoptions()`:

Property	Default setting	Description
opts.atol	1e-8	Absolute tolerance
opts.rtol	1e-6	Relative tolerance
opts.method	'adams'	Solver method. Can be 'adams' (non-stiff) or 'bdf' (stiff)
opts.order	12	Order of solver. Must be <=12 for 'adams' and <=5 for 'bdf'
opts.nsteps	1000	Max. number of steps to take for each interval
opts.first_step	0	Size of initial step. 0 = determined automatically by solver.
opts.min_step	0	Minimum step size. 0 = determined automatically by solver.
opts.max_step	0	Maximum step size. 0 = determined automatically by solver.
opts.tidy	True	Whether to run tidyup function on time-independent Hamiltonian.
opts.num_cpus	installed num of processors	Integer number of cpu's used by mcsolve.
opts.rhs_filename	None	RHS filename when using compiled time-dependent Hamiltonians.
opts.rhs_reuse	False	Reuse compiled RHS function. Useful for repeatative tasks.
opts.gui	True (if GUI)	Use the mcsolve progessbar. Defaults to False on Windows.
opts.mc_avg	True	Average over trajectories for expectation values from mcsolve.

As an example, let us consider changing the number of processors used, turn the GUI off, and strengthen the absolute tolerance. There are two equivalent ways to do this using the Odeoptions class. First way,

```
>>> opts = Odeoptions()
>>> opts.num_cpus = 3
>>> opts.gui = False
>>> opts.atol = 1e-10
```

or one can use an inline method,

```
>>> opts = Odeoptions(num_cpus=3, gui=False, atol=1e-10)
```

Note that the order in which you input the options does not matter. Using either method, the resulting *opts* variable is now:

```
>>> print opts
Odeoptions properties:
----------------------
atol:          1e-10
rtol:          1e-06
method:        adams
order:         12
nsteps:        1000
```

```
first_step:     0
min_step:       0
max_step:       0
tidy:           True
num_cpus:       3
rhs_filename:   None
rhs_reuse:      False
gui:            False
mc_avg:     True
```

To use these new settings we can use the keyword argument `options` in either the func:*qutip.mesolve* and `qutip.mcsolve` function. We can modify the last example as:

```
>>> mesolve(H0, psi0, tlist, c_op_list, [sigmaz()], options=opts)
>>> mesolve(hamiltonian_t, psi0, tlist, c_op_list, [sigmaz()], H_args, options=opts)
```

or:

```
>>> mcsolve(H0, psi0, tlist, ntraj,c_op_list, [sigmaz()], options=opts)
>>> mcsolve(hamiltonian_t, psi0, tlist, ntraj, c_op_list, [sigmaz()], H_args, options=opts)
```

4.6 Solving for Steady-State Solutions

4.6.1 Introduction

For open quantum systems with decay rates larger than the corresponding excitation rate, the system will tend toward a steady-state as $t \to \infty$ that satisfies the equation

$\frac{\partial \rho_{ss}}{\partial t} = \mathcal{L}\rho_{ss} = 0.$

For many these systems, solving for the asymptotic density matrix ρ_{ss} can be achieved using an iterative method faster than master equation or Monte Carlo simulations. In QuTiP, the steady-state solution for a system Hamiltonian or Liouvillian is given by `qutip.steady.steadystate` or `qutip.steady.steady`, respectively. Both of these functions use a shifted inverse power method with a random initial state that finds the zero eigenvalue. In general, it is best to use the `qutip.steady.steadystate` function with a given Hamiltonian and list of collapse operators. This function will automatically build the Louivillian for you and then call the `qutip.steady.steady` function.

4.6.2 Using the Steadystate Solver

A general call to the steady-state solver `qutip.steady.steadystate` may be accomplished using the command:

```
>>> rho_ss = steadystate(H, c_op_list)
```

where `H` is a quantum object representing the system Hamiltonian, and `c_op_list` is a list of quantum objects for the system collapse operators. The output, labeled as `rho_ss`, is the steady-state solution for the system dynamics. Although this method will produce the required solution in most situations, there are additional options that may be set if the algorithm does not converge properly. These optional parameters may be used by calling the steady-state solver as:

```
>>> rho_ss = steadystate(H, c_op_list, maxiter, tol, method, use_umfpack, use_precond)
```

where `maxiter` is the maximum number of iterations performed by the solver, `tol` is the requested solution tolerance, and `method` indicates whether the linear equation solver uses a direct solver "solve" or an iterative stabilized bi-conjugate gradient "bicg" solution method. In general, the direct solver is faster, but takes more memory than the iterative method. The advantage of the iterative method is the memory efficiency of this routine, allowing for extremely large system sizes. The downside is that it takes much longer than the direct method, especially when the condition number of the Liouvillian matrix is large, as typically occurs. To overcome this, the

steady state solvers also include a preconditioner that attempts to normalize the condition number of the system. This incomplete LU preconditioner is invoked by using the "use_precond=True" flag in combination with the iterative solver. As a first try, it is recommended to begin with the direct solver before using the iterative `bicg` method. More information on these options may be found in the `qutip.steady.steadystate` section of the API.

This solver can also use a Liouvillian constructed from a Hamiltonian and collapse operators as the input variable when called using the `qutip.steady.steady` function:

```
>>> rho_ss = steady(L)
```

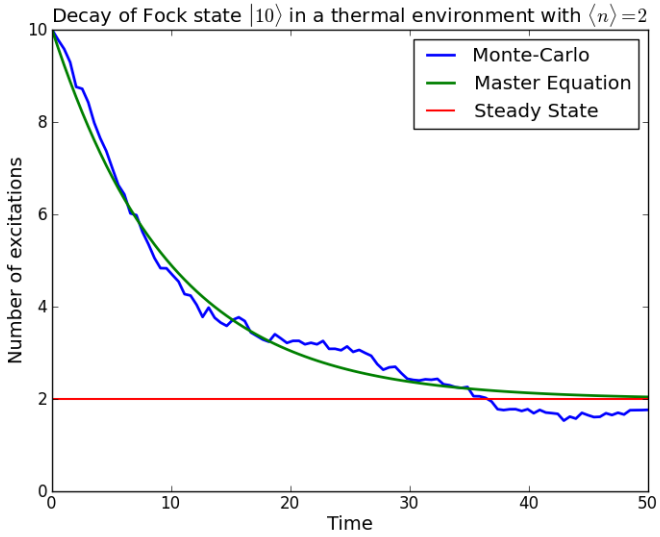
where `L` is the Louvillian. This function also takes the previously mentioned optional parameters. We do however recommend using the `qutip.steady.steadystate` function if you are using a standard Liouvillian as it will automatically build the system Liouvillian and call `qutip.steady.steady` for you.

4.6.3 Example: Harmonic Oscillator in Thermal Bath

A simple example of a system that reaches a steady state is a harmonic oscillator coupled to a thermal environment. Below we consider a harmonic oscillator, initially in a $|10\rangle$ number state, and weakly coupled to a thermal environment characterized by an average particle expectation value of $\langle n \rangle = 2$. We calculate the evolution via master equation and Monte Carlo methods, and see that they converge to the steady-state solution. Here we choose to perform only a few Monte Carlo trajectories so we can distinguish this evolution from the master-equation solution.

```
from qutip import *
from pylab import *
from scipy import *

# Define paramters
N = 20  # number of basis states to consider
a = destroy(N)
H = a.dag() * a
psi0 = basis(N, 10)  # initial state
kappa = 0.1  # coupling to oscillator

# collapse operators
c_op_list = []
n_th_a = 2  # temperature with average of 2 excitations
rate = kappa * (1 + n_th_a)
if rate > 0.0:
    c_op_list.append(sqrt(rate) * a)  # decay operators
rate = kappa * n_th_a
if rate > 0.0:
    c_op_list.append(sqrt(rate) * a.dag())  # excitation operators

# find steady-state solution
final_state = steadystate(H, c_op_list)
# find expectation value for particle number in steady state
fexpt = expect(a.dag() * a, final_state)

tlist = linspace(0, 50, 100)
# monte-carlo
mcdata = mcsolve(H, psi0, tlist, c_op_list, [a.dag() * a], ntraj=100)
# master eq.
medata = mesolve(H, psi0, tlist, c_op_list, [a.dag() * a])

plot(tlist, mcdata.expect[0], tlist, medata.expect[0], lw=2)
# plot steady-state expt. value as horizontal line (should be = 2)
axhline(y=fexpt, color='r', lw=1.5)
ylim([0, 10])
xlabel('Time', fontsize=14)
ylabel('Number of excitations', fontsize=14)
```

```
legend(('Monte-Carlo', 'Master Equation', 'Steady State'))
title('Decay of Fock state $\left|10\\rangle\\right.$' +
      ' in a thermal environment with $\langle n\\rangle=2$')
show()
```



4.7 An Overview of the Eseries Class

4.7.1 Exponential-series representation of time-dependent quantum objects

The eseries object in QuTiP is a representation of an exponential-series expansion of time-dependent quantum objects (a concept borrowed from the quantum optics toolbox).

An exponential series is parameterized by its amplitude coefficients c_i and rates r_i, so that the series takes the form $E(t) = \sum_i c_i e^{r_i t}$. The coefficients are typically quantum objects (type Qobj: states, operators, etc.), so that the value of the eseries also is a quantum object, and the rates can be either real or complex numbers (describing decay rates and oscillation frequencies, respectively). Note that all amplitude coefficients in an exponential series must be of the same dimensions and composition.

In QuTiP, an exponential series object is constructed by creating an instance of the class `qutip.eseries`:

```
In [1]: es1 = eseries(sigmax(), 1j)
```

where the first argument is the amplitude coefficient (here, the sigma-X operator), and the second argument is the rate. The eseries in this example represents the time-dependent operator $\sigma_x e^{it}$.

To add more terms to an `qutip.eseries` object we simply add objects using the + operator:

```
In [1]: omega=1.0

In [2]: es2 = eseries(0.5 * sigmax(), 1j * omega) + eseries(0.5 * sigmax(), -1j * omega)
```

The `qutip.eseries` in this example represents the operator $0.5\sigma_x e^{i\omega t} + 0.5\sigma_x e^{-i\omega t}$, which is the exponential series representation of $\sigma_x \cos(\omega t)$. Alternatively, we can also specify a list of amplitudes and rates when the `qutip.eseries` is created:

```
In [1]: es2 = eseries([0.5 * sigmax(), 0.5 * sigmax()], [1j * omega, -1j * omega])
```

We can inspect the structure of an `qutip.eseries` object by printing it to the standard output console:

```
In [1]: es2
Out[1]: ESERIES object: 2 terms
Hilbert space dimensions: [[2], [2]]
Exponent #0 = -1j
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[ 0.   0.5]
 [ 0.5  0. ]]
Exponent #1 = 1j
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[ 0.   0.5]
 [ 0.5  0. ]]
```

and we can evaluate it at time t by using the `qutip.eseries.esval` function:

```
In [1]: esval(es2, 0.0)        # equivalent to es2.value(0.0)
Out[1]:
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[ 0.  1.]
 [ 1.  0.]]
```

or for a list of times `[0.0, 1.0 * pi, 2.0 * pi]`:

```
In [1]: tlist = [0.0, 1.0 * pi, 2.0 * pi]

In [2]: esval(es2, tlist)     # equivalent to es2.value(tlist)
Out[2]:
array([ Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[ 0.  1.]
 [ 1.  0.]],
       Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[ 0. -1.]
 [-1.  0.]],
       Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[ 0.  1.]
 [ 1.  0.]]], dtype=object)
```

To calculate the expectation value of an time-dependent operator represented by an `qutip.eseries`, we use the `qutip.expect` function. For example, consider the operator $\sigma_x \cos(\omega t) + \sigma_z \sin(\omega t)$, and say we would like to know the expectation value of this operator for a spin in its excited state (`rho = fock_dm(2,1)` produce this state):

```
In [1]: es3 = eseries([0.5*sigmaz(), 0.5*sigmaz()], [1j, -1j]) + eseries([-0.5j*sigmax(), 0.5j*si

In [2]: rho = fock_dm(2, 1)

In [3]: es3_expect = expect(rho, es3)

In [4]: es3_expect
Out[4]: ESERIES object: 2 terms
Hilbert space dimensions: [[1, 1]]
Exponent #0 = -1j
(-0.5+0j)
Exponent #1 = 1j
(-0.5+0j)


In [5]: es3_expect.value([0.0, pi/2])
Out[5]: array([ -1.00000000e+00,  -6.12323400e-17])
```

Note the expectation value of the `qutip.eseries` object, `expect(rho, es3)`, itself is an `qutip.eseries`, but with amplitude coefficients that are C-numbers instead of quantum operators. To evaluate the C-number `qutip.eseries` at the times *tlist* we use `esval(es3_expect, tlist)`, or, equivalently, `es3_expect.value(tlist)`.

4.7.2 Applications of exponential series

The exponential series formalism can be useful for the time-evolution of quantum systems. One approach to calculating the time evolution of a quantum system is to diagonalize its Hamiltonian (or Liouvillian, for dissipative systems) and to express the propagator (e.g., $\exp(-iHt)\rho\exp(iHt)$) as an exponential series.

The QuTiP function `qutip.essolve.ode2es` and `qutip.essolve` use this method to evolve quantum systems in time. The exponential series approach is particularly suitable for cases when the same system is to be evolved for many different initial states, since the diagonalization only needs to be performed once (as opposed to e.g. the ode solver that would need to be ran independently for each initial state).

As an example, consider a spin-1/2 with a Hamiltonian pointing in the σ_z direction, and that is subject to noise causing relaxation. For a spin originally is in the up state, we can create an `qutip.eseries` object describing its dynamics by using the `qutip.es2ode` function:

```
In [1]: psi0 = basis(2,1)

In [2]: H = sigmaz()

In [3]: L = liouvillian(H, [sqrt(1.0) * destroy(2)])

In [4]: es = ode2es(L, psi0)
```

The `qutip.essolve.ode2es` function diagonalizes the Liouvillian L and creates an exponential series with the correct eigenfrequencies and amplitudes for the initial state ψ_0 (*psi0*).

We can examine the resulting `qutip.eseries` object by printing a text representation:

```
In [1]: es
Out[1]: ESERIES object: 2 terms
Hilbert space dimensions: [[2], [2]]
Exponent #0 = (-1+0j)
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[-1.  0.]
 [ 0.  1.]]
Exponent #1 = 0j
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[ 1.  0.]
 [ 0.  0.]]
```

or by evaluating it and arbitrary points in time (here at 0.0 and 1.0):

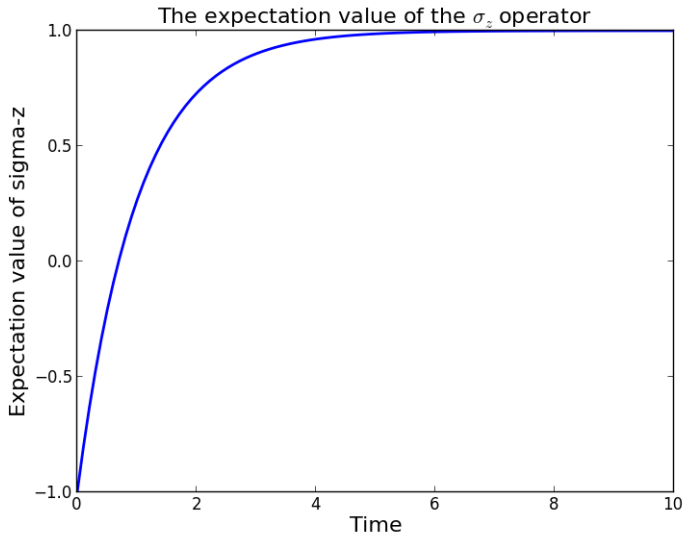
```
In [1]: es.value([0.0, 1.0])
Out[1]:
array([ Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[ 0.  0.]
 [ 0.  1.]],
       Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isherm = True
Qobj data =
[[ 0.63212056  0.        ]
 [ 0.          0.36787944]]], dtype=object)
```

and the expectation value of the exponential series can be calculated using the `qutip.expect` function:

```
In [1]: es_expect = expect(sigmaz(), es)
```

The result *es_expect* is now an exponential series with c-numbers as amplitudes, which easily can be evaluated at arbitrary times:

```
In [1]: es_expect.value([0.0, 1.0, 2.0, 3.0])
Out[1]: array([-1.        ,  0.26424112,  0.72932943,  0.90042586])

In [1]: tlist = linspace(0.0, 10.0, 100)

In [2]: sz_expect = es_expect.value(tlist)

In [3]: from pylab import *

In [4]: plot(tlist, sz_expect, lw=2);

In [5]: xlabel("Time", fontsize=16); ylabel("Expectation value of sigma-z", fontsize=16);

In [6]: title("The expectation value of the $\sigma_{z}$ operator", fontsize=16);
```



4.8 Two-time correlation functions

With the QuTiP time-evolution functions (for example `qutip.mesolve` and `qutip.mcsolve`), a state vector or density matrix can be evolved from an initial state at t_0 to an arbitrary time t, $\rho(t) = V(t, t_0)\left\{\rho(t_0)\right\}$, where $V(t, t_0)$ is the propagator defined by the equation of motion. The resulting density matrix can then be used to evaluate the expectation values of arbitrary combinations of *same-time* operators.

To calculate *two-time* correlation functions on the form $\langle A(t+\tau)B(t)\rangle$, we can use the quantum regression theorem [see, e.g., Gardineer and Zoller, *Quantum Noise*, Springer, 2004] to write

$$\langle A(t+\tau)B(t)\rangle = \mathrm{Tr}\left[AV(t+\tau,t)\left\{B\rho(t)\right\}\right] = \mathrm{Tr}\left[AV(t+\tau,t)\left\{BV(t,0)\left\{\rho(0)\right\}\right\}\right]$$

We therefore first calculate $\rho(t) = V(t,0)\left\{\rho(0)\right\}$ using one of the QuTiP evolution solvers with $\rho(0)$ as initial state, and then again use the same solver to calculate $V(t+\tau,t)\left\{B\rho(t)\right\}$ using $B\rho(t)$ as initial state.

Note that if the intial state is the steady state, then $\rho(t) = V(t,0)\left\{\rho_{\mathrm{ss}}\right\} = \rho_{\mathrm{ss}}$ and

$$\langle A(t+\tau)B(t)\rangle = \mathrm{Tr}\left[AV(t+\tau,t)\left\{B\rho_{\mathrm{ss}}\right\}\right] = \mathrm{Tr}\left[AV(\tau,0)\left\{B\rho_{\mathrm{ss}}\right\}\right] = \langle A(\tau)B(0)\rangle\,,$$

which is independent of t, so that we only have one time coordinate τ.

QuTiP provides a family of functions that assists in the process of calculating two-time correlation functions. The available functions and their usage is show in the table below. Each of these functions can use one of the following evolution solvers: Master-equation, Exponential series and the Monte-Carlo. The choice of solver is defined by the optional argument `solver`.

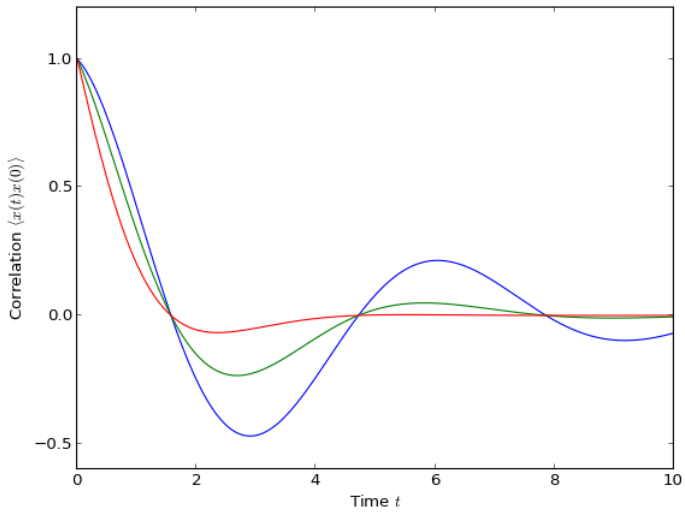
QuTiP function	Correlation function
qutip.correlation.correlation_2op_2t or qutip.correlation.correlation_2op_2t	$\langle A(t+\tau)B(t)\rangle$ or $\langle A(t)B(t+\tau)\rangle$.
qutip.correlation.correlation_2op_1t or qutip.correlation.correlation_2op_1t	$\langle A(\tau)B(0)\rangle$ or $\langle A(0)B(\tau)\rangle$.
qutip.correlation.correlation_4op_1t	$\langle A(0)B(\tau)C(\tau)D(0)\rangle$.
qutip.correlation.correlation_4op_2t	$\langle A(t)B(t+\tau)C(t+\tau)D(t)\rangle$.

The most common use-case is to calculate correlation functions of the kind $\langle A(\tau)B(0)\rangle$, in which case we use the correlation function solvers that start from the steady state, e.g., the `qutip.correlation.correlation_2op_1t` function. These correlation function sovlers return a vector or matrix (in general complex) with the correlations as a function of the delays times.

4.8.1 Steadystate correlation function

The following code demonstrates how to calculate the $\langle x(t)x(0)\rangle$ correlation for a leaky cavity with three different relaxation rates.

```
from qutip import *
from scipy import *

tlist = linspace(0,10.0,200)
a = destroy(10)
x = a.dag() + a
H = a.dag() * a

corr1 = correlation_ss(H, tlist, [sqrt(0.5) * a], x, x)
corr2 = correlation_ss(H, tlist, [sqrt(1.0) * a], x, x)
corr3 = correlation_ss(H, tlist, [sqrt(2.0) * a], x, x)

from pylab import *
plot(tlist, real(corr1), tlist, real(corr2), tlist, real(corr3))
xlabel(r'Time $t$')
ylabel(r'Correlation $\left<x(t)x(0)\right>$')
show()
```



4.8.2 Emission spectrum

Given a correlation function $\langle A(\tau)B(0)\rangle$ we can define the corresponding power spectrum as

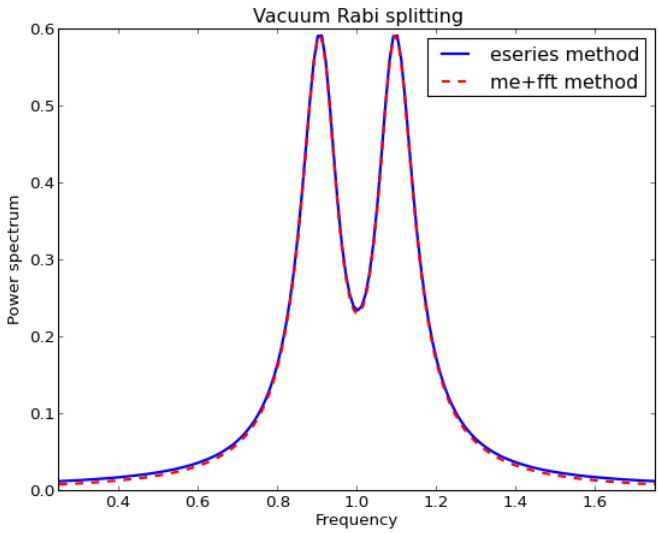
$$S(\omega) = \int_{-\infty}^{\infty} \langle A(\tau)B(0)\rangle\, e^{-i\omega\tau} d\tau.$$

In QuTiP, we can calculate $S(\omega)$ using either `qutip.correlation.spectrum_ss`, which first calculates the correlation function using the `qutip.essolve.essolve` solver and then performs the Fourier transform semi-analytically, or we can use the function `qutip.correlation.spectrum_correlation_fft` to numerically calculate the Fourier transform of a given correlation data using FFT.

The following example demonstrates how these two functions can be used to obtain the emission power spectrum.

```
from qutip import *
import pylab as plt
from scipy import *
from scipy import *
N = 4                       # number of cavity fock states
wc = wa = 1.0 * 2 * pi      # cavity and atom frequency
g  = 0.1 * 2 * pi           # coupling strength
kappa = 0.75                # cavity dissipation rate
gamma = 0.25                # atom dissipation rate

# Jaynes-Cummings Hamiltonian
a  = tensor(destroy(N), qeye(2))
sm = tensor(qeye(N), destroy(2))
H = wc * a.dag() * a + wa * sm.dag() * sm + g * (a.dag() * sm + a * sm.dag())

# collapse operators
n_th = 0.25
c_ops = [sqrt(kappa * (1 + n_th)) * a, sqrt(kappa * n_th) * a.dag(), sqrt(gamma) * sm]

# calculate the correlation function using the mesolve solver, and then fft to
# obtain the spectrum. Here we need to make sure to evaluate the correlation
# function for a sufficient long time and sufficiently high sampling rate so
# that the discrete Fourier transform (FFT) captures all the features in the
# resulting spectrum.
tlist = linspace(0, 100, 5000)
corr = correlation_ss(H, tlist, c_ops, a.dag(), a)
wlist1, spec1 = spectrum_correlation_fft(tlist, corr)
```

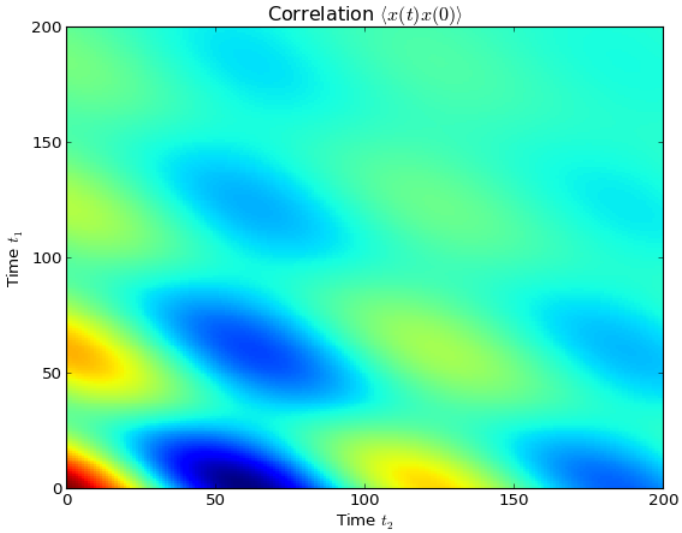
```
# calculate the power spectrum using spectrum_ss, which internally uses essolve
# to solve for the dynamics
wlist2 = linspace(0.25, 1.75, 200) * 2 * pi
spec2 = spectrum_ss(H, wlist2, c_ops, a.dag(), a)

# plot the spectra
fig, ax = plt.subplots(1, 1)
ax.plot(wlist1 / (2 * pi), spec1, 'b', lw=2, label='eseries method')
ax.plot(wlist2 / (2 * pi), spec2, 'r--', lw=2, label='me+fft method')
ax.legend()
ax.set_xlabel('Frequency')
ax.set_ylabel('Power spectrum')
ax.set_title('Vacuum Rabi splitting')
ax.set_xlim(wlist2[0]/(2*pi), wlist2[-1]/(2*pi))
plt.show()
```



4.8.3 Non-steadystate correlation function

More generally, we can also calculate correlation functions of the kind $\langle A(t_1 + t_2)B(t_1)\rangle$, i.e., the correlation function of a system that is not in its steadystate. In QuTiP, we can evoluate such correlation functions using the function `qutip.correlation.correlation`. The default behavior of this function is to return a matrix with the correlations as a function of the two time coordinates (t_1 and t_2).

```
from qutip import *
from scipy import *

tlist = linspace(0, 10.0, 200)
a = destroy(10)
x = a.dag() + a
H = a.dag() * a
alpha = 2.5
rho0 = coherent_dm(10, alpha)
corr = correlation(H, rho0, tlist, tlist, [sqrt(0.25) * a], x, x)

from pylab import *
pcolor(corr)
xlabel(r'Time $t_2$')
ylabel(r'Time $t_1$')
title(r'Correlation $\left<x(t)x(0)\right>$')
show()
```



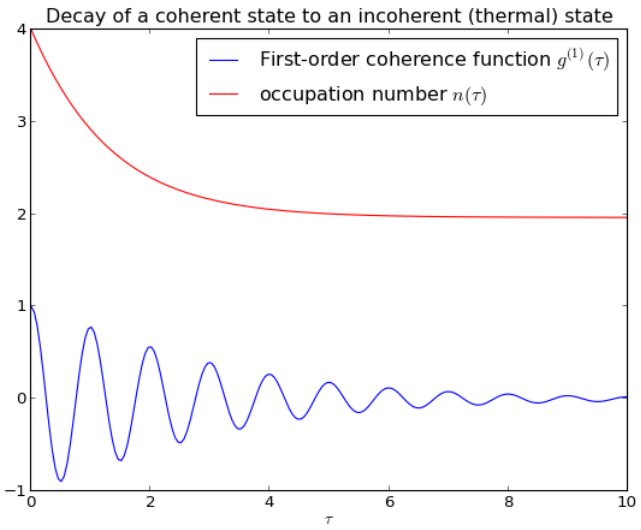
However, in some cases we might be interested in the correlation functions on the form $\langle A(t_1+t_2)B(t_1)\rangle$, but only as a function of time coordinate t_2. In this case we can also use the `qutip.correlation.correlation` function, if we pass the density matrix at time t_1 as second argument, and *None* as third argument. The `qutip.correlation.correlation` function then returns a vector with the correlation values corresponding to the times in *taulist* (the fourth argument).

Example: first-order optical coherence function

This example demonstrates how to calculate a correlation function on the form $\langle A(\tau)B(0)\rangle$ for a non-steady initial state. Consider an oscillator that is interacting with a thermal environment. If the oscillator initially is in a coherent state, it will gradually decay to a thermal (incoherent) state. The amount of coherence can be quantified using the first-order optical coherence function $g^{(1)}(\tau) = \frac{\langle a^\dagger(\tau)a(0)\rangle}{\sqrt{\langle a^\dagger(\tau)a(\tau)\rangle\langle a^\dagger(0)a(0)\rangle}}$. For a coherent state $|g^{(1)}(\tau)| = 1$, and for a completely incoherent (thermal) state $g^{(1)}(\tau) = 0$. The following code calculates and plots $g^{(1)}(\tau)$ as a function of τ.

```
from qutip import *
from scipy import *

N = 15
taulist = linspace(0,10.0,200)
a = destroy(N)
H = 2 * pi * a.dag() * a

# collapse operator
G1 = 0.75
n_th = 2.00  # bath temperature in terms of excitation number
c_ops = [sqrt(G1 * (1 + n_th)) * a, sqrt(G1 * n_th) * a.dag()]

# start with a coherent state
rho0 = coherent_dm(N, 2.0)

# first calculate the occupation number as a function of time
n = mesolve(H, rho0, taulist, c_ops, [a.dag() * a]).expect[0]

# calculate the correlation function G1 and normalize with n to obtain g1
G1 = correlation(H, rho0, None, taulist, c_ops, a.dag(), a)
g1 = G1 / sqrt(n[0] * n)

from pylab import *
plot(taulist, g1, 'b')
```

```
plot(taulist, n, 'r')
title('Decay of a coherent state to an incoherent (thermal) state')
xlabel(r'$\tau$')
legend((r'First-order coherence function $g^{(1)}(\tau)$',
        r'occupation number $n(\tau)$'))
show()
```



For convenience, the steps for calculating the first-order coherence function have been collected in the function `qutip.correlation.coherence_function_g1`.

Example: second-order optical coherence function

The second-order optical coherence function, with time-delay τ, is defined as

$$g^{(2)}(\tau) = \frac{\langle a^\dagger(0)a^\dagger(\tau)a(\tau)a(0)\rangle}{\langle a^\dagger(0)a(0)\rangle^2}$$

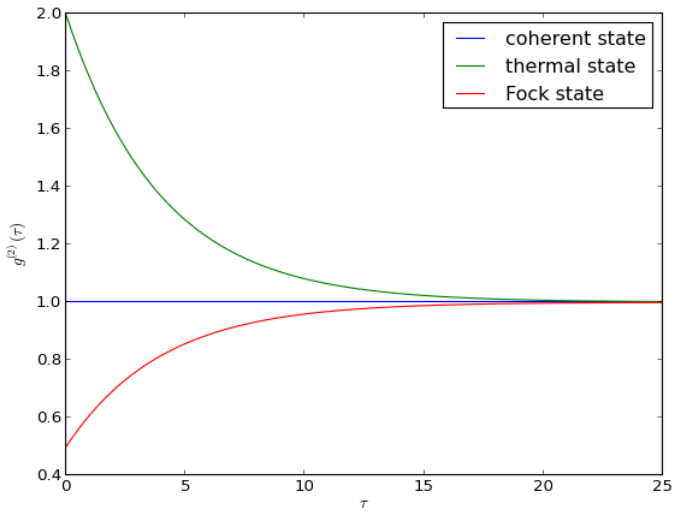
For a coherent state $g^{(2)}(\tau) = 1$, for a thermal state $g^{(2)}(\tau = 0) = 2$ and it decreases as a function of time (bunched photons, they tend to appear together), and for a Fock state with n photons $g^{(2)}(\tau = 0) = n(n-1)/n^2 < 1$ and it increases with time (anti-bunched photons, more likely to arrive separated in time).

To calculate this type of correlation function with QuTiP, we can use `qutip.correlation.correlation_4op_1t`, which computes a correlation function on the form $\langle A(0)B(\tau)C(\tau)D(0)\rangle$ (four operators, one delay-time vector).

The following code calculates and plots $g^{(2)}(\tau)$ as a function of τ for a coherent, thermal and fock state.

```
import pylab as plt
from qutip import *
from scipy import *

N = 25
taulist = linspace(0, 25.0, 200)
a = destroy(N)
H = 2 * pi * a.dag() * a

kappa = 0.25
n_th = 2.0  # bath temperature in terms of excitation number
c_ops = [sqrt(kappa * (1 + n_th)) * a, sqrt(kappa * n_th) * a.dag()]

states = [{'state': coherent_dm(N, sqrt(2.0)), 'label': "coherent state"},
          {'state': thermal_dm(N, 2.0), 'label': "thermal state"},
```

```
            {'state': fock_dm(N, 2), 'label': "Fock state"}]

fig, ax = plt.subplots(1, 1)

for state in states:
    rho0 = state['state']

    # first calculate the occupation number as a function of time
    n = mesolve(H, rho0, taulist, c_ops, [a.dag() * a]).expect[0]

    # calculate the correlation function G2 and normalize with n(0)n(t) to
    # obtain g2
    G2 = correlation_4op_1t(H, rho0, taulist, c_ops, a.dag(), a.dag(), a, a)
    g2 = G2 / (n[0] * n)

    ax.plot(taulist, real(g2), label=state['label'])

ax.legend(loc=0)
ax.set_xlabel(r'$\tau$')
ax.set_ylabel(r'$g^{(2)}(\tau)$')
plt.show()
```



For convenience, the steps for calculating the second-order coherence function have been collected in the function `qutip.correlation.coherence_function_g2`.

4.9 Plotting on the Bloch Sphere

Important: Updated in QuTiP version 2.2.

4.9.1 Introduction

When studying the dynamics of a two-level system, it is often convent to visualize the state of the system by plotting the state-vector or density matrix on the Bloch sphere. In QuTiP, we have created two different classes to allow for easy creation and manipulation of data sets, both vectors and data points, on the Bloch sphere. The `qutip.Bloch` class, uses Matplotlib to render the Bloch sphere, where as `qutip.Bloch3d` uses the Mayavi rendering engine to generate a more faithful 3D reconstruction of the Bloch sphere.

4.9.2 The Bloch and Bloch3d Classes

In QuTiP, creating a Bloch sphere is accomplished by calling either:

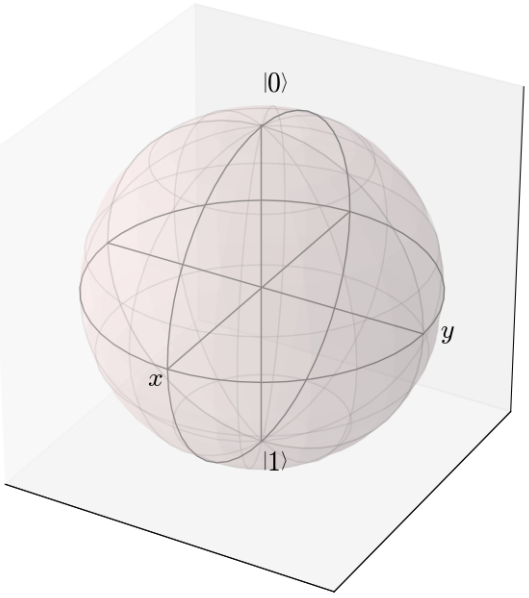
```
In [3]: b=Bloch()
```

which will load an instance of the `qutip.Bloch` class, or using:

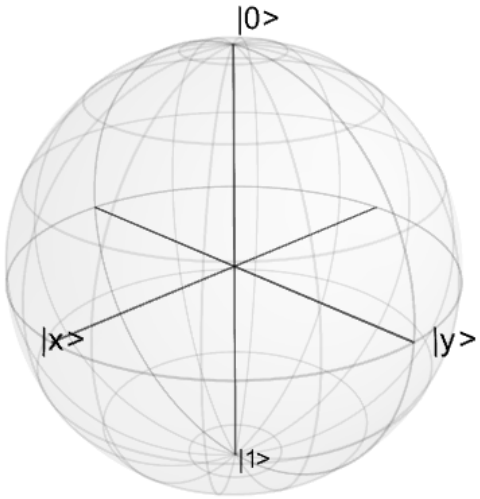
```
In [4]: b3d=Bloch3d()
```

that loads the `qutip.Bloch3d` version. Before getting into the details of these objects, we can simply plot the blank Bloch sphere associated with these instances via:

```
In [5]: b.show()
```



or

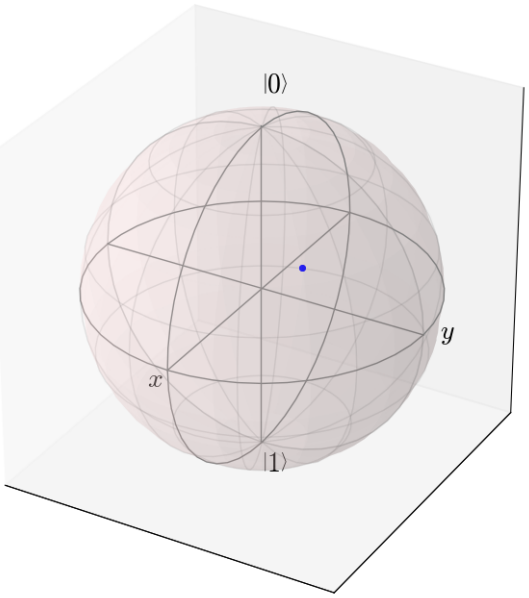


In addition to the show() command, the Bloch class has the following functions:

Name	Input Parameters (#=optional)	Description
add_points(pnts,#meth)	pnts list/array of (x,y,z) points, meth='m' (default meth='s') will plot a collection of points as multi-colored data points.	Adds a single or set of data points to be plotted on the sphere.
add_states(state,#kind)	*state* Qobj or list/array of Qobj's representing state or density matrix of a two-level system, *kind* (optional) string specifying if state should be plotted as point ('point') or vector (default).	Input multiple states as a list or array
add_vectors(vec)	*vec* list/array of (x,y,z) points giving direction and length of state vectors.	adds single or multiple vectors to plot.
clear()		Removes all data from Bloch sphere. Keeps customized figure properties.
save(#format,#dirc)	*format* format (default='png') of output file, *dirc* (default=cwd) output directory	Saves Bloch sphere to a file.
show()		Generates Bloch sphere with given data.

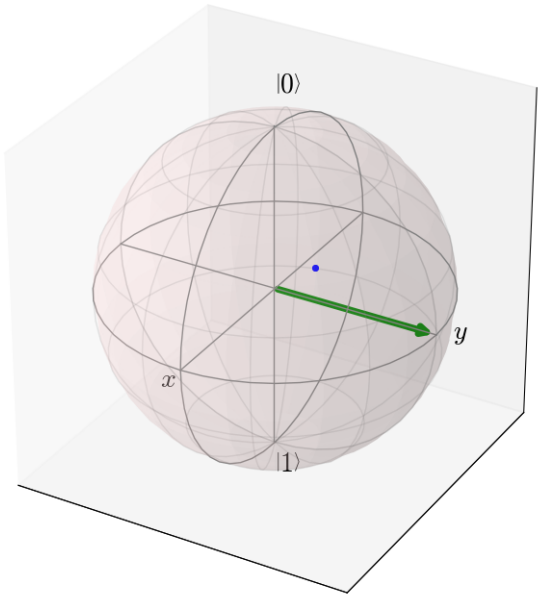
As an example, we can add a single data point:

```
In [6]: pnt=[1/sqrt(3),1/sqrt(3),1/sqrt(3)]

In [7]: b.add_points(pnt)

In [8]: b.show()
```



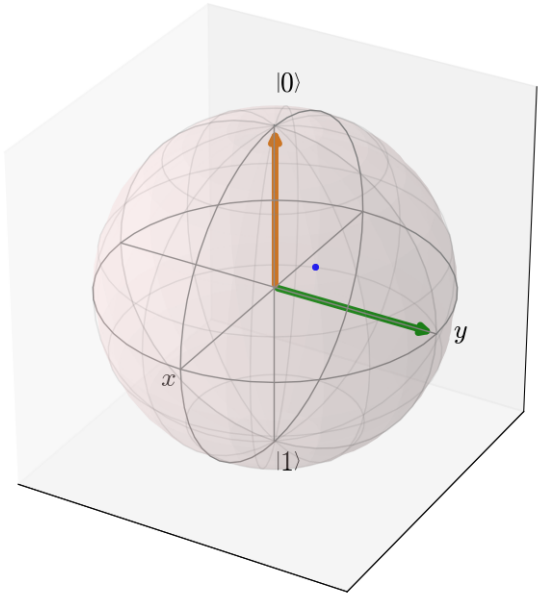
and then a single vector:

```
In [9]: vec=[0,1,0]

In [10]: b.add_vectors(vec)

In [11]: b.show()
```



and then add another vector corresponding to the $|\mathrm{up}\rangle$ state:

```
In [12]: up=basis(2,0)

In [13]: b.add_states(up)

In [14]: b.show()
```

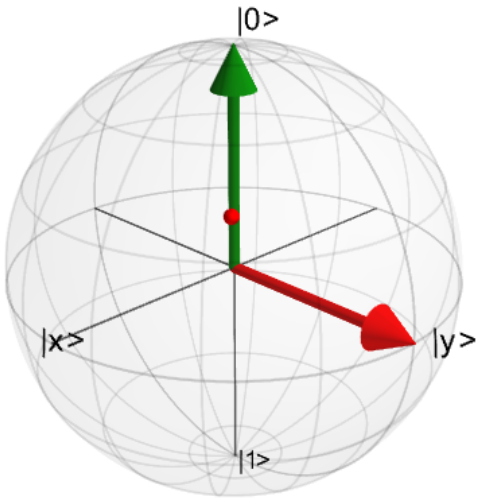


Notice that when we add more than a single vector (or data point), a different color will automatically be applied to the later data set (mod 4). In total, the code for constructing our Bloch sphere with one vector, one state, and a single data point is:

```
>>> b=Bloch()
>>> pnt=[1/sqrt(3),1/sqrt(3),1/sqrt(3)]
>>> b.add_points(pnt)
>>> #b.show()
```

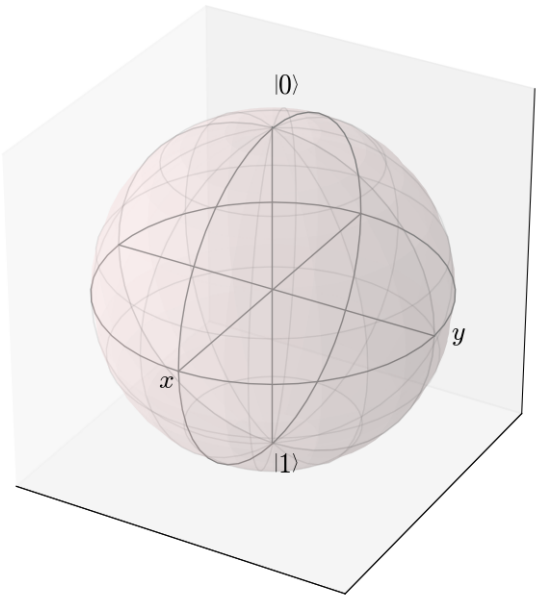
```
>>> vec=[0,1,0]
>>> b.add_vectors(vec)
>>> #b.show()
>>> up=basis(2,0)
>>> b.add_states(up)
>>> b.show()
```

where we have commented out the extra *show()* commands. Replacing `b=Bloch()` with `b=Bloch3d()` in the above code generates the following 3D Bloch sphere.



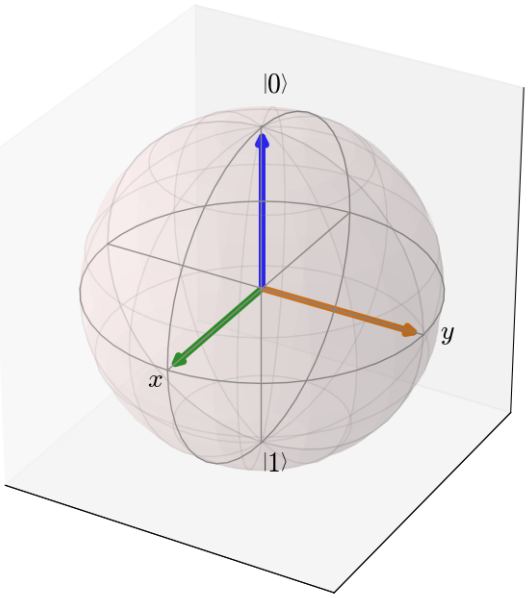
We can also plot multiple points, vectors, and states at the same time by passing list or arrays instead of individual elements. Before giving an example, we can use the *clear()* command to remove the current data from our Bloch sphere instead of creating a new instance:

```
In [15]: b.clear()

In [16]: b.show()
```



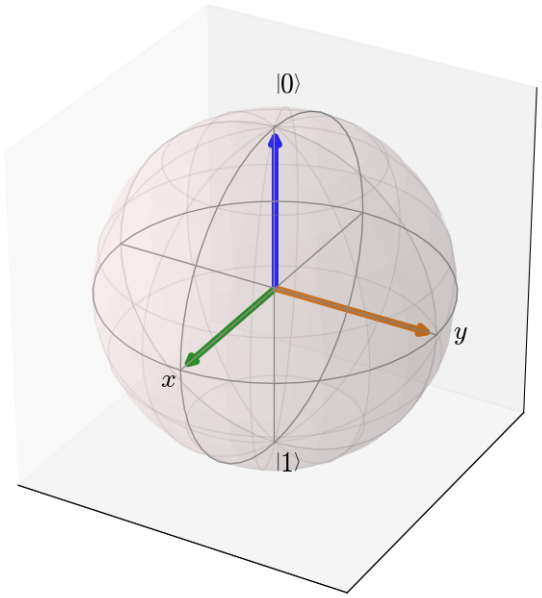
Now on the same Bloch sphere, we can plot the three states associated with the x, y, and z directions:

```
In [17]: x=(basis(2,0)+(1+0j)*basis(2,1)).unit()

In [18]: y=(basis(2,0)+(0+1j)*basis(2,1)).unit()

In [19]: z=(basis(2,0)+(0+0j)*basis(2,1)).unit()

In [20]: b.add_states([x,y,z])

In [21]: b.show()
```



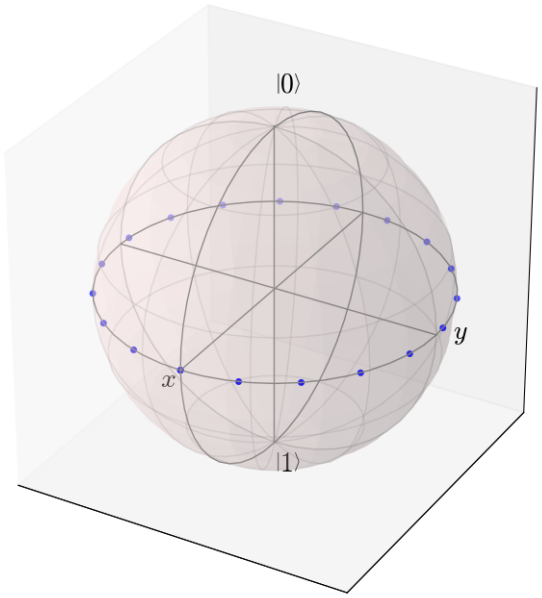
a similar method works for adding vectors:

```
In [22]: b.clear()

In [23]: vec=[[1,0,0],[0,1,0],[0,0,1]]

In [24]: b.add_vectors(vec)

In [25]: b.show()
```



Adding multiple points to the Bloch sphere works slightly differently than adding multiple states or vectors. For example, lets add a set of 20 points around the equator (after calling *clear()*):

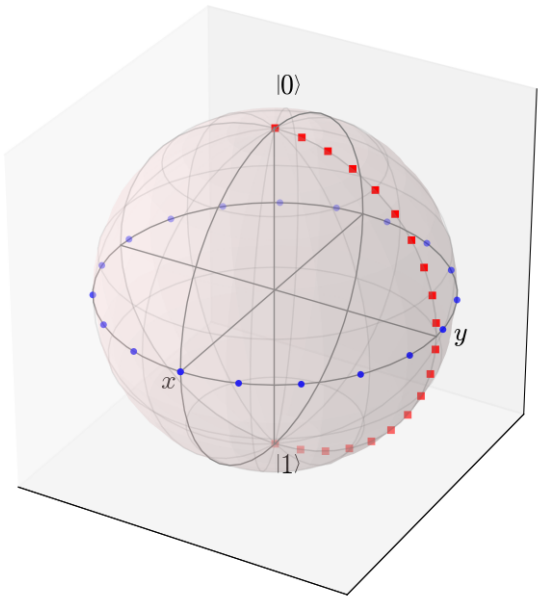
```
In [27]: xp=[cos(th) for th in linspace(0,2*pi,20)]

In [28]: yp=[sin(th) for th in linspace(0,2*pi,20)]

In [29]: zp=zeros(20)

In [30]: pnts=[xp,yp,zp]

In [31]: b.add_points(pnts)

In [32]: b.show()
```



Notice that, in contrast to states or vectors, each point remains the same color as the initial point. This is because

adding multiple data points using the add_points function is interpreted, by default, to correspond to a single data point (single qubit state) plotted at different times. This is very useful when visualizing the dynamics of a qubit. An example of this is given in the example . If we want to plot additional qubit states we can call additional *add_points* functions:

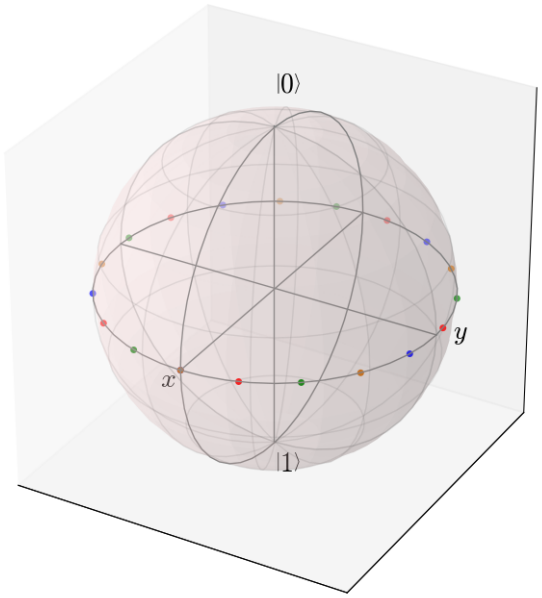
```
In [33]: xz=zeros(20)

In [34]: yz=[sin(th) for th in linspace(0,pi,20)]

In [35]: zz=[cos(th) for th in linspace(0,pi,20)]

In [36]: b.add_points([xz,yz,zz])

In [37]: b.show()
```



The color and shape of the data points is varied automatically by the Bloch class. Notice how the color and point markers change for each set of data. Again, we have had to call `add_points` twice because adding more than one set of multiple data points is *not* supported by the `add_points` function.

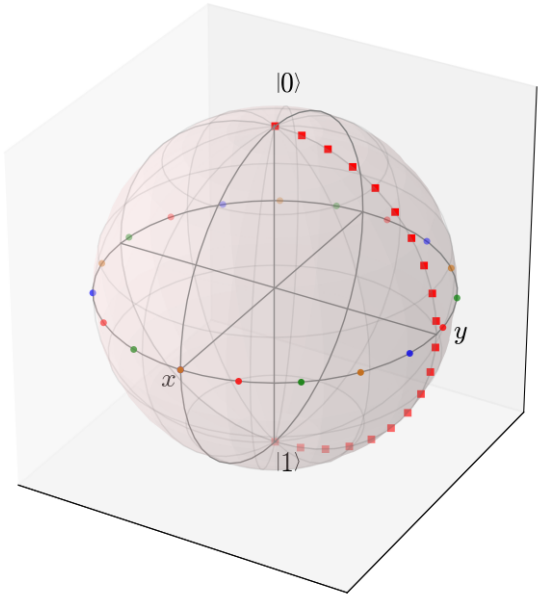
What if we want to vary the color of our points. We can tell the `qutip.Bloch` class to vary the color of each point according to the colors listed in the `b.point_color` list (see *Configuring the Bloch sphere* below). Again after `clear()`:

```
In [39]: xp=[cos(th) for th in linspace(0,2*pi,20)]

In [40]: yp=[sin(th) for th in linspace(0,2*pi,20)]

In [41]: zp=zeros(20)

In [42]: pnts=[xp,yp,zp]

In [43]: b.add_points(pnts,'m') # <-- add a 'm' string to signify 'multi' colored points

In [44]: b.show()
```



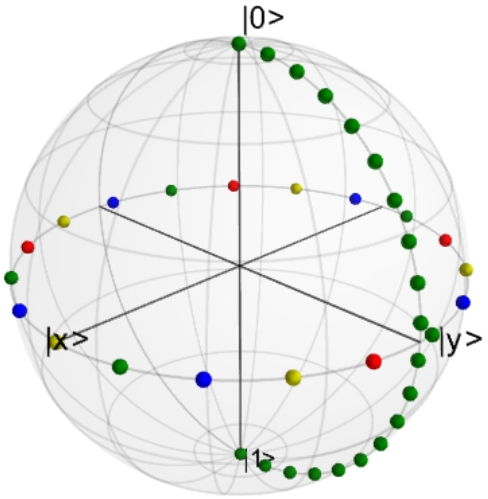
Now, the data points cycle through a variety of predefined colors. Now lets add another set of points, but this time we want the set to be a single color, representing say a qubit going from the $|\mathrm{up}\rangle$ state to the $|\mathrm{down}\rangle$ state in the y-z plane:

```
In [45]: xz=zeros(20)

In [46]: yz=[sin(th) for th in linspace(0,pi,20)]

In [47]: zz=[cos(th) for th in linspace(0,pi,20)]

In [48]: b.add_points([xz,yz,zz]) # no 'm'

In [49]: b.show()
```



Again, the same plot can be generated using the `qutip.Bloch3d` class by replacing `Bloch` with `Bloch3d`:

A more slick way of using this 'multi' color feature is also given in the example, where we set the color of the



markers as a function of time.

Differences Between Bloch and Bloch3d

While in general the `Bloch` and `Bloch3d` classes are interchangeable, there are some important differences to consider when choosing between them.

- The `Bloch` class uses Matplotlib to generate figures. As such, the data plotted on the sphere is in reality just a 2D object. In contrast the `Bloch3d` class uses the 3D rendering engine from VTK via mayavi to generate the sphere and the included data. In this sense the `Bloch3d` class is much more advanced, as objects are rendered in 3D leading to a higher quality figure.
- Only the `Bloch` class can be embedded in a Matplotlib figure window. Thus if you want to combine a Bloch sphere with another figure generated in QuTiP, you can not use `Bloch3d`. Of course you can always post-process your figures using other software to get the desired result.
- Due to limitations in the rendering engine, the `Bloch3d` class does not support LaTex for text. Again, you can get around this by post-processing.
- The user customizable attributes for the `Bloch` and `Bloch3d` classes are not identical. Therefore, if you change the properties of one of the classes, these changes will cause an exception if the class is switched.

4.9.3 Configuring the Bloch sphere

Bloch Class Options

At the end of the last section we saw that the colors and marker shapes of the data plotted on the Bloch sphere are automatically varied according to the number of points and vectors added. But what if you want a different choice of color, or you want your sphere to be purple with different axes labels? Well then you are in luck as the Bloch class has 22 attributes which one can control. Assuming `b=Bloch()`:

Attribute	Function	Default Setting
b.axes	Matplotlib axes instance for animations. Set by `axes` keyword arg.	`None`
b.fig	User supplied Matplotlib Figure instance. Set by `fig` keyword arg.	`None`
b.font_color	Color of fonts	*'black'*
b.font_size	Size of fonts	20
b.frame_alpha	Transparency of wireframe	0.1
b.frame_color	Color of wireframe	*'gray'*
b.frame_width	Width of wireframe	1
b.point_color	List of colors for Bloch point markers to cycle through	*['b','r','g','#CC6600']*
b.point_marker	List of point marker shapes to cycle through	*['o','s','d','^']*
b.point_size	List of point marker sizes (not all markers look the same size when plotted)	*[55,62,65,75]*
b.sphere_alpha	Transparency of Bloch sphere	0.2
b.sphere_color	Color of Bloch sphere	*'#FFDDDD'*
b.size	Sets size of figure window	*[7,7]* (700x700 pixels)
b.vector_color	List of colors for Bloch vectors to cycle through	*['g','#CC6600','b','r']*
b.vector_width	Width of Bloch vectors	4
b.view	Azimuthal and Elevation viewing angles	*[-60,30]*
b.xlabel	Labels for x-axis	*['x',''']* +x and -x (labels use LaTeX)
b.xlpos	Position of x-axis labels	*[1.1,-1.1]*
b.ylabel	Labels for y-axis	*['y','']* +y and -y (labels use LaTeX)
b.ylpos	Position of y-axis labels	*[1.2,-1.2]*
b.zlabel	Labels for z-axis	*['$\left\|0\right>$','$\left\|1\right>$']* +z and -z (labels use LaTeX)
b.zlpos	Position of z-axis labels	*[1.2,-1.2]*

Bloch3d Class Options

The Bloch3d sphere is also customizable. Note however that the attributes for the `Bloch3d` class are not in one-to-one correspondence to those of the `Bloch` class due to the different underlying rendering engines. Assuming `b=Bloch3d()`:

Attribute	Function	Default Setting
b.fig	User supplied Mayavi Figure instance. Set by `fig` keyword arg.	`None`
b.font_color	Color of fonts	*'black'*
b.font_scale	Scale of fonts	0.08
b.frame	Draw wireframe for sphere?	True
b.frame_alpha	Transparency of wireframe	0.05
b.frame_color	Color of wireframe	*'gray'*
b.frame_num	Number of wireframe elements to draw	8
b.frame_radius	Radius of wireframe lines	0.005
b.point_color	List of colors for Bloch point markers to cycle through	*['r', 'g', 'b', 'y']*
b.point_mode	Type of point markers to draw	*sphere*
b.point_size	Size of points	0.075
b.sphere_alpha	Transparency of Bloch sphere	0.1
b.sphere_color	Color of Bloch sphere	*'#808080'*
b.size	Sets size of figure window	*[500,500]* (500x500 pixels)
b.vector_color	List of colors for Bloch vectors to cycle through	*['r', 'g', 'b', 'y']*
b.vector_width	Width of Bloch vectors	3
b.view	Azimuthal and Elevation viewing angles	*[45,65]*
b.xlabel	Labels for x-axis	['|x>', ''] +x and -x
b.xlpos	Position of x-axis labels	*[1.07,-1.07]*
b.ylabel	Labels for y-axis	*['y','']* +y and -y
b.ylpos	Position of y-axis labels	*[1.07,-1.07]*
b.zlabel	Labels for z-axis	*['|0>', '|1>']* +z and -z
b.zlpos	Position of z-axis labels	*[1.07,-1.07]*

These properties can also be accessed via the print command:

```
In [50]: b=Bloch()

In [51]: print(b)
Bloch data:
-----------
Number of points:  0
Number of vectors: 0

Bloch sphere properties:
------------------------
font_color:      black
font_size:       20
frame_alpha:     0.2
frame_color:     gray
frame_width:     1
point_color:     ['b', 'r', 'g', '#CC6600']
point_marker:    ['o', 's', 'd', '^']
point_size:      [25, 32, 35, 45]
sphere_alpha:    0.2
sphere_color:    #FFDDDD
size:            [7, 7]
vector_color:    ['g', '#CC6600', 'b', 'r']
vector_width:    5
vector_style:    -|>
vector_mutation: 20
view:            [-60, 30]
xlabel:          ['$x$', '']
xlpos:           [1.2, -1.2]
ylabel:          ['$y$', '']
ylpos:           [1.1, -1.1]
zlabel:          ['$\\left|0\\right>$', '$\\left|1\\right>$']
zlpos:           [1.2, -1.2]
```

4.9.4 Animating with the Bloch sphere

The Bloch class was designed from the outset to generate animations. To animate a set of vectors or data points the basic idea is: plot the data at time t1, save the sphere, clear the sphere, plot data at t2,... The Bloch sphere will automatically number the output file based on how many times the object has been saved (this is stored in b.savenum). The easiest way to animate data on the Bloch sphere is to use the `save()` method and generate a series of images to convert into an animation. However, as of Matplotlib version 1.1, creating animations is built-in. We will demonstrate both methods by looking at the decay of a qubit on the bloch sphere.

Example: Qubit Decay

The code for calculating the expectation values for the Pauli spin operators of a qubit decay is given below. This code is common to both animation examples.

```
from qutip import *
from scipy import *
def qubit_integrate(w, theta, gamma1, gamma2, psi0, tlist):
    # operators and the hamiltonian
    sx = sigmax(); sy = sigmay(); sz = sigmaz(); sm = sigmam()
    H = w * (cos(theta) * sz + sin(theta) * sx)
    # collapse operators
    c_op_list = []
    n_th = 0.5 # temperature
    rate = gamma1 * (n_th + 1)
    if rate > 0.0: c_op_list.append(sqrt(rate) * sm)
    rate = gamma1 * n_th
    if rate > 0.0: c_op_list.append(sqrt(rate) * sm.dag())
    rate = gamma2
    if rate > 0.0: c_op_list.append(sqrt(rate) * sz)


    # evolve and calculate expectation values
    output = mesolve(H, psi0, tlist, c_op_list, [sx, sy, sz])
    return output.expect[0], output.expect[1], output.expect[2]

## calculate the dynamics
w     = 1.0 * 2 * pi    # qubit angular frequency
theta = 0.2 * pi        # qubit angle from sigma_z axis (toward sigma_x axis)
gamma1 = 0.5       # qubit relaxation rate
gamma2 = 0.2       # qubit dephasing rate
# initial state
a = 1.0
psi0 = (a* basis(2,0) + (1-a)*basis(2,1))/(sqrt(a**2 + (1-a)**2))
tlist = linspace(0,4,250)
#expectation values for ploting
sx, sy, sz = qubit_integrate(w, theta, gamma1, gamma2, psi0, tlist)
```

Generating Images for Animation

An example of generating images for generating an animation outside of Python is given below:

```
b=Bloch()
b.vector_color = ['r']
b.view=[-40,30]
for i in xrange(len(sx)):
    b.clear()
    b.add_vectors([sin(theta),0,cos(theta)])
    b.add_points([sx[:i+1],sy[:i+1],sz[:i+1]])
    b.save(dirc='temp') #saving images to temp directory in current working directory
```

Generating an animation using ffmpeg (for example) is fairly simple:

```
ffmpeg -r 20 -b 1800 -i bloch_%01d.png bloch.mp4
```

Directly Generating an Animation

Important: Generating animations directly from Matplotlib requires installing either mencoder or ffmpeg. While either choice works on linux, it is best to choose ffmpeg when running on the Mac. If using macports just do: `sudo port install ffmpeg`.

The code to directly generate an mp4 movie of the Qubit decay is as follows:

```
from pylab import *
import matplotlib.animation as animation
from mpl_toolkits.mplot3d import Axes3D

fig = figure()
ax = Axes3D(fig,azim=-40,elev=30)
sphere=Bloch(axes=ax)

def animate(i):
    sphere.clear()
    sphere.add_vectors([sin(theta),0,cos(theta)])
    sphere.add_points([sx[:i+1],sy[:i+1],sz[:i+1]])
    sphere.make_sphere()
    return ax

def init():
    sphere.vector_color = ['r']
    return ax

ani = animation.FuncAnimation(fig, animate, np.arange(len(sx)),
                              init_func=init, blit=True, repeat=False)
ani.save('bloch_sphere.mp4', fps=20, clear_temp=True)
```

The resulting movie may be viewed here: Bloch_Decay.mp4

4.10 Visualization of quantum states and processes

Visualization is often an important complement to a simulation of a quantum mechanical system. The first method of visualization that comes to mind might be to plot the expectation values of a few selected operators. But on top of that, it can often be instructive to visualize for example the state vectors or density matices that describe the state of the system, or how the state is transformed as a function of time (see process tomography below). In this section we demonstrate how QuTiP and matplotlib can be used to perform a few types of visualizations that often can provide additional understanding of quantum system.

4.10.1 Fock-basis probability distribution

In quantum mechanics probability distributions plays an important role, and as in statistics, the expectation values computed from a probability distribution does not reveal the full story. For example, consider an quantum harmonic oscillator mode with Hamiltonian $H = \hbar\omega a^\dagger a$, which is in a state described by its density matrix ρ, and which on average is occupied by two photons, $\mathrm{Tr}[\rho a^\dagger a] = 2$. Given this information we cannot say whether the oscillator is in a Fock state, a thermal state, a coherent state, etc. By visualizing the photon distribution in the Fock state basis important clues about the underlying state can be obtained.

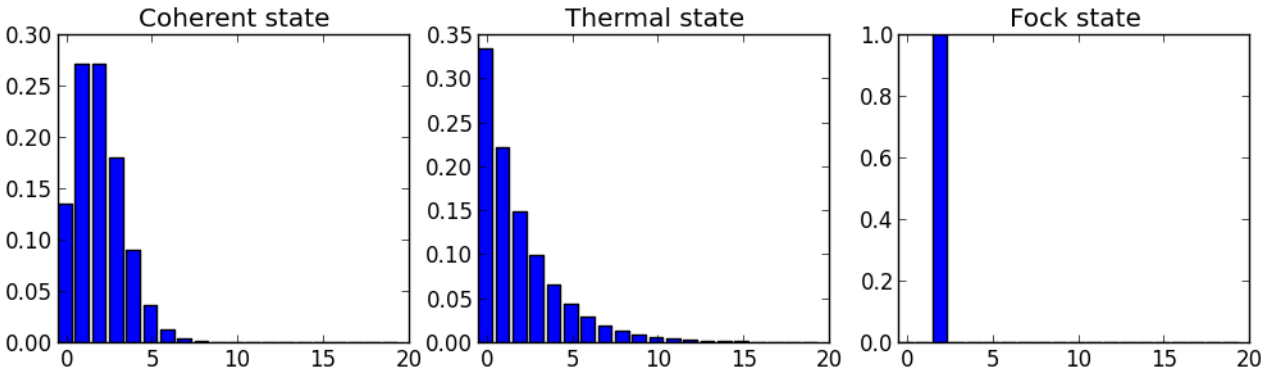
One convenient way to visualize a probability distribution is to use histograms. Consider the following histogram visualization of the number-basis probability distribution, which can be obtained from the diagonal of the density matrix, for a few possible oscillator states with on average occupation of two photons.

First we generate the density matrices for the coherent, thermal and fock states.

```
In [1]: N = 20

In [2]: rho_coherent = coherent_dm(N, sqrt(2))

In [3]: rho_thermal = thermal_dm(N, 2)

In [4]: rho_fock = fock_dm(N, 2)
```

Next, we plot histograms of the diagonals of the density matrices:

```
In [1]: fig, axes = subplots(1, 3, figsize=(12,3))

In [2]: bar0 = axes[0].bar(arange(0, N)-.5, rho_coherent.diag())

In [3]: lbl0 = axes[0].set_title("Coherent state")

In [4]: lim0 = axes[0].set_xlim([-.5, N])

In [5]: bar1 = axes[1].bar(arange(0, N)-.5, rho_thermal.diag())

In [6]: lbl1 = axes[1].set_title("Thermal state")

In [7]: lim1 = axes[1].set_xlim([-.5, N])

In [8]: bar2 = axes[2].bar(arange(0, N)-.5, rho_fock.diag())

In [9]: lbl2 = axes[2].set_title("Fock state")

In [10]: lim2 = axes[2].set_xlim([-.5, N])

In [11]: show()
```

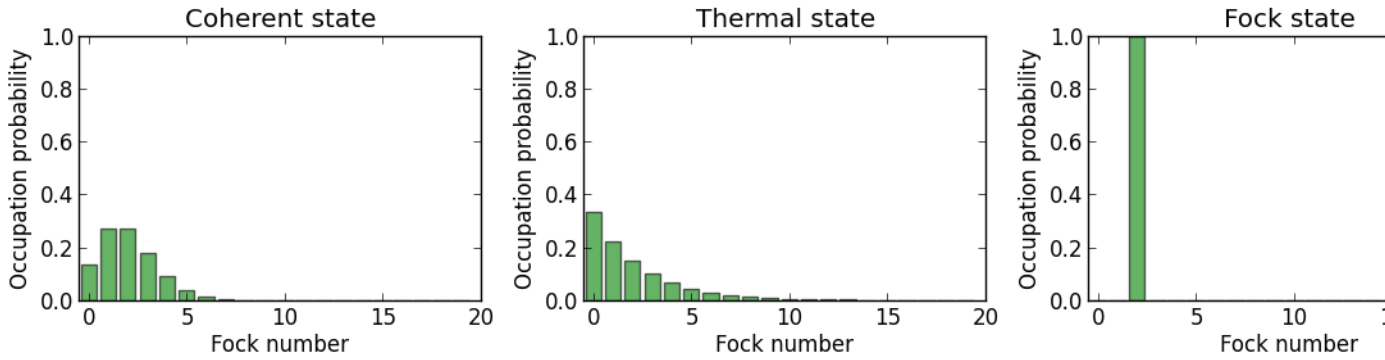


All these states correspond to an average of two photons, but by visualizing the photon distribution in Fock basis the differences between these states are easily appreciated.

One frequently need to visualize the Fock-distribution in the way described above, so QuTiP provides a convenience function for doing this, see `qutip.visualization.fock_distribution`, and the following example:

```
In [1]: fig, axes = subplots(1, 3, figsize=(12,3))

In [2]: fock_distribution(rho_coherent, fig=fig, ax=axes[0], title="Coherent state");
```

```
In [3]: fock_distribution(rho_thermal, fig=fig, ax=axes[1], title="Thermal state");

In [4]: fock_distribution(rho_fock, fig=fig, ax=axes[2], title="Fock state");

In [5]: fig.tight_layout()

In [6]: show()
```



4.10.2 Quasi-probability distributions

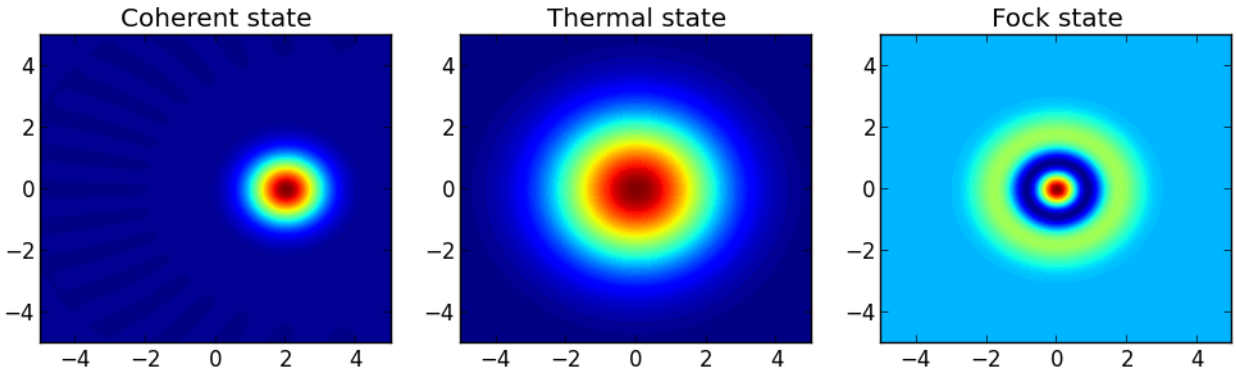
The probability distribution in the number (Fock) basis only describes the occupation probabilities for a discrete set of states. A more complete phase-space probability-distribution-like function for harmonic modes are the Wigner and Husumi Q-functions, which are full descriptions of the quantum state (equivalent to the density matrix). These are called quasi-distribution functions because unlike real probability distribution functions they can for example be negative. In addition to being more complete descriptions of a state (compared to only the occupation probabilities plotted above), these distributions are also great for demonstrating if a quantum state is quantum mechanical, since for example a negative Wigner function is a definite indicator that a state is distinctly nonclassical.

Wigner function

In QuTiP, the Wigner function for a harmonic mode can be calculated with the function `qutip.wigner.wigner`. It takes a ket or a density matrix as input, together with arrays that define the ranges of the phase-space coordinates (in the x-y plane). In the following example the Wigner functions are calculated and plotted for the same three states as in the previous section.

```
In [1]: xvec = linspace(-5,5,200)

In [2]: W_coherent = wigner(rho_coherent, xvec, xvec)

In [3]: W_thermal = wigner(rho_thermal, xvec, xvec)

In [4]: W_fock = wigner(rho_fock, xvec, xvec)

In [5]: # plot the results

In [6]: fig, axes = subplots(1, 3, figsize=(12,3))

In [7]: cont0 = axes[0].contourf(xvec, xvec, W_coherent, 100)

In [8]: lbl0 = axes[0].set_title("Coherent state")

In [9]: cont1 = axes[1].contourf(xvec, xvec, W_thermal, 100)
```

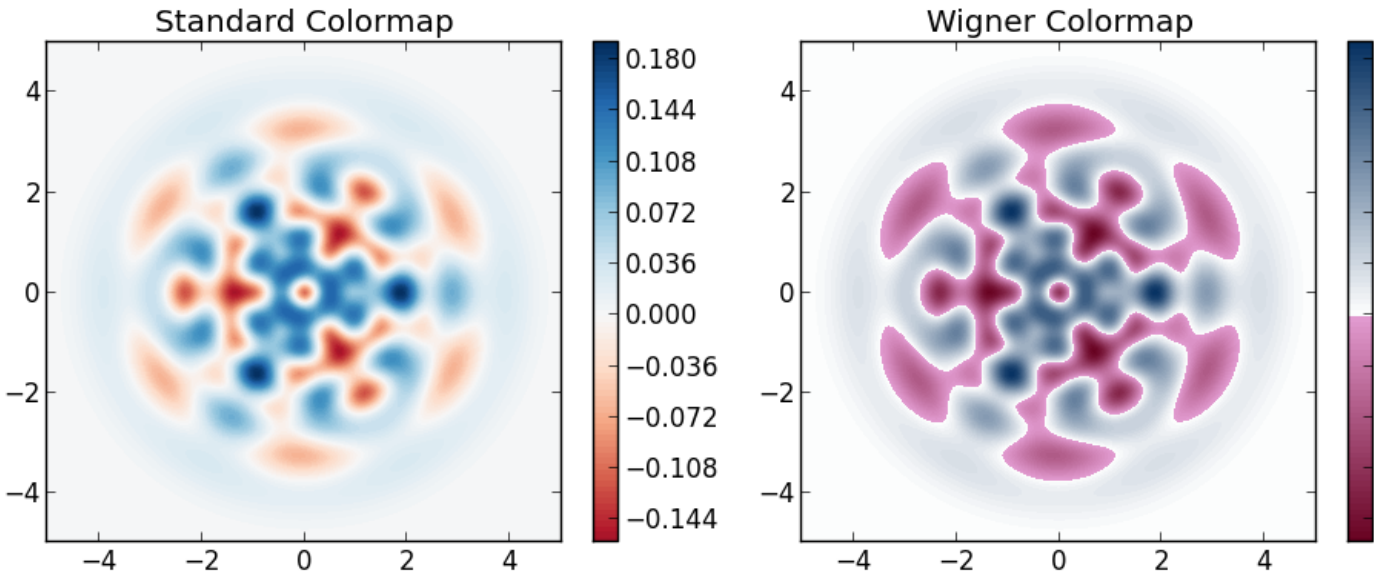
```
In [10]: lbl1 = axes[1].set_title("Thermal state")

In [11]: cont0 = axes[2].contourf(xvec, xvec, W_fock, 100)

In [12]: lbl2 = axes[2].set_title("Fock state")

In [13]: show()
```



Custom Color Maps

The main objective when plotting a Wigner function is to demonstrate that the underlying state is nonclassical, as indicated by negative values in the Wigner function. Therefore, making these negative values stand out in a figure is helpful for both analysis and publication purposes. Unfortunately, all of the color schemes used in Matplotlib (or any other plotting software) are linear colormaps where small negative values tend to be near the same color as the zero values, and are thus hidden. To fix this dilemma, QuTiP includes a nonlinear colormap function `qutip.visualization.wigner_cmap` that colors all negative values differently than positive or zero values. Below is a demonstration of how to use this function in your Wigner figures:

```
In [1]: psi = (basis(10, 0) + basis(10, 3) + basis(10, 9)).unit()

In [2]: xvec = linspace(-5, 5, 500)

In [3]: W = wigner(psi, xvec, xvec)

In [4]: wmap = wigner_cmap(W)  # Generate Wigner colormap

In [5]: nrm = mpl.colors.Normalize(-W.max(), W.max())

In [6]: fig, axes = subplots(1, 2, figsize=(10, 4))

In [7]: plt1 = axes[0].contourf(xvec, xvec, W, 100, cmap=cm.RdBu, norm=nrm)

In [8]: axes[0].set_title("Standard Colormap");

In [9]: cb1 = colorbar(plt1, ax=axes[0])

In [10]: plt2 = axes[1].contourf(xvec, xvec, W, 100, cmap=wmap)  # Apply Wigner colormap

In [11]: axes[1].set_title("Wigner Colormap");

In [12]: cb2 = fig.colorbar(plt2, ax=axes[1])

In [13]: fig.tight_layout()

In [14]: show()
```



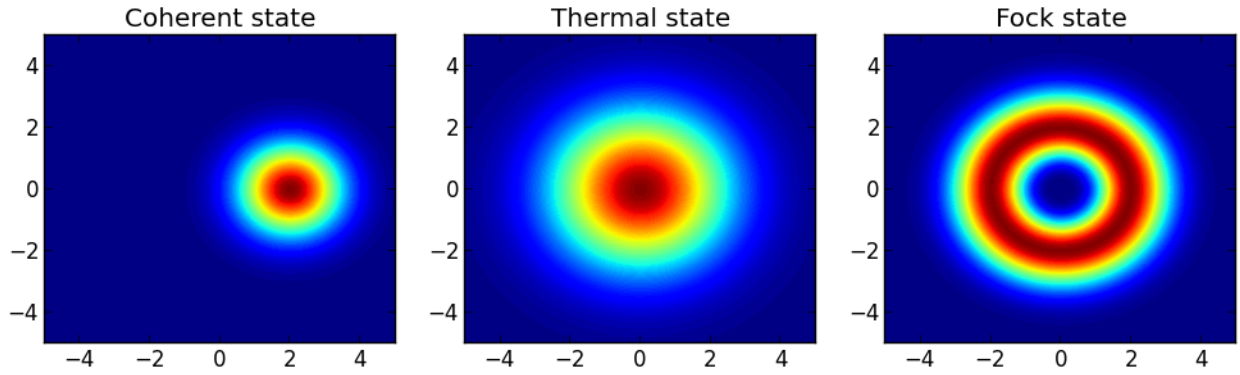
Husimi Q-function

The Husimi Q function is, like the Wigner function, a quasiprobability distribution for harmonic modes. It is defined as

$$Q(\alpha) = \frac{1}{\pi} \langle\alpha|\rho|\alpha\rangle$$

where $|\alpha\rangle$ is a coherent state and $\alpha = x + iy$. In QuTiP, the Husimi Q function can be computed given a state ket or density matrix using the function `qutip.wigner.qfunc`, as demonstrated below.

```
In [1]: Q_coherent = qfunc(rho_coherent, xvec, xvec)

In [2]: Q_thermal = qfunc(rho_thermal, xvec, xvec)

In [3]: Q_fock = qfunc(rho_fock, xvec, xvec)

In [4]: fig, axes = subplots(1, 3, figsize=(12,3))

In [5]: cont0 = axes[0].contourf(xvec, xvec, Q_coherent, 100)

In [6]: lbl0 = axes[0].set_title("Coherent state")

In [7]: cont1 = axes[1].contourf(xvec, xvec, Q_thermal, 100)

In [8]: lbl1 = axes[1].set_title("Thermal state")

In [9]: cont0 = axes[2].contourf(xvec, xvec, Q_fock, 100)

In [10]: lbl2 = axes[2].set_title("Fock state")

In [11]: show()
```



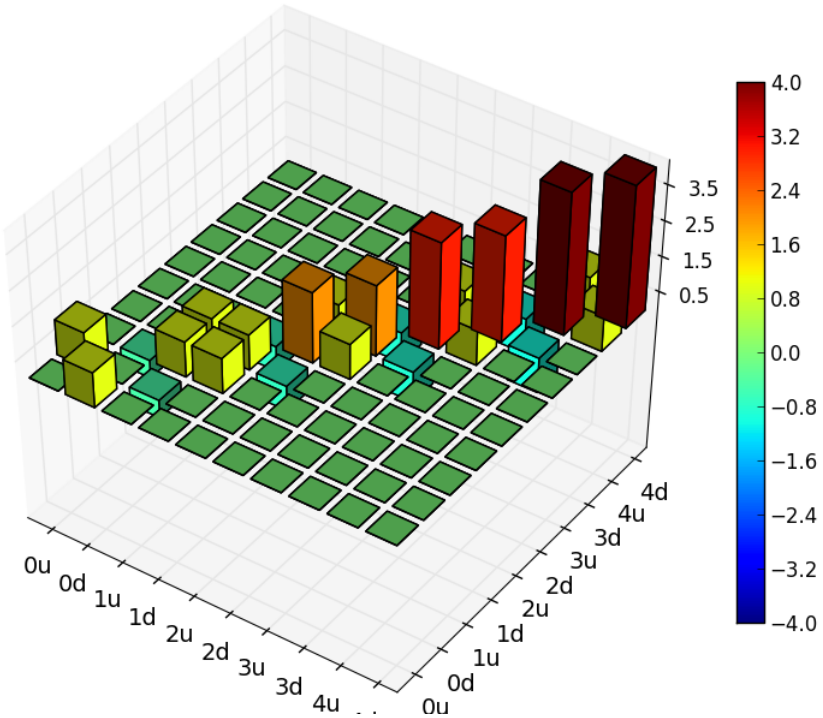
4.10.3 Visualizing operators

Sometimes, it may also be useful to directly visualizing the underlying matrix representation of an operator. The density matrix, for example, is an operator whose elements can give insights about the state it represents, but one might also be interesting in plotting the matrix of an Hamiltonian to inspect the structure and relative importance of various elements.

QuTiP offers a few functions for quickly visualizing matrix data in the form of histograms, `qutip.visualization.matrix_histogram` and `qutip.visualization.matrix_histogram_complex`, and as Hinton diagram of weighted squares, `qutip.visualization.hinton`. These functions takes a `qutip.Qobj.Qobj` as first argument, and optional arguments to, for example, set the axis labels and figure title (see the function's documentation for details).

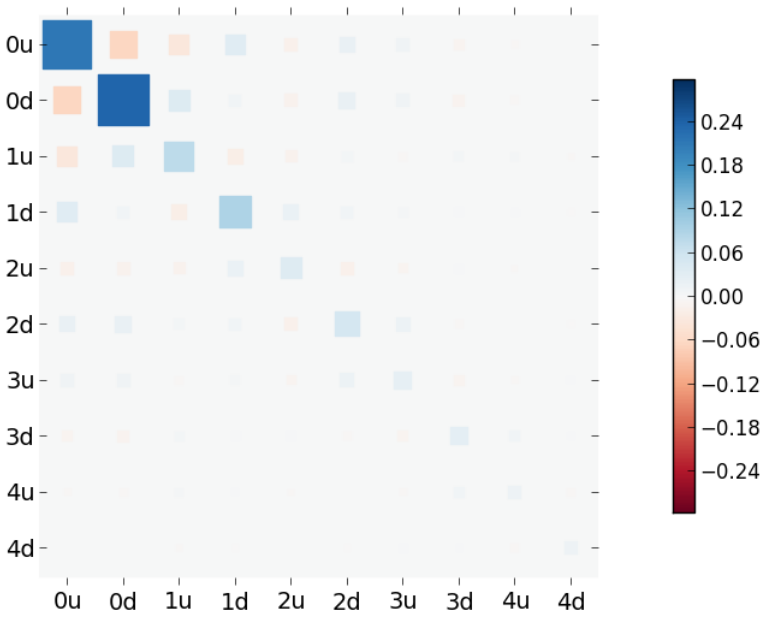
For example, to illustrate the use of `qutip.visualization.matrix_histogram`, let's visualize of the Jaynes-Cummings Hamiltonian:

```
In [1]: N = 5

In [2]: a = tensor(destroy(N), qeye(2))

In [3]: b = tensor(qeye(N), destroy(2))

In [4]: sx = tensor(qeye(N), sigmax())

In [5]: H = a.dag() * a + sx - 0.5 * (a * b.dag() + a.dag() * b)

In [6]: # visualize H

In [7]: lbls_list = [[str(d) for d in range(N)], ["u", "d"]]

In [8]: xlabels = []

In [9]: for inds in tomography._index_permutations([len(lbls) for lbls in lbls_list]):
   ...:     xlabels.append("".join([lbls_list[k][inds[k]] for k in range(len(lbls_list))]))
   ...:

In [10]: ax = matrix_histogram(H, xlabels, xlabels, limits=[-4,4])

In [11]: ax.view_init(azim=-55, elev=45)

In [12]: show()
```



Similarly, we can use the function `qutip.visualization.hinton`, which is used below to visualize the corresponding steadystate density matrix:

```
In [1]: rho_ss = steadystate(H, [sqrt(0.1) * a, sqrt(0.4) * b.dag()])
warning: (almost) singular matrix! (estimated cond. number: 8.33e+14)

In [2]: ax = hinton(rho_ss, xlabels=xlabels, ylabels=xlabels)

In [3]: show()
```



4.10.4 Quantum process tomography

Quantum process tomography (QPT) is a useful technique for characterizing experimental implementations of quantum gates involving a small number of qubits. It can also be a useful theoretical tool that can give insight in how a process transforms states, and it can be used for example to study how noise or other imperfections deteriorate a gate. Whereas a fidelity or distance measure can give a single number that indicates how far from ideal a gate is, a quantum process tomography analysis can give detailed information about exactly what kind of errors various imperfections introduce.

The idea is to construct a transformation matrix for a quantum process (for example a quantum gate) that describes how the density matrix of a system is transformed by the process. We can then decompose the transformation in some operator basis that represent well-defined and easily interpreted transformations of the input states.

To see how this works [see e.g. Mohseni et al., PRA 77, 032322 (2008) for more details], consider a process that is described by quantum map $\epsilon(\rho_{\rm in}) = \rho_{\rm out}$, which can be written

$$\epsilon(\rho_{\rm in}) = \rho_{\rm out} = \sum_{i}^{N^2} A_i \rho_{\rm in} A_i^\dagger, \tag{4.20}$$

where N is the number of states of the system (that is, ρ is represented by an $[N \times N]$ matrix). Given an orthogonal operator basis of our choice $\{B_i\}_i^{N^2}$, which satisfies $\mathrm{Tr}[B_i^\dagger B_j] = N\delta_{ij}$, we can write the map as

$$\epsilon(\rho_{\rm in}) = \rho_{\rm out} = \sum_{mn} \chi_{mn} B_m \rho_{\rm in} B_n^\dagger. \tag{4.21}$$

where $\chi_{mn} = \sum_{ij} b_{im} b_{jn}^*$ and $A_i = \sum_m b_{im} B_m$. Here, matrix χ is the transformation matrix we are after, since it describes how much $B_m \rho_{\rm in} B_n^\dagger$ contributes to $\rho_{\rm out}$.

In a numerical simulation of a quantum process we usually do not have access to the quantum map in the form Eq. (4.20). Instead, what we usually can do is to calculate the propagator U for the density matrix in superoperator form, using for example the QuTiP function `qutip.propagator.propagator`. We can then write

$$\epsilon(\tilde{\rho}_{\rm in}) = U\tilde{\rho}_{\rm in} = \tilde{\rho}_{\rm out}$$

where $\tilde{\rho}$ is the vector representation of the density matrix ρ. If we write Eq. (4.21) in superoperator form as well we obtain

$$\tilde{\rho}_{\rm out} = \sum_{mn} \chi_{mn} \tilde{B}_m \tilde{B}_n^\dagger \tilde{\rho}_{\rm in} = U\tilde{\rho}_{\rm in}.$$

so we can identify

$$U = \sum_{mn} \chi_{mn} \tilde{B}_m \tilde{B}_n^\dagger.$$

Now this is a linear equation systems for the $N^2 \times N^2$ elements in χ. We can solve it by writing χ and the superoperator propagator as $[N^4]$ vectors, and likewise write the superoperator product $\tilde{B}_m \tilde{B}_n^\dagger$ as a $[N^4 \times N^4]$ matrix M:

$$U_I = \sum_{J}^{N^4} M_{IJ} \chi_J$$

with the solution

$$\chi = M^{-1} U.$$

Note that to obtain χ with this method we have to construct a matrix M with a size that is the square of the size of the superoperator for the system. Obviously, this scales very badly with increasing system size, but this method can still be a very useful for small systems (such as system comprised of a small number of coupled qubits).

Implementation in QuTiP

In QuTiP, the procedure described above is implemented in the function `qutip.tomography.qpt`, which returns the χ matrix given a density matrix propagator. To illustrate how to use this function, let's consider the i-SWAP gate for two qubits. In QuTiP the function `qutip.gates.iswap` generates the unitary transformation for the state kets:

```
In [1]: U_psi = iswap()
```

To be able to use this unitary transformation matrix as input to the function `qutip.tomography.qpt`, we first need to convert it to a transformation matrix for the corresponding density matrix:

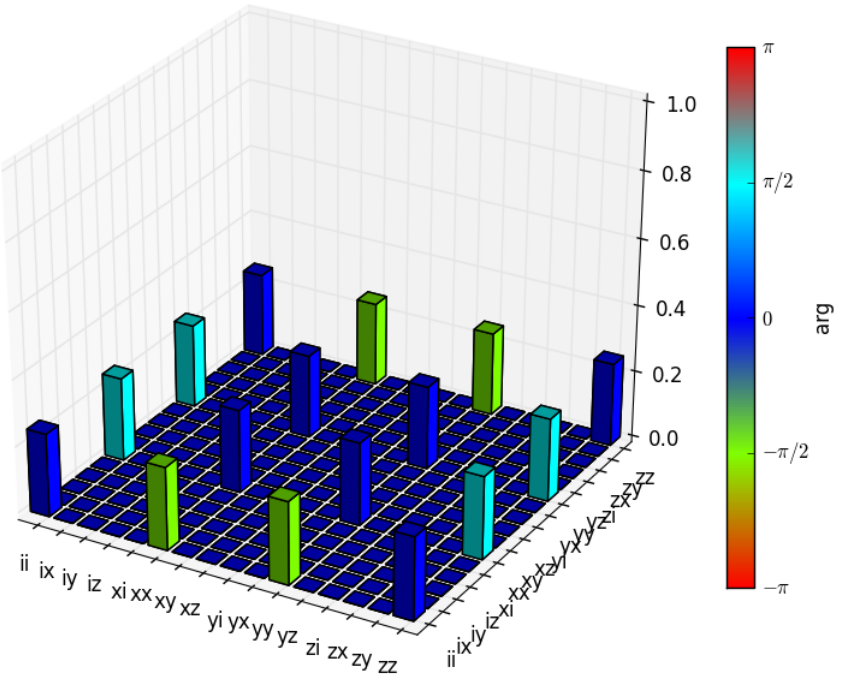
```
In [1]: U_rho = spre(U_psi) * spost(U_psi.dag())
```

Next, we construct a list of operators that define the basis $\{B_i\}$ in the form of a list of operators for each composite system. At the same time, we also construct a list of corresponding labels that will be used when plotting the χ matrix.

```
In [1]: op_basis = [[qeye(2), sigmax(), sigmay(), sigmaz()]] * 2

In [2]: op_label = [["i", "x", "y", "z"]] * 2
```

We are now ready to compute χ using `qutip.tomography.qpt`, and to plot it using `qutip.tomography.qpt_plot_combined`.

```
In [1]: chi = qpt(U_rho, op_basis)

In [2]: fig = qpt_plot_combined(chi, op_label, r'$i$SWAP')

In [3]: show()
```



For a slightly more advanced example, where the density matrix propagator is calculated from the dynamics of a system defined by its Hamiltonian and collapse operators using the function `qutip.propagator.propagator`, see *Process tomography matrix for a 2-qubit iSWAP gate*.

4.11 Using QuTiP's Built-in Parallel for-loop

Often one is interested in the output of a given function as a single-parameter is varied. For instance, in the *Cavity-Qubit Steadystate* example, we calculate the steady-state response as the driving frequency is varied. In cases such as this, where each iteration is independent of the others, we can speedup the calculation by performing the iterations in parallel. In QuTiP, parallel computations may be performed using the `qutip.parfor` (parallel-for-loop) function.

To use the parfor function we need to define a function of a single-variable, and the range over which this variable is to be iterated. For example:

```
In [1]: def func1(x): return x, x**2, x**3

In [2]: [a,b,c] = parfor(func1, range(10))

In [3]: print(a)
[0 1 2 3 4 5 6 7 8 9]

In [4]: print(b)
[ 0  1  4  9 16 25 36 49 64 81]

In [5]: print(c)
[  0   1   8  27  64 125 216 343 512 729]
```

One can also use a single output variable as:

```
In [1]: x = parfor(func1, range(10))

In [2]: print(x[0])
[0 1 2 3 4 5 6 7 8 9]

In [3]: print(x[1])
[ 0  1  4  9 16 25 36 49 64 81]

In [4]: print(x[2])
[  0   1   8  27  64 125 216 343 512 729]
```

The `qutip.parfor` function is not limited to just numbers, but also works for a variety of outputs:

```
In [1]: def func2(x): return x, Qobj(x), 'a' * x

In [2]: [a, b, c] = parfor(func2, range(5))

In [3]: print(a)
[0 1 2 3 4]

In [4]: print(b)
[ Quantum object: dims = [[1], [1]], shape = [1, 1], type = oper, isherm = True
Qobj data =
[[ 0.]]
 Quantum object: dims = [[1], [1]], shape = [1, 1], type = oper, isherm = True
Qobj data =
[[ 1.]]
 Quantum object: dims = [[1], [1]], shape = [1, 1], type = oper, isherm = True
Qobj data =
[[ 2.]]
 Quantum object: dims = [[1], [1]], shape = [1, 1], type = oper, isherm = True
Qobj data =
[[ 3.]]
 Quantum object: dims = [[1], [1]], shape = [1, 1], type = oper, isherm = True
Qobj data =
[[ 4.]]]
```

```
In [5]: print(c)
['' 'a' 'aa' 'aaa' 'aaaa']
```

Although qutip.parfor allows functions with only one input, we can in fact pass more an a single variable by using a list of lists. Sounds confusing, but it is quite easy.

```
In [1]: def func1(args): index, x=args; print(index); return x, x**2, x**3

In [2]: args = [[k, 2 * k] for k in range(10)]  # create list of lists with more than one variabl

In [3]: args
Out[3]:
[[0, 0],
 [1, 2],
 [2, 4],
 [3, 6],
 [4, 8],
 [5, 10],
 [6, 12],
 [7, 14],
 [8, 16],
 [9, 18]]

In [4]: [a, b, c] = parfor(func1, args)

In [5]: print(a)
[ 0  2  4  6  8 10 12 14 16 18]
```

Parfor is also useful for repeated tasks such as generating plots corresponding to the dynamical evolution of your system, or simultaneously simulating different parameter configurations.

4.11.1 IPython-based parfor

When QuTiP is used with IPython interpreter, there is an alternative parallel for-loop implementation in the QuTiP module qutip.ipynbtools, see qutip.ipynbtools.parfor. The advantage of this parfor implementation is based on IPythons powerful framework for parallelization, so the compute processes are not confined to run on the same host as the main process.

4.12 Saving QuTiP Objects and Data Sets

With time-consuming calculations it is often necessary to store the results to files on disk, so it can be post-processed and archived. In QuTiP there are two facilities for storing data: Quantum objects can be stored to files and later read back as python pickles, and numerical data (vectors and matrices) can be exported as plain text files in for example CSV (comma-separated values), TSV (tab-separated values), etc. The former method is preferred when further calculations will be performed with the data, and the latter when the calculations are completed and data is to be imported into a post-processing tool (e.g. for generating figures).

4.12.1 Storing and loading QuTiP objects

To store and load arbitrary QuTiP related objects (qutip.Qobj, qutip.Odedata, etc.) there are two functions: qutip.fileio.qsave and qutip.fileio.qload. The function qutip.fileio.qsave takes an arbitrary object as first parameter and an optional filename as second parameter (default filename is *qutip_data.qu*). The filename extension is always *.qu*. The function qutip.fileio.qload takes a mandatory filename as first argument and loads and returns the objects in the file.

To illustrate how these functions can be used, consider a simple calculation of the steadystate of the harmonic oscillator:

```
In [1]: a = destroy(10); H = a.dag() * a ; c_ops = [sqrt(0.5) * a, sqrt(0.25) * a.dag()]

In [2]: rho_ss = steadystate(H, c_ops)
warning: (almost) singular matrix! (estimated cond. number: 2.11e+15)
```

The steadystate density matrix *rho_ss* is an instance of `qutip.Qobj`. It can be stored to a file *steadystate.qu* using

```
In [1]: qsave(rho_ss, 'steadystate')

In [2]: ls *.qu
density_matrix_vs_time.qu        steadystate.qu
```

and it can later be loaded again, and used in further calculations:

```
In [1]: rho_ss_loaded = qload('steadystate')
Loaded Qobj object:
Quantum object: dims = [[10], [10]], shape = [10, 10], type = oper, isHerm = True


In [2]: a = destroy(10)

In [3]: expect(a.dag() * a, rho_ss_loaded)
Out[3]: 0.9902248289345087
```

The nice thing about the `qutip.fileio.qsave` and `qutip.fileio.qload` functions is that almost any object can be stored and load again later on. We can for example store a list of density matrices as returned by `qutip.mesolve`:

```
In [1]: a = destroy(10); H = a.dag() * a ; c_ops = [sqrt(0.5) * a, sqrt(0.25) * a.dag()]

In [2]: psi0 = rand_ket(10)

In [3]: tlist = linspace(0, 10, 10)

In [4]: dm_list = mesolve(H, psi0, tlist, c_ops, [])

In [5]: qsave(dm_list, 'density_matrix_vs_time')
```

And it can then be loaded and used again, for example in an other program:

```
In [1]: dm_list_loaded = qload('density_matrix_vs_time')
Loaded Odedata object:
Odedata object with mesolve data.
---------------------------------
states = True
num_collapse = 0

In [2]: a = destroy(10)

In [3]: expect(a.dag() * a, dm_list_loaded.states)
Out[3]:
array([ 3.9693692 ,  3.15523135,  2.58863898,  2.17836153,  1.87650168,
        1.65268587,  1.48602212,  1.3615946 ,  1.26854669,  1.19888947])
```

4.12.2 Storing and loading datasets

The `qutip.fileio.qsave` and `qutip.fileio.qload` are great, but the file format used is only understood by QuTiP (python) programs. When data must be exported to other programs the preferred method is to store the data in the commonly used plain-text file formats. With the QuTiP functions `qutip.file_data_store` and `qutip.file_data_read` we can store and load **numpy** arrays and matrices to files on disk using a

deliminator-separated value format (for example comma-separated values CSV). Almost any program can handle this file format.

The `qutip.file_data_store` takes two mandatory and three optional arguments:

```
>>> file_data_store(filename, data, numtype="complex", numformat="decimal", sep=",")
```

where *filename* is the name of the file, *data* is the data to be written to the file (must be a *numpy* array), *numtype* (optional) is a flag indicating numerical type that can take values *complex* or *real*, *numformat* (optional) specifies the numerical format that can take the values *exp* for the format *1.0e1* and *decimal* for the format *10.0*, and *sep* (optional) is an arbitrary single-character field separator (usually a tab, space, comma, semicolon, etc.).

A common use for the `qutip.file_data_store` function is to store the expectation values of a set of operators for a sequence of times, e.g., as returned by the `qutip.mesolve` function, which is what the following example does:

```
In [1]: a = destroy(10); H = a.dag() * a ; c_ops = [sqrt(0.5) * a, sqrt(0.25) * a.dag()]

In [2]: psi0 = rand_ket(10)

In [3]: tlist = linspace(0, 100, 100)

In [4]: medata = mesolve(H, psi0, tlist, c_ops, [a.dag() * a, a + a.dag(), -1j * (a - a.dag())])

In [5]: shape(medata.expect)
Out[5]: (3, 100)

In [6]: shape(tlist)
Out[6]: (100,)

In [7]: output_data = vstack((tlist, medata.expect))    # join time and expt data

In [8]: file_data_store('expect.dat', output_data.T) # Note the .T for transpose!

In [9]: ls *.dat
expect.dat

In [10]: !head expect.dat
# Generated by QuTiP: 100x4 complex matrix in decimal format [',' separated values].
0.0000000000+0.0000000000j,3.9270628029+0.0000000000j,2.4911722134+0.0000000000j,0.0526288333+0.0
1.0101010101+0.0000000000j,3.1653181006+0.0000000000j,1.2265971700+0.0000000000j,-1.7540785216+0.
2.0202020202+0.0000000000j,2.6432133825+0.0000000000j,-0.7052937240+0.0000000000j,-1.7108568933+0
3.0303030303+0.0000000000j,2.2534650107+0.0000000000j,-1.5829838112+0.0000000000j,-0.2782841006+0
4.0404040404+0.0000000000j,1.9582840262+0.0000000000j,-0.9409412662+0.0000000000j,1.0365131887+0.
5.0505050505+0.0000000000j,1.7332920649+0.0000000000j,0.3276358769+0.0000000000j,1.1767046907+0.0
6.0606060606+0.0000000000j,1.5611898058+0.0000000000j,1.0224414436+0.0000000000j,0.3051768696+0.0
7.0707070707+0.0000000000j,1.4292559318+0.0000000000j,0.7016592696+0.0000000000j,-0.6146884939+0.
8.0808080808+0.0000000000j,1.3279705752+0.0000000000j,-0.1286322772+0.0000000000j,-0.8057591818+0
```

In this case we didn't really need to store both the real and imaginary parts, so instead we could use the *numtype="real"* option:

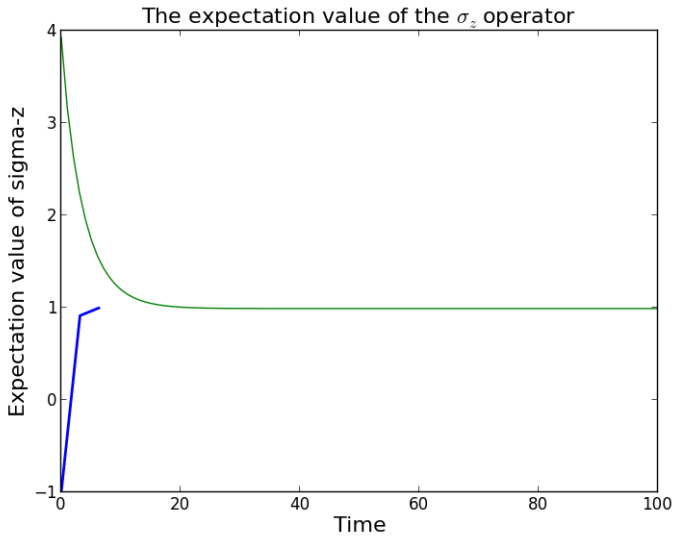
```
In [1]: file_data_store('expect.dat', output_data.T, numtype="real")

In [2]: !head -n5 expect.dat
# Generated by QuTiP: 100x4 real matrix in decimal format [',' separated values].
0.0000000000,3.9270628029,2.4911722134,0.0526288333
1.0101010101,3.1653181006,1.2265971700,-1.7540785216
2.0202020202,2.6432133825,-0.7052937240,-1.7108568933
3.0303030303,2.2534650107,-1.5829838112,-0.2782841006
```

and if we prefer scientific notation we can request that using the *numformat="exp"* option

```
In [1]: file_data_store('expect.dat', output_data.T, numtype="real", numformat="exp")

In [2]: ! head -n 5 expect.dat
# Generated by QuTiP: 100x4 real matrix in exp format [',' separated values].
0.0000000000e+00,3.9270628029e+00,2.4911722134e+00,5.2628833300e-02
1.0101010101e+00,3.1653181006e+00,1.2265971700e+00,-1.7540785216e+00
2.0202020202e+00,2.6432133825e+00,-7.0529372402e-01,-1.7108568933e+00
3.0303030303e+00,2.2534650107e+00,-1.5829838112e+00,-2.7828410064e-01
```

Loading data previously stored using `qutip.file_data_store` (or some other software) is a even easier. Regardless of which deliminator was used, if data was stored as complex or real numbers, if it is in decimal or exponential form, the data can be loaded using the `qutip.file_data_read`, which only takes the filename as mandatory argument.

```
In [1]: input_data = file_data_read('expect.dat')

In [2]: shape(input_data)
Out[2]: (100, 4)

In [3]: from pylab import *

In [4]: plot(input_data[:,0], input_data[:,1]);  # plot the data
Out[4]: [<matplotlib.lines.Line2D at 0x11c696990>]
```



(If a particularly obscure choice of deliminator was used it might be necessary to use the optional second argument, for example *sep*="_" if _ is the deliminator).

4.13 Generating Random Quantum States & Operators

QuTiP includes a collection of random state generators for simulations, theorem evaluation, and code testing:

Function	Description
rand_ket	Random ket-vector
rand_dm	Random density matrix
rand_herm	Random Hermitian matrix
rand_unitary	Random Unitary matrix

See the API documentation: *Random Operators and States* for details.

In all cases, these functions can be called with a single parameter N that indicates a NxN matrix (*rand_dm*, *rand_herm*, *rand_unitary*), or a $Nx1$ vector (*rand_ket*), should be generated. For example:

```
In [1]: rand_ket(5)
Out[1]:
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[-0.61859377+0.27781741j]
 [-0.10179937-0.029525j  ]
 [-0.23271574-0.01080088j]
 [-0.32786446+0.05990194j]
 [-0.56806336+0.20216773j]]
```

or

```
In [1]: rand_herm(5)
Out[1]:
Quantum object: dims = [[5], [5]], shape = [5, 5], type = oper, isherm = True
Qobj data =
[[ 0.00000000+0.j         -0.01838978+0.23642313j  0.00000000+0.j
   0.00000000+0.j         -0.38595091-0.17504323j]
 [-0.01838978-0.23642313j -0.32106831+0.j         -0.50495285-0.03332274j
  -0.36673595-0.08724806j  0.00000000+0.j        ]
 [ 0.00000000+0.j         -0.50495285+0.03332274j -0.07622525+0.j
  -0.20588139+0.06755127j -0.78145714-0.05807419j]
 [ 0.00000000+0.j         -0.36673595+0.08724806j -0.20588139-0.06755127j
  -0.56523034+0.j         -0.40709747-0.11377031j]
 [-0.38595091+0.17504323j  0.00000000+0.j         -0.78145714+0.05807419j
  -0.40709747+0.11377031j -0.11054850+0.j        ]]
```

In this previous example, we see that the generated Hermitian operator contains a fraction of elements that are identically equal to zero. The number of nonzero elements is called the *density* and can be controlled by calling any of the random state/operator generators with a second argument between 0 and 1. By default, the density for the operators is *0.75* where as ket vectors are completely dense (*1*). For example:

```
In [1]: rand_dm(5, 0.5)
Out[1]:
Quantum object: dims = [[5], [5]], shape = [5, 5], type = oper, isherm = True
Qobj data =
[[ 0.04820518+0.j          0.10817515-0.00166962j  0.09524373-0.00805622j
   0.00000000+0.j          0.00000000+0.j        ]
 [ 0.10817515+0.00166962j  0.24280899+0.j          0.21401135-0.01477978j
   0.00000000+0.j          0.00000000+0.j        ]
 [ 0.09524373+0.00805622j  0.21401135+0.01477978j  0.18952881+0.j
   0.00000000+0.j          0.00000000+0.j        ]
 [ 0.00000000+0.j          0.00000000+0.j          0.00000000+0.j
   0.03891402+0.j          0.00000000+0.j        ]
 [ 0.00000000+0.j          0.00000000+0.j          0.00000000+0.j
   0.00000000+0.j          0.48054299+0.j        ]]
```

has rougly half nonzero elements, or equivalently a density of 0.5.

Important: In the case of a density matrix, setting the density too low will result in not enough diagonal elements to satisfy $Tr(\rho) = 1$.

4.13.1 Composite random objects

In many cases, one is interested in generating random quantum objects that correspond to composite systems generated using the `qutip.tensor.tensor` function. Specifying the tensor structure of a quantum object is

done using the *dims* keyword argument in the same fashion as one would do for a `qutip.Qobj` object:

```
In [1]: rand_dm(4, 0.5, dims=[[2,2], [2,2]])
Out[1]:
Quantum object: dims = [[2, 2], [2, 2]], shape = [4, 4], type = oper, isherm = True
Qobj data =
[[ 0.06000795+0.j         0.00000000+0.j         0.00000000+0.j
   0.17864934-0.1445353j]
 [ 0.00000000+0.j         0.00000000+0.j         0.00000000+0.j
   0.00000000+0.j       ]
 [ 0.00000000+0.j         0.00000000+0.j         0.00000000+0.j
   0.00000000+0.j       ]
 [ 0.17864934+0.1445353j  0.00000000+0.j         0.00000000+0.j
   0.93999205+0.j       ]]
```

4.14 Modifying Internal QuTiP Settings

4.14.1 User Accessible Parameters

Note: This section deals with modifying the internal QuTiP settings. Use only if you know what you are doing.

In this section we show how to modify a few of the internal parameters used by QuTiP. The settings that can be modified are given in the following table:

Setting	Description	Options
qutip_graphics	Use matplotlib	True / False
qutip_gui	Pick GUI library, or disable.	"PYSIDE","PYQT4","NONE"
auto_herm	Automatically calculate the hermicity of quantum objects.	True / False
auto_tidyup	Automatically tidyup quantum objects.	True / False
auto_tidyup_atol	Tolerance used by tidyup	any *float* value > 0
num_cpus	Number of CPU's used for multiprocessing.	*int* between 1 and # cpu's
debug	Show debug printouts.	True / False

4.14.2 Example: Changing Settings

The two most important settings are `auto_tidyup` and `auto_tidyup_atol` as they control whether the small elements of a quantum object should be removed, and what number should be considered as the cut-off tolerance. Modifying these, or any other parameters, is quite simple:

```
>>> qutip.settings.auto_tidyup = False
>>> qutip.settings.qutip_gui = "NONE"
```

These settings will be used for the current QuTiP session only and will need to be modified again when restarting QuTiP. If running QuTiP from a script file, then place the *qutip.setings.xxxx* commands immediately after *from qutip import ** at the top of the script file. If you want to reset the parameters back to their default values then call the reset command:

```
>>> qutip.settings.reset()
```

4.14.3 Persistent Settings

When QuTiP is imported, it looks for the file `.qutiprc` in the user's home directory. If this file is found, it will be loaded and overwrite the QuTiP default settings, which allows for persistent changes in the QuTiP settings to

be made. A sample `.qutiprc` file is show below. The syntax is a simple key-value format, where the keys and possible values are described in the table above:

```
# QuTiP Graphics
qutip_graphics="YES"
# QuTiP GUI selection
qutip_gui=PYSIDE
# use auto tidyup
auto_tidyup=True
# detect hermiticity
auto_herm=True
# use auto tidyup absolute tolerance
auto_tidyup_atol=1e-12
# number of cpus
num_cpus=4
# debug
debug=False
```

CHAPTER

FIVE

QUTIP EXAMPLE SCRIPTS

5.1 Running Examples

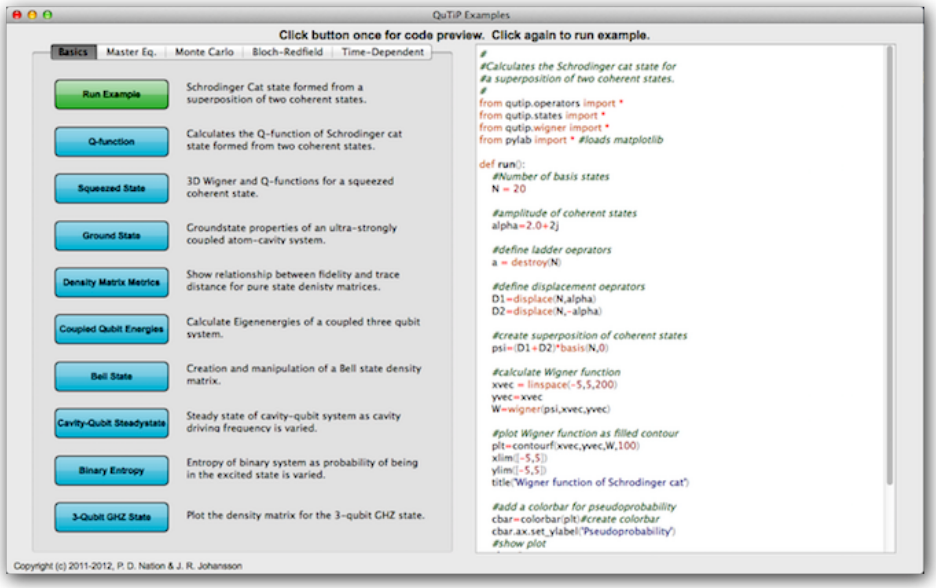
QuTip includes 20 built in demos from the examples below that demonstrate the usage of the built in functions for solving a variety of problems. To run the demos, load the QuTiP package:

```
>>> from qutip import *
```

and run the demos function:

```
>>> demos()
```

This will generate the examples GUI, or a command line list of demos, depending on the availability of the graphics libraries:



If you do not have any graphics libraries installed, or they are disabled, then the demos function *must* be run from the terminal.

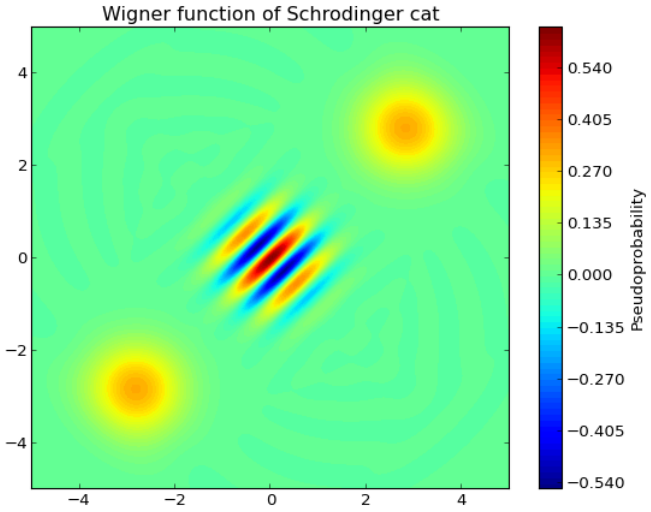
5.2 List of Built-in Examples

5.2.1 Basic Examples

Wigner Function for Schrödinger Cat State

Calculates the Wigner distribution for a Schrödinger Cat state composed of two coherent states $\alpha_1 = -2 - 2j$, and $\alpha_2 = 2 + 2j$.

```
#
# Calculates the Schrodinger cat state for
# a superposition of two coherent states.
#
from qutip import *
from pylab import *  # loads matplotlib


def run():
    # Number of basis states
    N = 20

    # amplitude of coherent states
    alpha = 2.0 + 2j

    # define displacement oeprators
    D1 = displace(N, alpha)
    D2 = displace(N, -alpha)

    # create superposition of coherent states
    psi = (D1 + D2) * basis(N, 0)

    # calculate Wigner function
    xvec = linspace(-5, 5, 200)
    yvec = xvec
    W = wigner(psi, xvec, yvec)

    # plot Wigner function as filled contour
    plt = contourf(xvec, yvec, W, 100)
    xlim([-5, 5])
    ylim([-5, 5])
    title('Wigner function of Schrodinger cat')

    # add a colorbar for pseudoprobability
    cbar = colorbar(plt)  # create colorbar
    cbar.ax.set_ylabel('Pseudoprobability')
    # show plot
    show()

if __name__ == "__main__":
    run()
```

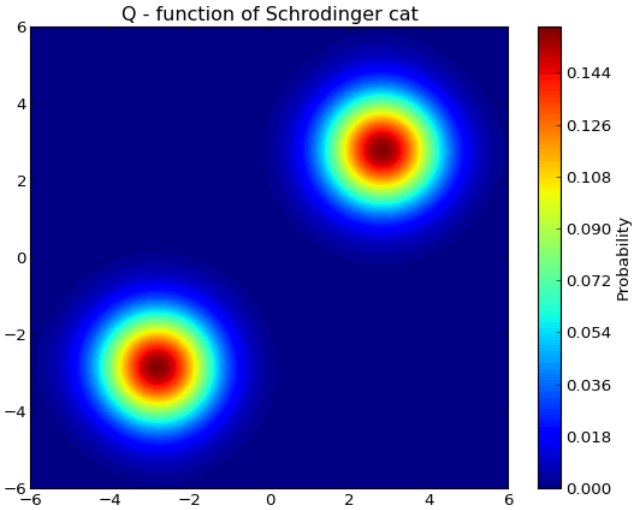


Q Function for Schrödinger Cat State

Calculates the Q distribution for a Schrödinger Cat state composed of two coherent states $\alpha_1 = -2 - 2j$, and $\alpha_2 = 2 + 2j$.

```
#
# Calculates the Q-function of Schrodinger cat state
# formed from a superposition of two coherent states.
#
from qutip import *
from pylab import *  # loads matplotlib


def run():
    # Number of basis states
    N = 20

    # amplitude of coherent states
    alpha = 2.0 + 2j

    # define displacement oeprators
    D1 = displace(N, alpha)
    D2 = displace(N, -alpha)

    # create superposition of coherent states
    psi = (D1 + D2) * basis(N, 0)

    # calculate Q-function
    xvec = linspace(-6, 6, 200)
    yvec = xvec
    Q = qfunc(psi, xvec, yvec)

    # plot Q-function as filled contour
    plt = contourf(xvec, yvec, Q, 100)
    xlim([-6, 6])
    ylim([-6, 6])
    title('Q - function of Schrodinger cat')

    # add a colorbar for pseudoprobability
    cbar = colorbar(plt)  # create colorbar
    cbar.ax.set_ylabel('Probability')
```

```
    # show plot
    show()

if __name__ == "__main__":
    run()
```

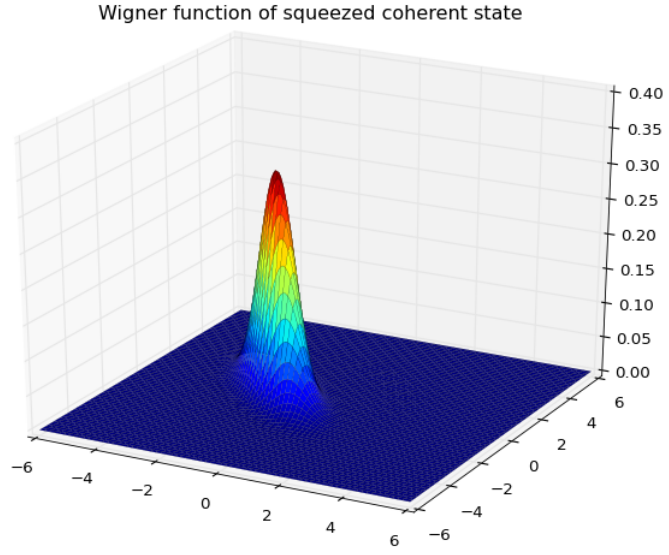


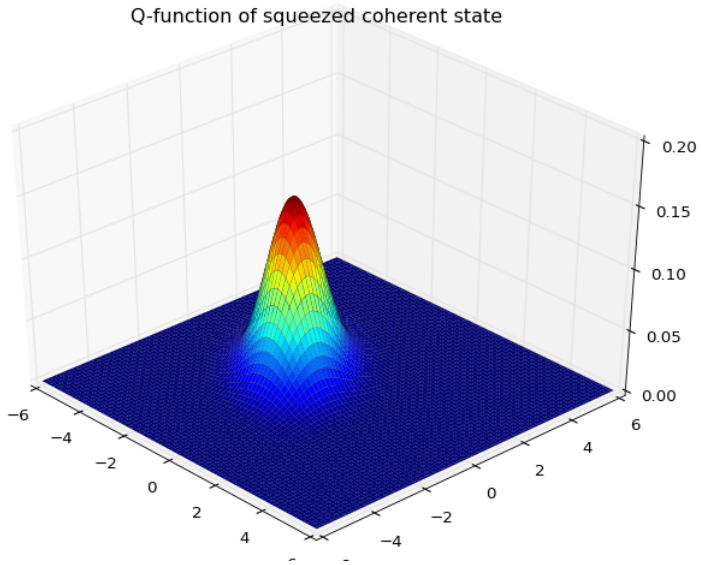
Squeezed State

3D Wigner and Q functions for a squeezed coherent state where $\alpha = -1.0$ is the coherent state amplitude and $\epsilon = 0.5j$ is the squeezing parameter.

```
#
# 3D Wigner and Q-functions for
# a squeezed coherent state.
#
from qutip import *
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from pylab import *


def run():
    # setup constants:
    N = 20
    alpha = -1.0  # Coherent amplitude of field
    epsilon = 0.5j  # Squeezing parameter

    D = displace(N, alpha)      # Displacement
    S = squeez(N, epsilon)      # Squeezing
    psi = D * S * basis(N, 0)   # Apply to vacuum state

    xvec = linspace(-6, 6, 150)
    X, Y = meshgrid(xvec, xvec)
    W = wigner(psi, xvec, xvec)

    Q = qfunc(psi, xvec, xvec)
    fig = figure()
    ax = Axes3D(fig, azim=-62, elev=25)
    ax.plot_surface(X, Y, W, rstride=2, cstride=2, cmap=cm.jet, lw=.1)
    ax.set_xlim3d(-6, 6)
    ax.set_xlim3d(-6, 6)
```

```
    ax.set_zlim3d(0, 0.4)
    title('Wigner function of squeezed coherent state')
    show()

    fig = figure()
    ax2 = Axes3D(fig, azim=-43, elev=37)
    ax2.plot_surface(X, Y, Q, rstride=2, cstride=2, cmap=cm.jet, lw=.1)
    ax2.set_xlim3d(-6, 6)
    ax2.set_xlim3d(-6, 6)
    ax2.set_zlim3d(0, 0.2)
    title('Q-function of squeezed coherent state')
    show()

if __name__ == "__main__":
    run()
```



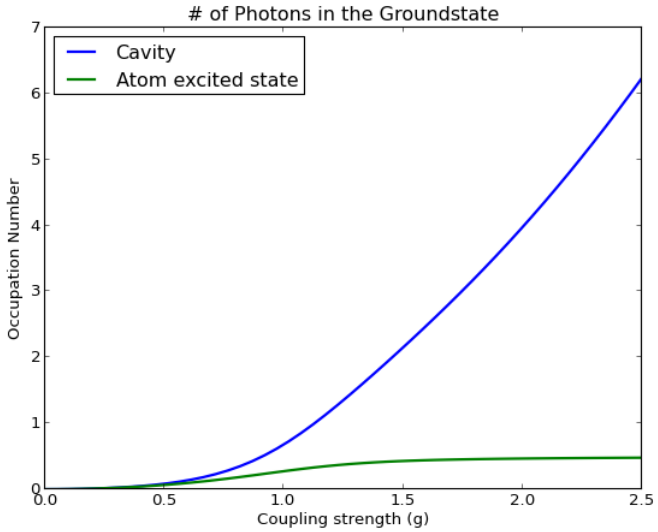


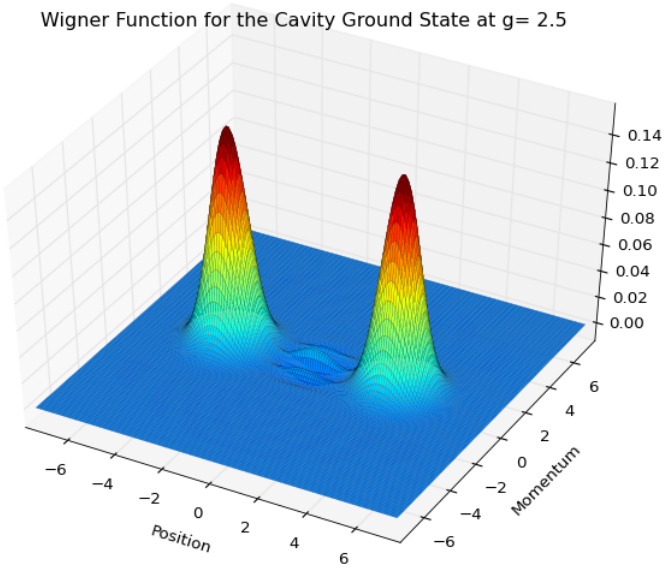
Ground State of Cavity + Qubit

Groundstate properties of an ultra-strongly coupled atom-cavity system as the coupling strength is varied.

```
#
# Groundstate properties of an ultra-strongly coupled atom-cavity system.
#
from qutip import *
from pylab import *
from mpl_toolkits.mplot3d import Axes3D


def compute(N, wc, wa, glist, use_rwa):

    # Pre-compute operators for the hamiltonian
    a = tensor(destroy(N), qeye(2))
    sm = tensor(qeye(N), destroy(2))
    nc = a.dag() * a
    na = sm.dag() * sm

    na_expt = zeros(len(glist))
    nc_expt = zeros(len(glist))
    grnd_kets = zeros(len(glist), dtype=object)
    for idx, g in enumerate(glist):

        # recalculate the hamiltonian for each value of g in glist
        # use non-RWA Hamiltonian
        if use_rwa:
            H = wc * nc + wa * na + g * (a.dag() * sm + a * sm.dag())
        else:
            H = wc * nc + wa * na + g * (a.dag() + a) * (sm + sm.dag())

        # find the groundstate of the composite system
        gval, grndstate = H.groundstate()
        # ground state expectation values for qubit and cavity occupation
        # number
        na_expt[idx] = expect(na, grndstate)
        nc_expt[idx] = expect(nc, grndstate)
        grnd_kets[idx] = grndstate

    return nc_expt, na_expt, grnd_kets


def run():
    #
    # set up the calculation
    #
    wc = 1.0 * 2 * pi    # cavity frequency
    wa = 1.0 * 2 * pi    # atom frequency
    N = 25               # number of cavity fock states
    use_rwa = False      # Set to True to see that non-RWA is necessary

    # array of coupling strengths to calcualate ground state
    glist = linspace(0, 2.5, 50) * 2 * pi

    # run computation
    nc, na, grnd_kets = compute(N, wc, wa, glist, use_rwa)

    # plot the cavity and atom occupation numbers as a function of
    # coupling strength
    figure(1)
    plot(glist / (2 * pi), nc, lw=2)
    plot(glist / (2 * pi), na, lw=2)
    legend(("Cavity", "Atom excited state"), loc=0)
    xlabel('Coupling strength (g)')
    ylabel('Occupation Number')
    title('# of Photons in the Groundstate')
```

```
    show()

    # partial trace over qubit
    rho_cavity = ptrace(grnd_kets[-1], 0)

    # calculate Wigner function for cavity mode
    xvec = linspace(-7.5, 7.5, 150)
    X, Y = meshgrid(xvec, xvec)
    W = wigner(rho_cavity, xvec, xvec)

    # plot Wigner function
    fig = figure()
    ax = Axes3D(fig, azim=-61, elev=43)
    ax.plot_surface(X, Y, W, rstride=1, cstride=1, cmap=cm.jet,
                    linewidth=0.1, vmax=0.15, vmin=-0.05)
    title("Wigner Function for the Cavity Ground State at g= " +
          str(1. / (2 * pi) * glist[-1]))
    ax.set_xlabel('Position')
    ax.set_ylabel('Momentum')
    show()


if __name__ == "__main__":
    run()
```

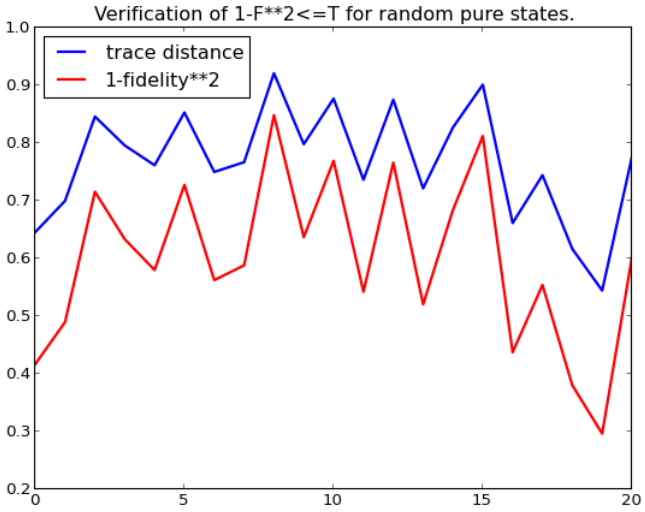




Density Matrix Metrics

Shows the relationship $1 - F^2 \leq T$ between fidelity F and trace distance T for pure-state density matrices.

```
#
# Prove that 1-F**2 <= T for pure state density matricies
# where F and T are the fidelity and trace distance metrics,
# respectively using randomly generated ket vectors.
#
from qutip import *
from pylab import *


def run():
    N = 21  # number of kets to generate

    # create arrays of pure density matrices from random kets using ket2dm
    x = array([ket2dm(rand_ket(10)) for k in range(N)])
    y = array([ket2dm(rand_ket(10)) for k in range(N)])

    # calculate trace distance and fidelity between states in x & y
    T = array([tracedist(x[k], y[k]) for k in range(N)])
    F = array([fidelity(x[k], y[k]) for k in range(N)])

    # plot T and 1-F**2 where x=range(N)
    plot(range(N), T, 'b', range(N), 1 - F ** 2, 'r', lw=2)
    title("Verification of 1-F**2<=T for random pure states.")
    legend(("trace distance", "1-fidelity**2"), loc=0)
    show()

if __name__ == "__main__":
    run()
```

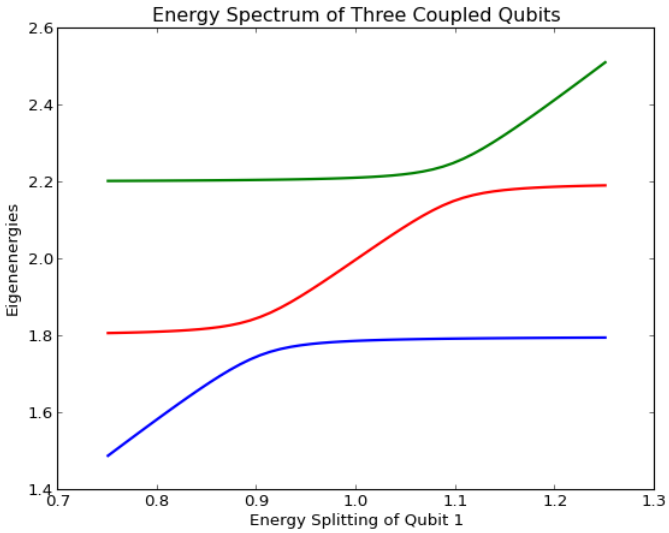


Energy-levels of a Coupled-Qubit System

Calculates the eigenenergies of a coupled three-qubit system.

```
#
# Energy spectrum of three coupled qubits.
#
from qutip import *
from pylab import *


def compute(w1list, w2, w3, g12, g13):
    # Pre-compute operators for the hamiltonian

    # qubit 1 operators
    sz1 = tensor(sigmaz(), qeye(2), qeye(2))
    sx1 = tensor(sigmax(), qeye(2), qeye(2))

    # qubit 2 operators
    sz2 = tensor(qeye(2), sigmaz(), qeye(2))
    sx2 = tensor(qeye(2), sigmax(), qeye(2))

    # qubit 3 operators
    sz3 = tensor(qeye(2), qeye(2), sigmaz())
    sx3 = tensor(qeye(2), qeye(2), sigmax())

    # preallocate output array
    evals_mat = zeros((len(w1list), 2 * 2 * 2))
    for idx, w1 in enumerate(w1list):

        # evaluate the Hamiltonian
        H = w1 * sz1 + w2 * sz2 + w3 * sz3 + g12 * sx1 * sx2 + g13 * sx1 * sx3

        # find the energy eigenvalues and vectors of the composite system
        evals, evecs = H.eigenstates()
        evals_mat[idx, :] = evals

    return evals_mat


def run():
    #
```

```
    # set up the calculation
    #
    w1 = 1.0 * 2 * pi    # atom 1 frequency: sweep this one
    w2 = 0.9 * 2 * pi    # atom 2 frequency
    w3 = 1.1 * 2 * pi    # atom 3 frequency
    g12 = 0.05 * 2 * pi    # atom1-atom2 coupling strength
    g13 = 0.05 * 2 * pi    # atom1-atom3 coupling strength

    # range of qubit 1 frequencies
    w1list = linspace(0.75, 1.25, 50) * 2 * pi

    # run computation
    evals_mat = compute(w1list, w2, w3, g12, g13)

    #
    # plot the energy eigenvalues
    #
    figure(1)
    colors = ['b', 'r', 'g']  # list of colors for plotting
    for n in [1, 2, 3]:
        plot(w1list / (2 * pi), (evals_mat[:, n] - evals_mat[:, 0]) / (2 * pi),
             colors[n - 1], lw=2)

    xlabel('Energy Splitting of Qubit 1')
    ylabel('Eigenenergies')
    title('Energy Spectrum of Three Coupled Qubits')

    show()


if __name__ == "__main__":
    run()
```



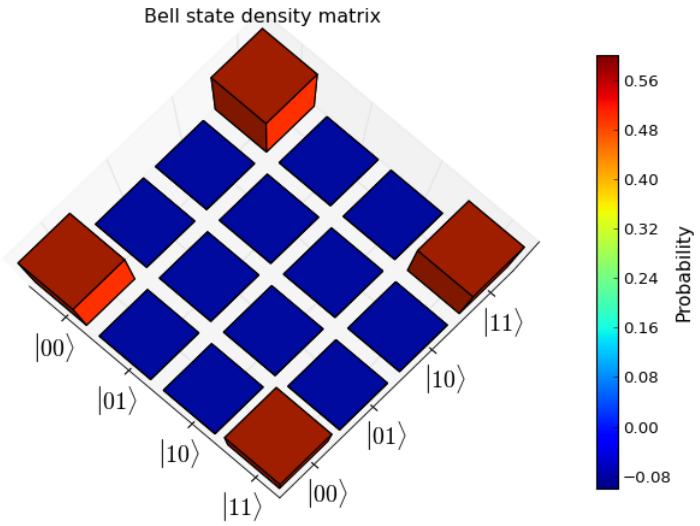
Bell State

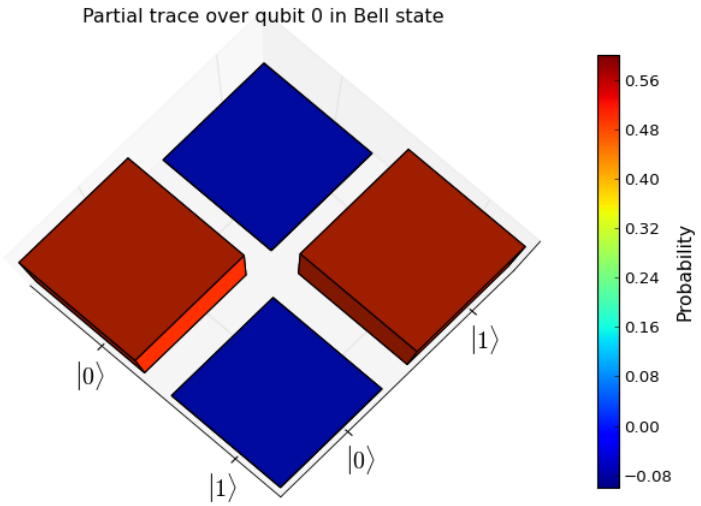
Creation and manipulation of a Bell state density matrix. The Bell state is given as $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$

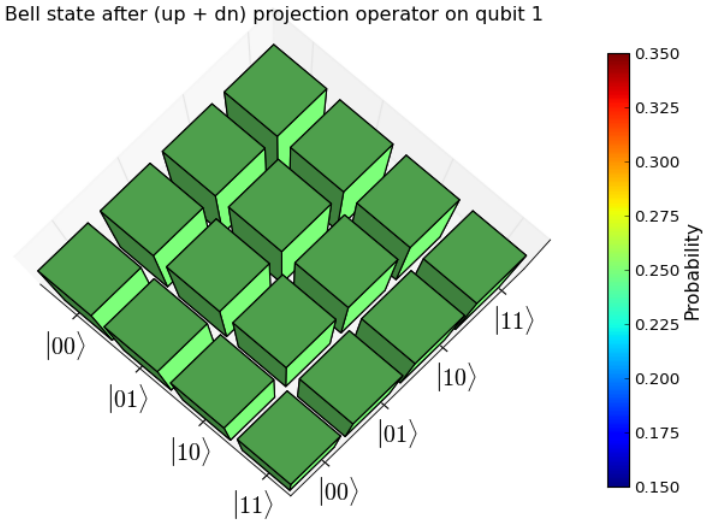
```
#
# Creation and manipulation of a Bell state with
# 3D histogram plot output.
#
```

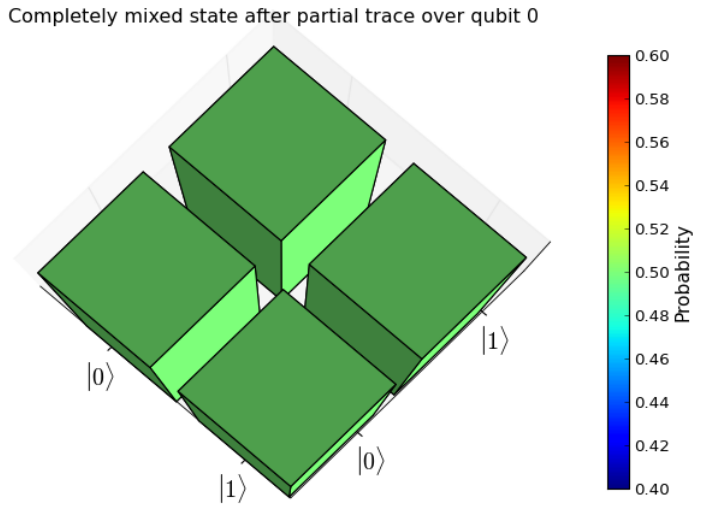
```
from qutip import *
from pylab import *
import matplotlib as mpl
from matplotlib import pyplot, cm
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt


def qubit_hist(Q, xlabels, ylabels, title):
    # Plots density matrix 3D histogram from Qobj
    # xlabels and ylabels are list of strings for axes tick labels
    num_elem = prod(Q.shape)        # num. of elements to plot
    xpos, ypos = meshgrid(range(Q.shape[0]), range(Q.shape[1]))
    xpos = xpos.T.flatten() - 0.5  # center bars on integer value of x-axis
    ypos = ypos.T.flatten() - 0.5  # center bars on integer value of y-axis
    zpos = zeros(num_elem)          # all bars start at z=0
    dx = 0.8 * ones(num_elem)       # width of bars in x-direction
    dy = dx.copy()  # width of bars in y-direction (same as x-dir here)
    dz = real(Q.full().flatten())  # height of bars from density matrix

    # generate list of colors for probability data
    # add +-0.1 in case all elements are the same (colorbar will fail).
    nrm = mpl.colors.Normalize(min(dz) - 0.1, max(dz) + 0.1)
    colors = cm.jet(nrm(dz))

    # plot figure
    fig = plt.figure()
    ax = Axes3D(fig, azim=-47, elev=85)
    ax.bar3d(xpos, ypos, zpos, dx, dy, dz, color=colors)

    # set x-axis tick labels and label font size. set x-ticks to integers
    ax.axes.w_xaxis.set_major_locator(IndexLocator(1, -0.5))
    ax.set_xticklabels(xlabels)
    ax.tick_params(axis='x', labelsize=18)

    # set y-axis tick labels and label font size. set y-ticks to integers
    ax.axes.w_yaxis.set_major_locator(IndexLocator(1, -0.5))
    ax.set_yticklabels(ylabels)
    ax.tick_params(axis='y', labelsize=18)

    # remove z-axis tick labels by moving them outside the plot range
    ax.axes.w_zaxis.set_major_locator(IndexLocator(2, 2))
    # set the plot range in the z-direction to fit data
    ax.set_zlim3d([min(dz) - 0.1, max(dz) + 0.1])
    plt.title(title)
    # add colorbar with color range normalized to data
    cax, kw = mpl.colorbar.make_axes(ax, shrink=.75, pad=.02)
    cb1 = mpl.colorbar.ColorbarBase(cax, cmap=cm.jet, norm=nrm)
    cb1.set_label("Probability", fontsize=14)
    show()


def run():
    # create Bell state
    up = basis(2, 0)
    dn = basis(2, 1)
    bell = (tensor([up, up]) + tensor([dn, dn])).unit()
    rho_bell = ket2dm(bell)

    x_bell_labels = [
        '$\left|00\\rangle\\right.$', '$\left|01\\rangle\\right.$',
        '$\left|10\\rangle\\right.$', '$\left|11\\rangle\\right.$']
    y_bell_labels = x_bell_labels
```

```
    title = 'Bell state density matrix'
    # plot Bell state density matrix
    qubit_hist(rho_bell, x_bell_labels, y_bell_labels, title)

    # trace over qubit 0
    bell_trace_1 = ptrace(rho_bell, 1)
    xlabels = ['$\left|0\\rangle\\right.$', '$\left|1\\rangle\\right.$']
    ylabels = xlabels
    title = 'Partial trace over qubit 0 in Bell state'
    # plot remaining qubit density matrix
    qubit_hist(bell_trace_1, xlabels, ylabels, title)

    # create projection operator
    left = (up + dn).unit()
    Omegaleft = tensor(qeye(2), left * left.dag())
    after = Omegaleft * bell
    after = ket2dm(after / after.norm())
    title = "Bell state after (up + dn) projection operator on qubit 1"
    # plot density matrix after projection
    qubit_hist(after, x_bell_labels, y_bell_labels, title)

    # plot partial trace of state 1
    after_trace = ptrace(after, 1)
    title = "Completely mixed state after partial trace over qubit 0"
    qubit_hist(after_trace, xlabels, ylabels, title)


if __name__ == '__main__':
    run()
```




Partial trace over qubit 0 in Bell state
|0⟩
|1⟩
|0⟩
|1⟩
0.56
0.48
0.40
0.32
0.24
0.16
0.08
0.00
−0.08
Probability


Bell state after (up + dn) projection operator on qubit 1
|00⟩
|01⟩
|10⟩
|11⟩
|00⟩
|01⟩
|10⟩
|11⟩
0.350
0.325
0.300
0.275
0.250
0.225
0.200
0.175
0.150
Probability

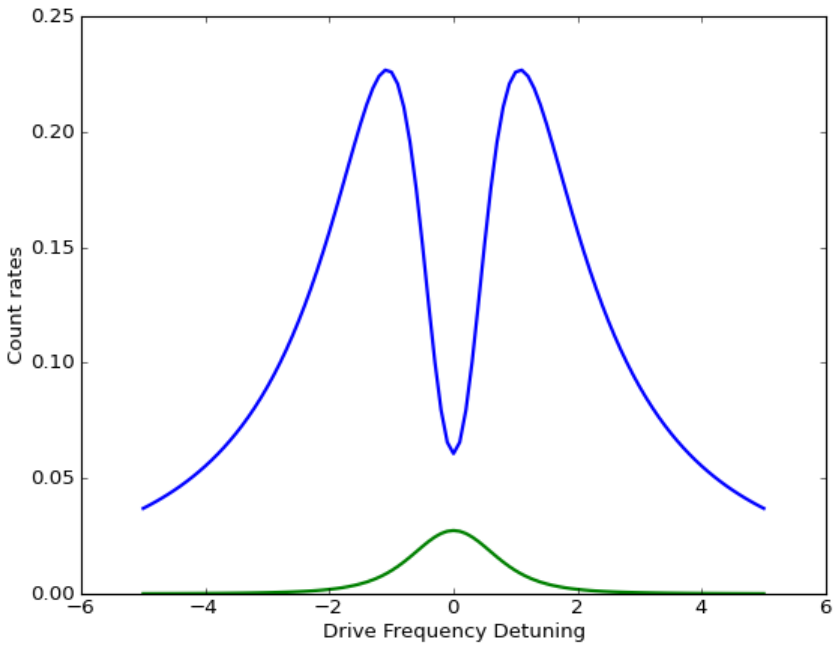

Completely mixed state after partial trace over qubit 0
|0⟩
|1⟩
|0⟩
|1⟩
0.60
0.58
0.56
0.54
0.52
0.50
0.48
0.46
0.44
0.42
0.40
Probability

Cavity-Qubit Steadystate

Find the steady state of a cavity-qubit system as a function of the cavity driving frequency.

```
#
# Steady-state density matrix of a two-level atom in a high-Q
# cavity for various driving frequencies calculated using
# iterative 'steady' solver.
#
# Adapted from 'probss' example in the qotoolbox by Sze M. Tan.
#
from qutip import *
from pylab import *


def probss(E, kappa, gamma, g, wc, w0, wl, N):
    # construct composite operators
    ida = qeye(N)
    idatom = qeye(2)
    a = tensor(destroy(N), idatom)
    sm = tensor(ida, sigmam())
    # Hamiltonian
    H = (w0 - wl) * sm.dag() * sm + (wc - wl) * a.dag() * a + \
        1j * g * (a.dag() * sm - sm.dag() * a) + E * (a.dag() + a)

    # Collapse operators
    C1 = sqrt(2 * kappa) * a
    C2 = sqrt(gamma) * sm
    C1dC1 = C1.dag() * C1
    C2dC2 = C2.dag() * C2

    # find steady state
    rhoss = steadystate(H, [C1, C2])

    # calculate expectation values
    count1 = expect(C1dC1, rhoss)
    count2 = expect(C2dC2, rhoss)
    infield = expect(a, rhoss)
    return count1, count2, infield


# setup the calculation
#-----------------------
# must be done before parfunc unless we
# want to pass all variables as one using
# zip function (see documentation for an example)
kappa = 2
gamma = 0.2
g = 1
wc = 0
w0 = 0
N = 5
E = 0.5
nloop = 101
wlist = linspace(-5, 5, nloop)


# define single-variable function for use in parfor
# cannot be defined inside run() since it needs to
# be passed into seperate threads.
def parfunc(wl):  # function of wl only
    count1, count2, infield = probss(E, kappa, gamma, g, wc, w0, wl, N)
    return count1, count2, infield
```

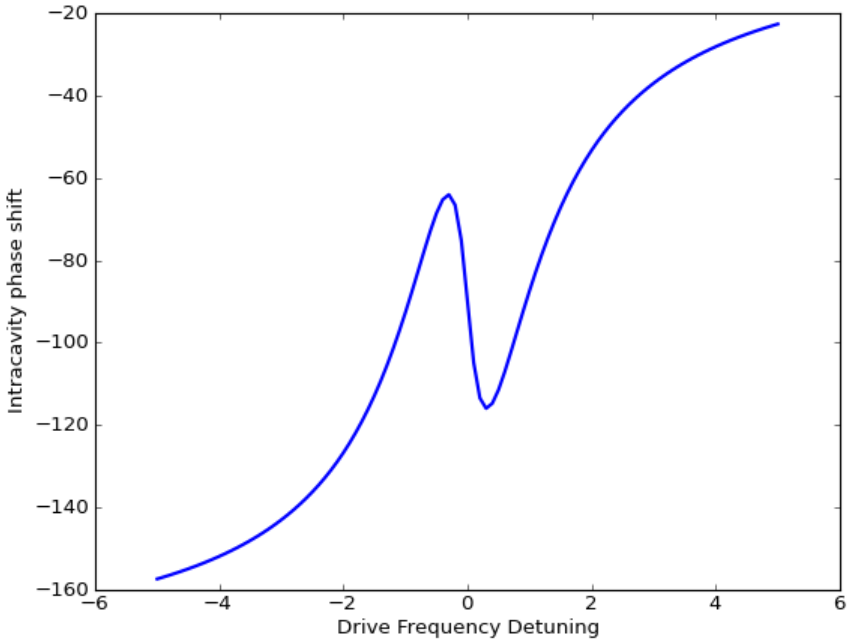
```
def run():
    # run parallel for-loop over wlist
    count1, count2, infield = parfor(parfunc, wlist)

    # plot cavity emission and qubit spontaneous emssion
    fig = figure(1)
    ax = fig.add_subplot(111)
    ax.plot(wlist, count1, wlist, count2, lw=2)
    xlabel('Drive Frequency Detuning')
    ylabel('Count rates')
    show()

    # plot phase shift of cavity light
    fig2 = figure(2)
    ax2 = fig2.add_subplot(111)
    ax2.plot(wlist, 180.0 * angle(infield) / pi, lw=2)
    xlabel('Drive Frequency Detuning')
    ylabel('Intracavity phase shift')
    show()


if __name__ == "__main__":
    run()
```



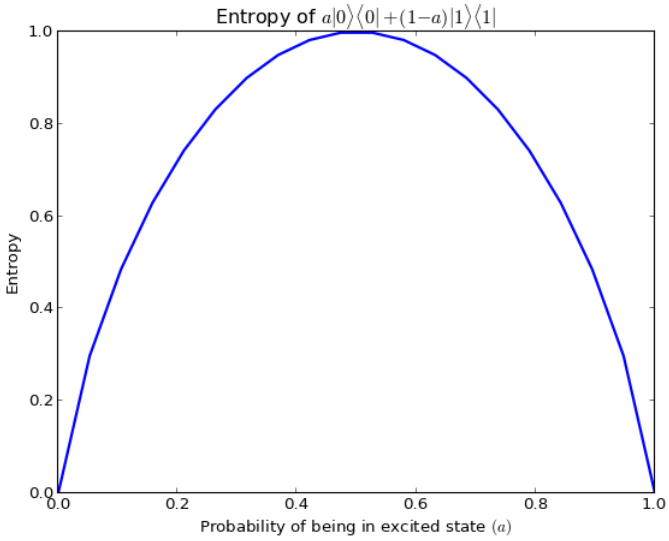
Binary Entropy

Entropy of a binary system $a\left|0\right\rangle\left\langle 0\right| + (1-a)\left|1\right\rangle\left\langle 1\right|$ as probability of being in the excited state a is varied.

```
#
# Entropy of binary system as probability
# of being in the excited state is varied.
#
from qutip import *
from pylab import *
```




```
def run():
    a = linspace(0, 1, 20)
    out = zeros(len(a))  # preallocate output array
    for k in range(len(a)):
        # a*|0><0|
        x = a[k] * ket2dm(basis(2, 0))
        # (1-a)*|1><1|
        y = (1 - a[k]) * ket2dm(basis(2, 1))
        rho = x + y
        # Von-Neumann entropy (base 2) of rho
        out[k] = entropy_vn(rho, 2)

    figure()
    plot(a, out, lw=2)
    xlabel(r'Probability of being in excited state $(a)$')
    ylabel(r'Entropy')
    title("Entropy of $a|0\\rangle\langle0|+(1-a)|1\\rangle\langle1|$")
    show()

if __name__ == '__main__':
    run()
```

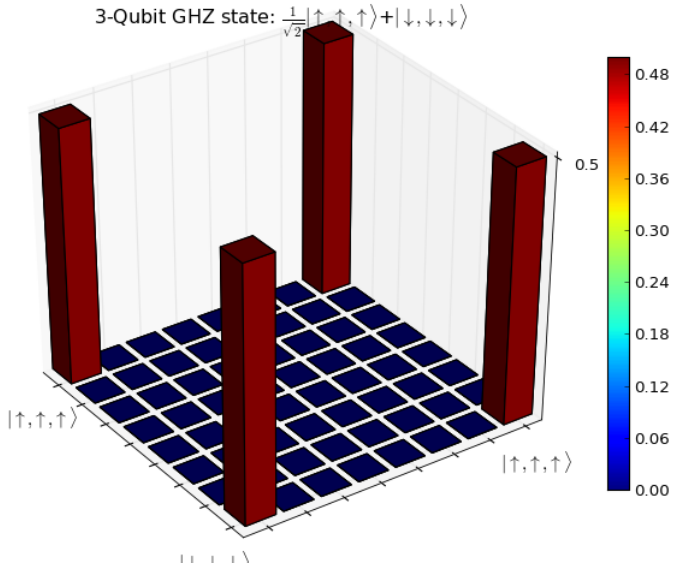


3-Qubit GHZ State

Plot the density matrix for the 3-qubit GHZ state $\frac{1}{\sqrt{2}}\left(|\uparrow,\uparrow,\uparrow\rangle + |\downarrow,\downarrow,\downarrow\rangle\right)$ found by simultaneous diagonalization.

```
#
# Plots the entangled superposition
# 3-qubit GHZ eigenstate |up,up,up> + |dn,dn,dn>
#
# From the xGHZ qotoolbox example by Sze M. Tan
#
from qutip import *
from pylab import *


def run():
    # create spin operators for the three qubits.
    sx1 = tensor(sigmax(), qeye(2), qeye(2))
    sy1 = tensor(sigmay(), qeye(2), qeye(2))

    sx2 = tensor(qeye(2), sigmax(), qeye(2))
    sy2 = tensor(qeye(2), sigmay(), qeye(2))

    sx3 = tensor(qeye(2), qeye(2), sigmax())
    sy3 = tensor(qeye(2), qeye(2), sigmay())

    # Calculate products
    op1 = sx1 * sy2 * sy3
    op2 = sy1 * sx2 * sy3
    op3 = sy1 * sy2 * sx3
    opghz = sx1 * sx2 * sx3

    # Find simultaneous eigenkets of op1,op2,op3 and opghz
    evalues, states = simdiag([op1, op2, op3, opghz])

    # convert last eigenstate to density matrix
    rho0 = ket2dm(states[-1])
    # create labels for density matrix plot
    upupup = "$|\\uparrow,\\uparrow,\\uparrow\\rangle$"
    dndndn = "$|\\downarrow,\\downarrow,\\downarrow\\rangle$"
    title = "3-Qubit GHZ state: $\\frac{1}{\\sqrt{2}}$" + upupup + "+" + dndndn
    xlabels = [""] * 8
```

```
    xlabels[0] = upupup  # set first xaxes label
    xlabels[-1] = dndndn  # set last xaxes label
    ylabels = [""] * 8
    ylabels[-1] = upupup  # set last yaxes label
    # generate plot with labels
    matrix_histogram(rho0, xlabels=xlabels, ylabels=ylabels, title=title)
    show()

if __name__ == '__main__':
    run()
```



5.2.2 Master Equation Examples

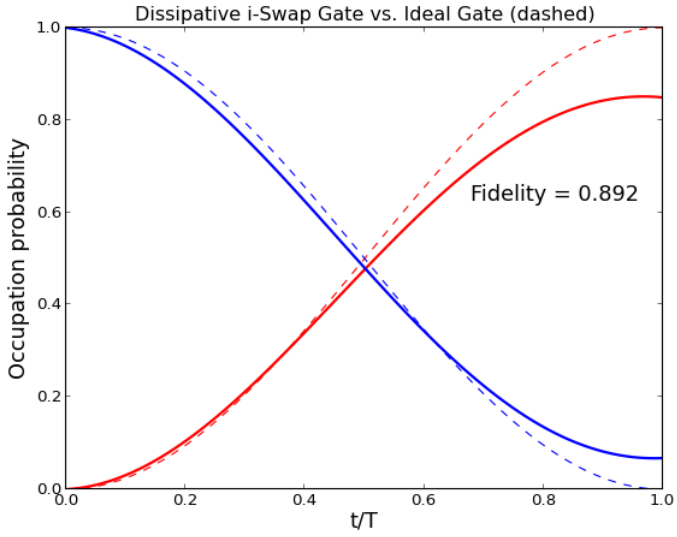
Two-qubit i-SWAP Gate

Dissipative i-Swap Gate vs. ideal gate. The accuracy of dissipative gate given by the fidelity.

This example demonstrates how to create a composite system of two qubits, define dissipation processes for each qubit, and solve for the dynamics of the system using the standard Lindblad master equation. It also shows how to obtain expectation values of select operators directly from the time-evolution solver.

```
#
# Dissipative $i$-SWAP gate vs ideal gate. Accuracy of gate given by Fidelity
# of final state and ideal final state.
#
from qutip import *
from pylab import *


def run():
    # setup system parameters
    g = 1.0 * 2 * pi     # coupling strength
    g1 = 0.75            # relaxation rate
    g2 = 0.05            # dephasing rate
    n_th = 0.75          # bath temperature
    T = pi / (4 * g)

    # construct Hamiltonian
    H = g * (tensor(sigmax(), sigmax()) + tensor(sigmay(), sigmay()))
    # construct inital state
    psi0 = tensor(basis(2, 1), basis(2, 0))
```

```
    # construct collapse operators
    c_op_list = []
    ## qubit 1 collapse operators
    sm1 = tensor(sigmam(), qeye(2))
    sz1 = tensor(sigmaz(), qeye(2))
    c_op_list.append(sqrt(g1 * (1 + n_th)) * sm1)
    c_op_list.append(sqrt(g1 * n_th) * sm1.dag())
    c_op_list.append(sqrt(g2) * sz1)
    ## qubit 2 collapse operators
    sm2 = tensor(qeye(2), sigmam())
    sz2 = tensor(qeye(2), sigmaz())
    c_op_list.append(sqrt(g1 * (1 + n_th)) * sm2)
    c_op_list.append(sqrt(g1 * n_th) * sm2.dag())
    c_op_list.append(sqrt(g2) * sz2)

    # evolve the dissipative system
    tlist = linspace(0, T, 100)
    medata = mesolve(H, psi0, tlist, c_op_list, [])
    # extract density matrices from Odedata object
    rho_list = medata.states
    # get final density matrix for fidelity comparison
    rho_final = rho_list[-1]

    # calculate expectation values
    n1 = expect(sm1.dag() * sm1, rho_list)
    n2 = expect(sm2.dag() * sm2, rho_list)

    # calculate the ideal evolution
    medata_ideal = mesolve(H, psi0, tlist, [], [])
    # extract states from Odedata object
    psi_list = medata_ideal.states

    # calculate expectation values
    n1_ideal = expect(sm1.dag() * sm1, psi_list)
    n2_ideal = expect(sm2.dag() * sm2, psi_list)
    # get last ket vector for comparision
    psi_ideal = psi_list[-1]
    # output is ket since no collapse operators.
    rho_ideal = ket2dm(psi_ideal)

    # calculate the fidelity of final states
    F = fidelity(rho_ideal, rho_final)

    # plot the results
    plot(tlist / T, n1, 'r', tlist / T, n2, 'b', lw=2)
    plot(tlist / T, n1_ideal, 'r--', tlist / T, n2_ideal, 'b--', lw=1)
    xlabel('t/T', fontsize=16)
    ylabel('Occupation probability', fontsize=16)
    figtext(0.65, 0.6, "Fidelity = %.3f" % F, fontsize=16)
    title("Dissipative i-Swap Gate vs. Ideal Gate (dashed)")
    ylim([0, 1])
    show()


if __name__ == '__main__':
    run()
```



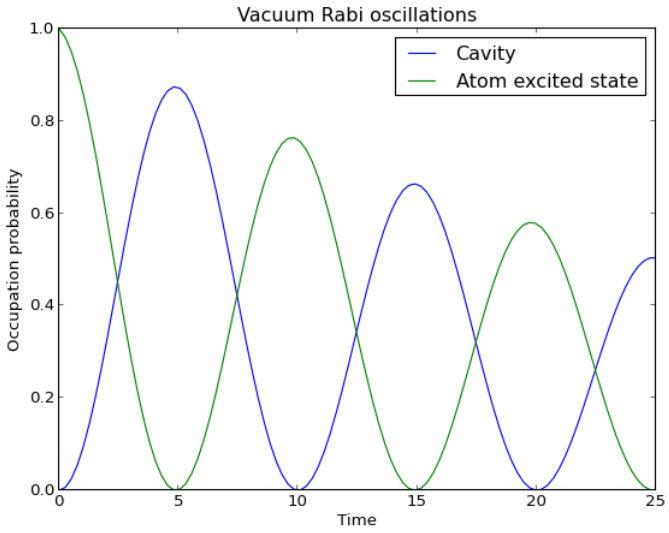
Vacuum Rabi Oscillations in the Jaynes-Cummings Model

Illustrates the vacuum Rabi oscillations in the Jaynes-Cummings model with dissipation.

This examples introduces the use of bosonic operators (`qutip.operators.destroy`, for the annihilation operator) and how to setup the Jaynes-Cummings model.

```
#
# Vacuum Rabi oscillations in the Jaynes-Cummings model with dissipation
#
from qutip import *
from pylab import *


def run():

    # Configure parameters
    wc = 1.0 * 2 * pi  # cavity frequency
    wa = 1.0 * 2 * pi  # atom frequency
    g = 0.05 * 2 * pi  # coupling strength
    kappa = 0.005       # cavity dissipation rate
    gamma = 0.05        # atom dissipation rate
    N = 5               # number of cavity fock states
    use_rwa = True

    # intial state
    psi0 = tensor(basis(N, 0), basis(2, 1))    # start with an excited atom

    # Hamiltonian
    a = tensor(destroy(N), qeye(2))
    sm = tensor(qeye(N), destroy(2))

    if use_rwa:
        # use the rotating wave approxiation
        H = wc * a.dag() * a + wa * sm.dag() * sm + \
            g * (a.dag() * sm + a * sm.dag())
    else:
        H = wc * a.dag() * a + wa * sm.dag() * sm + \
            g * (a.dag() + a) * (sm + sm.dag())

    # collapse operators
    c_op_list = []
```

```
    n_th_a = 0.0  # zero temperature
    rate = kappa * (1 + n_th_a)
    if rate > 0.0:
        c_op_list.append(sqrt(rate) * a)

    rate = kappa * n_th_a
    if rate > 0.0:
        c_op_list.append(sqrt(rate) * a.dag())

    rate = gamma
    if rate > 0.0:
        c_op_list.append(sqrt(rate) * sm)

    # evolve and calculate expectation values
    tlist = linspace(0, 25, 100)
    output = mesolve(H, psi0, tlist, c_op_list, [a.dag() * a, sm.dag() * sm])

    # plot the results
    plot(tlist, output.expect[0])
    plot(tlist, output.expect[1])
    legend(("Cavity", "Atom excited state"))
    xlabel('Time')
    ylabel('Occupation probability')
    title('Vacuum Rabi oscillations')
    show()

if __name__ == '__main__':
    run()
```



Single-atom lasing in a Jaynes-Cumming-like system

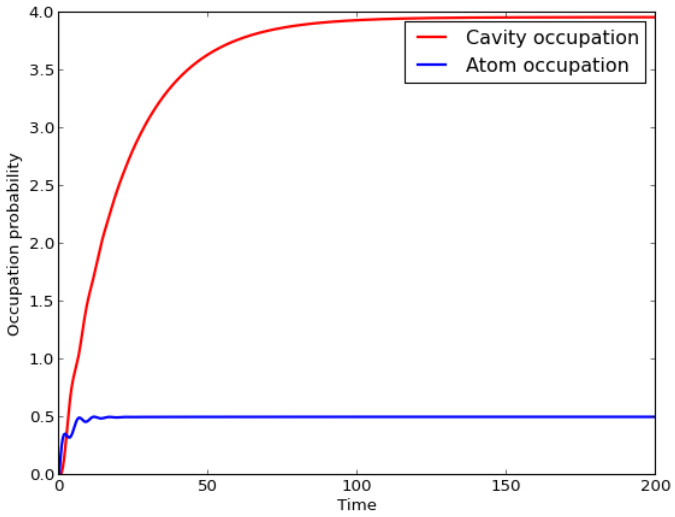
This example implements a simple model for single-atom lasing in a Jaynes-Cumming-like system. In addition to the standard Jaynes-Cumming model, there is an incoherent pumping that strive to create a population inversion in the atom.

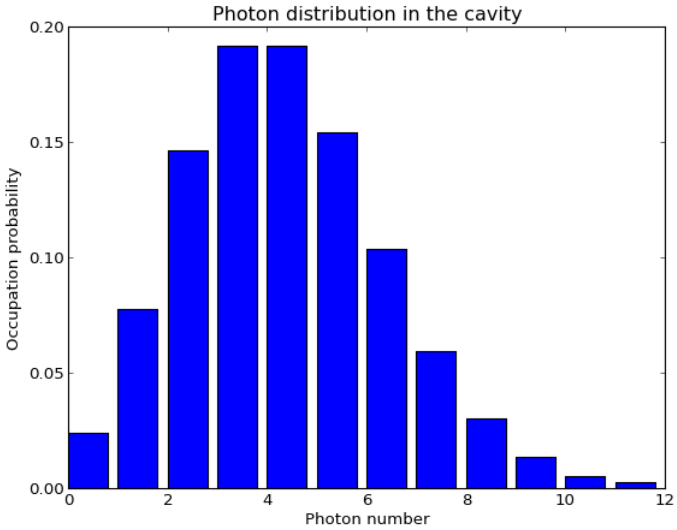
This examples demonstrates how reversed relaxation processes (note the atomic collapse operators) can be used to introduce incoherent pump processes a system.

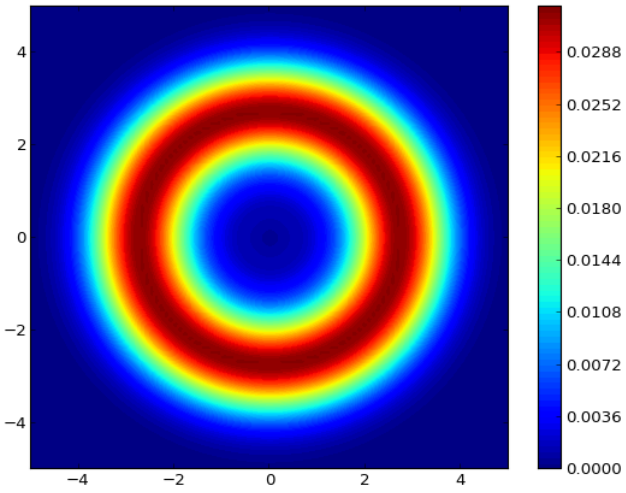
```
#
# Single-atom lasing in a Jaynes-Cumming-like system
#
```

```
from qutip import *
from pylab import *


def run():

    # Configure parameters
    N = 12              # number of cavity fock states
    wc = 2 * pi * 1.0   # cavity frequency
    wa = 2 * pi * 1.0   # atom frequency
    g = 2 * pi * 0.1    # coupling strength
    kappa = 0.05        # cavity dissipation rate
    gamma = 0.0         # atom dissipation rate
    pump = 0.4          # atom pump rate
    use_rwa = True

    # start without any excitations
    psi0 = tensor(basis(N, 0), basis(2, 0))

    # Hamiltonian
    a = tensor(destroy(N), qeye(2))
    sm = tensor(qeye(N), destroy(2))

    if use_rwa:  # use the rotating wave approxiation
        H = wc * a.dag() * a + wa * sm.dag() * sm + \
            g * (a.dag() * sm + a * sm.dag())
    else:
        H = wc * a.dag() * a + wa * sm.dag() * sm + \
            g * (a.dag() + a) * (sm + sm.dag())

    # collapse operators
    c_op_list = []

    n_th_a = 0.0  # zero temperature
    rate = kappa * (1 + n_th_a)
    if rate > 0.0:
        c_op_list.append(sqrt(rate) * a)

    rate = kappa * n_th_a
    if rate > 0.0:
        c_op_list.append(sqrt(rate) * a.dag())

    rate = gamma
    if rate > 0.0:
        c_op_list.append(sqrt(rate) * sm)

    rate = pump
    if rate > 0.0:
        c_op_list.append(sqrt(rate) * sm.dag())

    # evolve the system
    tlist = linspace(0, 200, 500)
    output = mesolve(H, psi0, tlist, c_op_list, [])

    # calculate expectation values
    nc = expect(a.dag() * a, output.states)
    na = expect(sm.dag() * sm, output.states)

    #
    # plot the time-evolution of the cavity and atom occupation
    #
    figure(1)
    plot(tlist, real(nc), 'r-', tlist, real(na), 'b-', lw=2)
```

```
    xlabel('Time')
    ylabel('Occupation probability')
    legend(("Cavity occupation", "Atom occupation"))

    #
    # plot the final photon distribution in the cavity
    #
    rho_final = output.states[-1]
    rho_cavity = ptrace(rho_final, 0)

    figure(2)
    bar(range(0, N), real(rho_cavity.diag()))
    xlabel("Photon number")
    ylabel("Occupation probability")
    title("Photon distribution in the cavity")

    #
    # plot the wigner function
    #
    xvec = linspace(-5, 5, 100)
    W = wigner(rho_cavity, xvec, xvec)
    X, Y = meshgrid(xvec, xvec)
    figure(3)
    contourf(X, Y, W, 100)
    colorbar()
    show()

if __name__ == '__main__':
    run()
```







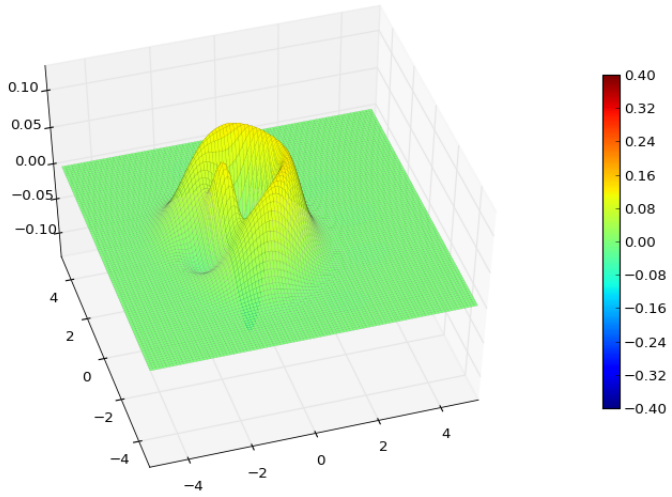
Dynamics of the Wigner distributions for the Jaynes-Cummings model

This example demonstrates the generation of Wigner distributions from the master equation evolution of the Jaynes-Cummings model. In particular, it introduce the use of the `qutip.wigner.wigner` to calculate the Wigner distribution from a density matrix for a bosonic mode, and the use of `qutip.Qobj.ptrace` to trace out the qubit from the total density matrix (to obtain the density matrix for the cavity alone).

```
#
# Dynamics of the Wigner distributions for the Jaynes-Cummings model
#
from qutip import *
from pylab import *
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm


def run():

    # Configure parameters
    N = 10          # number of cavity fock states
    wc = 2 * pi * 1.0   # cavity frequency
```

```
wa = 2 * pi * 1.0   # atom frequency
g = 2 * pi * 0.1   # coupling strength
kappa = 0.05    # cavity dissipation rate
gamma = 0.15    # atom dissipation rate
use_rwa = True

# a coherent initial state the in cavity
psi0 = tensor(coherent(N, 1.5), basis(2, 0))

# Hamiltonian
a = tensor(destroy(N), qeye(2))
sm = tensor(qeye(N), destroy(2))

if use_rwa:
    # use the rotating wave approxiation
    H = wc * a.dag() * a + wa * sm.dag() * sm + \
        g * (a.dag() * sm + a * sm.dag())
else:
    H = wc * a.dag() * a + wa * sm.dag() * sm + \
        g * (a.dag() + a) * (sm + sm.dag())

# collapse operators
c_op_list = []

n_th_a = 0.0  # zero temperature
rate = kappa * (1 + n_th_a)
if rate > 0.0:
    c_op_list.append(sqrt(rate) * a)

rate = kappa * n_th_a
if rate > 0.0:
    c_op_list.append(sqrt(rate) * a.dag())

rate = gamma
if rate > 0.0:
    c_op_list.append(sqrt(rate) * sm)

# evolve the system
tlist = linspace(0, 10, 100)
output = mesolve(H, psi0, tlist, c_op_list, [])

# calculate the wigner function
xvec = linspace(-5., 5., 100)
X, Y = meshgrid(xvec, xvec)

# for idx, rho in enumerate(output.states): # suggestion: try to loop over
# all rho
for idx, rho in enumerate([output.states[44]]):  # a selected time t=4.4

    rho_cavity = ptrace(rho, 0)
    W = wigner(rho_cavity, xvec, xvec)

    # plot the wigner function
    fig = figure(figsize=(9, 6))
    ax = Axes3D(fig, azim=-107, elev=49)
    ax.set_xlim3d(-5, 5)
    ax.set_ylim3d(-5, 5)
    ax.set_zlim3d(-0.30, 0.30)
    surf = ax.plot_surface(X, Y, W, rstride=1, cstride=1, cmap=cm.jet,
                           alpha=1.0, linewidth=0.05, vmax=0.4, vmin=-0.4)
    fig.colorbar(surf, shrink=0.65, aspect=20)
    # savefig("jc_model_wigner_"+str(idx)+".png")
```

```
    show()

if __name__ == '__main__':
    run()
```



The Dynamics of a Heisenberg Spin-1/2 Chain

This example demonstrates how to calculate the dynamics of a spin-1/2 Heisenberg chain, i.e., a sequence of two-level system (spin up or spin down) that are coupled to its nearest neighbors. The Hamiltonian for this system is

$$H = -\frac{1}{2}\sum_{n}^{N} h_n \sigma_z^n - \frac{1}{2}\sum_{n}^{N-1}\left[J_x^n \sigma_x^n \sigma_x^{n+1} + J_y^n \sigma_y^n \sigma_y^{n+1} + J_z^n \sigma_z^n \sigma_z^{n+1}\right], \tag{5.1}$$

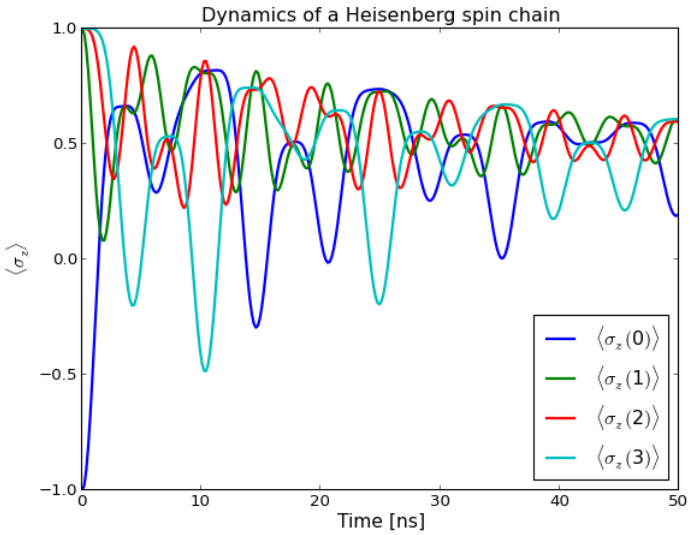
and the initial state used in the example is $|\psi_0\rangle = |\uparrow\downarrow\cdots\downarrow\rangle$.

This example is slightly more complicated than the previous in that it dynamically builds a composite Hamiltonian and initial state for a configurable number of two-level system (parameter `N=4` in the program).

```
#
# Dynamics of a Heisenberg spin 1/2 chain
#
from qutip import *
from pylab import *


def integrate(N, h, Jx, Jy, Jz, psi0, tlist, gamma, solver):

    #
    # Hamiltonian
    #
    # H = - 0.5 sum_n^N h_n sigma_z(n)
    #     - 0.5 sum_n^(N-1) [ Jx_n sigma_x(n) sigma_x(n+1) +
    #                         Jy_n sigma_y(n) sigma_y(n+1) +
    #                         Jz_n sigma_z(n) sigma_z(n+1)]
    #
    si = qeye(2)
    sx = sigmax()
    sy = sigmay()
    sz = sigmaz()

    sx_list = []
    sy_list = []
```

```
    sz_list = []
    for n in range(N):
        op_list = [si] * N
        op_list[n] = sx
        sx_list.append(tensor(op_list))
        op_list[n] = sy
        sy_list.append(tensor(op_list))
        op_list[n] = sz
        sz_list.append(tensor(op_list))

    # construct the hamiltonian
    H = 0

    # energy splitting terms
    for n in range(N):
        H += - 0.5 * h[n] * sz_list[n]

    # interaction terms
    for n in range(N - 1):
        H += - 0.5 * Jx[n] * sx_list[n] * sx_list[n + 1]
        H += - 0.5 * Jy[n] * sy_list[n] * sy_list[n + 1]
        H += - 0.5 * Jz[n] * sz_list[n] * sz_list[n + 1]

    # collapse operators
    c_op_list = []

    # spin dephasing
    for n in range(N):
        if gamma[n] > 0.0:
            c_op_list.append(sqrt(gamma[n]) * sz_list[n])

    # evolve and calculate expectation values
    if solver == "me":
        output = mesolve(H, psi0, tlist, c_op_list, sz_list)
    elif solver == "mc":
        output = mcsolve(H, psi0, tlist, c_op_list, sz_list)

    return output.expect


def run():

    solver = "me"   # select solver "me" or "mc"
    N = 4           # number of spins

    # array of spin energy splittings and coupling strengths. here we use
    # uniform parameters, but in general we don't have too
    h = 1.0 * 2 * pi * ones(N)
    Jz = 0.1 * 2 * pi * ones(N)
    Jx = 0.1 * 2 * pi * ones(N)
    Jy = 0.1 * 2 * pi * ones(N)

    # dephasing rate
    gamma = 0.01 * ones(N)

    # intial state, first spin in state |1>, the rest in state |0>
    psi_list = []
    psi_list.append(basis(2, 1))
    for n in range(N - 1):
        psi_list.append(basis(2, 0))
    psi0 = tensor(psi_list)

    tlist = linspace(0, 50, 300)
```

```
    sz_expt = integrate(N, h, Jx, Jy, Jz, psi0, tlist, gamma, solver)

    rc('font', family='Bitstream Vera Sans')
    for n in range(N):
        plot(tlist, real(sz_expt[n]), label=r'$\langle\sigma_z($' +
             str(n) + r'$)\rangle$', lw=2)
    xlabel(r'Time [ns]', fontsize=14)
    ylabel(r'$\langle\sigma_{z}\rangle$', fontsize=14)
    title(r'Dynamics of a Heisenberg spin chain')
    legend(loc="lower right")

    show()

if __name__ == "__main__":
    run()
```



Steady state calculation for a sideband-cooled nanomechanical resonator

In this example we look at how to calculate the steady state for a master equation. To demonstrate a steady state calculation we look at an example from nanomechanics: Consider a low-frequency nanomechanical resonator (with frequency ω_m lower than temperature) coupled to a high-frequency (optical) resonantor (with frequency ω_r higher than temperature). In the steady state the mechanical resonator is highly excited and the optical resonator is near its quantum ground state.

By applying a driving field to the high-frequency resonator with a frequency matching the frequency difference between the resonator, the two oscilators can effectively be brought into resonance in a rotating frame, allowing for excitation transfer from the low-frequency mechanical resonator to the high-frequency resonator (i.e., sideband cooling of the mechanical resonator).

The Hamiltonian considered for this problem is

$$H = \omega_r a^\dagger a + \omega_m b^\dagger b + g a^\dagger a (b + b^\dagger) + A \sin(\omega_d t)(a + a^\dagger),$$

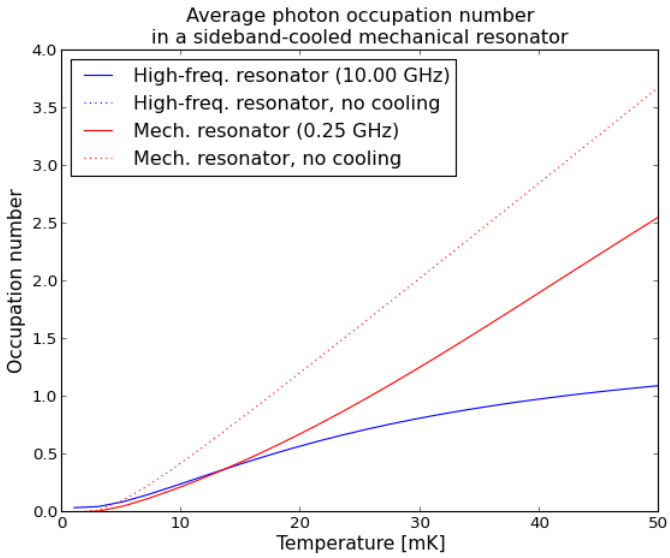
which in the rotating frame that eliminates the time-dependence of the driving term becomes

$$H = (\omega_r - \omega_d) a^\dagger a + \omega_m b^\dagger b + g a^\dagger a (b + b^\dagger) + \frac{1}{2} A (a + a^\dagger).$$

In the following code we look at the state state of this system as a function of the ambient temperature T.

```
#
# Steady state and photon occupation number for a sideband-cooled
# nanomechanical resonator, as a function of the ambient temperature.
#
from qutip import *
from pylab import *

# constants
hbar = 6.626e-34 / (2 * pi)
kB = 1.38e-23

#
# calculate the steadystate average photon count in the two resonators as a
# function of Temperature, with and without sideband cooling driving
#


def compute(T_vec, N_r, N_m, w_r, w_m, g, w_d, A, kappa_r, kappa_m):

    # pre-calculate operators
    a = tensor(destroy(N_r), qeye(N_m))  # for high-freq. mode
    b = tensor(qeye(N_r), destroy(N_m))  # for mechanical mode

    # Hamiltonian with driving, in the corresponding RWA
    H = (w_r - w_d) * a.dag() * a + w_m * b.dag() * b + \
        g * (a.dag() * a) * (b + b.dag()) + 0.5 * A * (a + a.dag())

    photon_count = zeros((len(T_vec), 4))

    for idx, T in enumerate(T_vec):

        # tempeature in frequency units [2*pi GHz]
        w_th = kB * (T * 1e-3) / hbar * 1e-9

        # collapse operators
        c_ops = []

        # collapse operators for high-frequency resonator
        n_r_th = 1.0 / (exp(w_r / w_th) - 1.0)
        rate = kappa_r * (1 + n_r_th)
        if rate > 0.0:
            c_ops.append(sqrt(rate) * a)        # relaxation
        rate = kappa_r * n_r_th
        if rate > 0.0:
            c_ops.append(sqrt(rate) * a.dag())  # thermal excitation

        # collapse operators for mechanical mode
        n_m_th = 1.0 / (exp(w_m / w_th) - 1.0)
        rate = kappa_m * (1 + n_m_th)
        if rate > 0.0:
            c_ops.append(sqrt(rate) * b)        # relaxation
        rate = kappa_m * n_m_th
        if rate > 0.0:
            c_ops.append(sqrt(rate) * b.dag())  # thermal excitation

        # find the steady state
        rho_ss = steadystate(H, c_ops)

        # calculate the photon numbers
        photon_count[idx, 0] = expect(rho_ss, a.dag() * a)
        photon_count[idx, 1] = n_r_th
        photon_count[idx, 2] = expect(rho_ss, b.dag() * b)
        photon_count[idx, 3] = n_m_th
```

```
    return photon_count


def run():

    N_r = 5               # number of fock states in high-frequency resonator
    N_m = 20              # number of fock states in mechanical resonator

    w_r = 2 * pi * 10.0   # high-freq. resonator frequency [2*pi GHz]
    w_m = 2 * pi * 0.25   # mechanical resonator frequency [2*pi GHz]

    g = 2 * pi * 0.01     # coupling strength

    w_d = w_r - w_m       # driving frequency at resonance
    A = 2 * pi * 0.10     # driving amplitude in frequency units

    # A = w_d = 0         # no cooling

    kappa_r = 0.001       # dissipation rate for high-frequency resoantor
    kappa_m = 0.001       # dissipation rate for mechanical resonator

    T_vec = linspace(1, 50.0, 25.0)  # Temperature [mK]

    # find the steady state occupation numbers
    photon_count = compute(T_vec, N_r, N_m, w_r, w_m, g,
                           w_d, A, kappa_r, kappa_m)

    # plot the results
    figure()

    plot(T_vec, photon_count[:, 0], 'b')
    plot(T_vec, photon_count[:, 1], 'b:')
    plot(T_vec, photon_count[:, 2], 'r')
    plot(T_vec, photon_count[:, 3], 'r:')

    xlabel(r'Temperature [mK]', fontsize=14)
    ylabel(r'Occupation number', fontsize=14)
    title("Average photon occupation number\n" +
          "in a sideband-cooled mechanical resonator")

    legend(("High-freq. resonator (%.2f GHz)" % (w_r / (2 * pi)),
            "High-freq. resonator, no cooling",
            "Mech. resonator (%.2f GHz)" % (w_m / (2 * pi)),
            "Mech. resonator, no cooling"), loc=2)
    show()

if __name__ == '__main__':
    run()
```

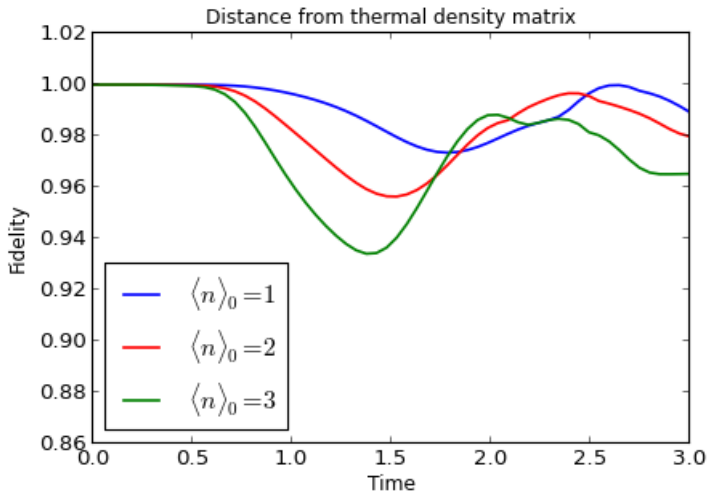


Measuring the distance between density matrices via the fidelity

Here we measure the distance of a single mode (mode #1) of a trilinear Hamiltonian from that of a thermal density matrix characterized by the expectation value of the number of excitations in the mode at time t. Here the pump mode (mode #0) is assumed to be in a initial coherent state with the given excitation number.

```
#
# Measuring the distance between density matrices via the fidelity
#
from qutip import *
from pylab import *


def run():
    fids = zeros((3, 60))  # initialize data matrix
    hilbert = [4, 5, 6]  # list of Hilbert space sizes
    num_sizes = [1, 2, 3]  # list of <n>'s for initial state of pump mode #0

    # loop over lists
    for j in range(3):
        # number of states for each mode
        N0 = hilbert[j]
        N1 = hilbert[j]
        N2 = hilbert[j]

        # define operators
        a0 = tensor(destroy(N0), qeye(N1), qeye(N2))
        a1 = tensor(qeye(N0), destroy(N1), qeye(N2))
        a2 = tensor(qeye(N0), qeye(N1), destroy(N2))

        # number operators for each mode
        num0 = a0.dag() * a0
        num1 = a1.dag() * a1
        num2 = a2.dag() * a2

        # initial state: coherent mode 0 & vacuum for modes #1 & #2
        alpha = sqrt(num_sizes[j])  # initial coherent state param for mode 0
        psi0 = tensor(coherent(N0, alpha), basis(N1, 0), basis(N2, 0))

        # trilinear Hamiltonian
        H = 1.0j * (a0 * a1.dag() * a2.dag() - a0.dag() * a1 * a2)
```

```
        # run odesolver
        tlist = linspace(0, 3, 60)
        output = mesolve(H, psi0, tlist, [], [])

        mode1 = [ptrace(k, 1) for k in output.states]     # extract mode #1
        num1 = [expect(num1, k) for k in output.states]   # get <n> for mode #1
        thermal = [thermal_dm(N1, k) for k in num1]       # calculate thermal
                                                          # matrix for <n>
        fids[j, :] = [fidelity(mode1[k], thermal[k])
                      for k in range(len(tlist))]         # calc. fidelity

    # plot the results
    fig = figure(figsize=[6, 4])
    plot(tlist, fids[0], 'b', tlist, fids[1], 'r', tlist, fids[2], 'g', lw=1.5)
    ylim([.86, 1.02])
    xlabel('Time', fontsize=11)
    ylabel('Fidelity', fontsize=11)
    title('Distance from thermal density matrix', fontsize=11)
    legend(('$\langle n\\rangle_{0}=1$', '$\langle n\\rangle_{0}=2$',
           '$\langle n\\rangle_{0}=3$'), loc=3)
    show()

if __name__ == "__main__":
    run()
```

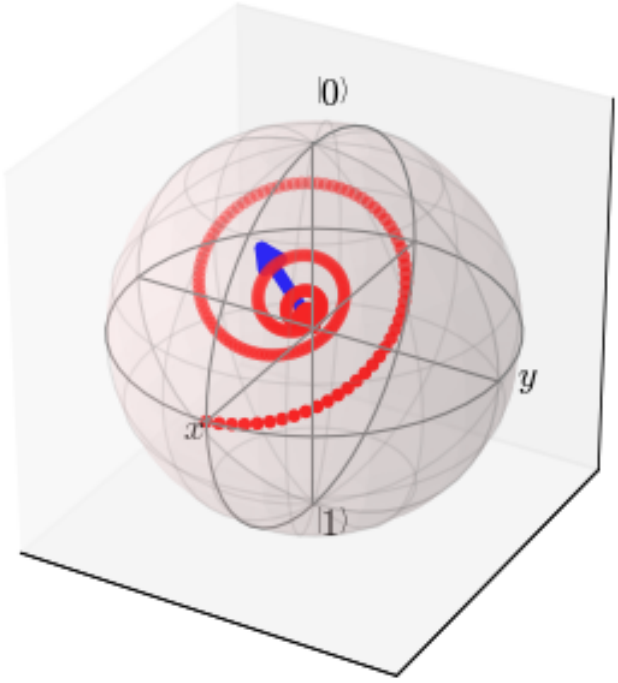


Decay of a qubit state coupled to a zero-temp. bath shown on a Bloch sphere

This example demonstrates how to visualize the dynamics of a two-level system on the Bloch sphere.

```
#
# Qubit dynamics shown in a Bloch sphere.
#
from qutip import *
from pylab import *


def qubit_integrate(w, theta, gamma1, gamma2, psi0, tlist):
    # Hamiltonian
    sx = sigmax()
    sy = sigmay()
    sz = sigmaz()
    sm = sigmam()
    H = w * (cos(theta) * sz + sin(theta) * sx)
```

```
    # collapse operators
    c_op_list = []
    n_th = 0  # zero temperature
    rate = gamma1 * (n_th + 1)
    if rate > 0.0:
        c_op_list.append(sqrt(rate) * sm)
    rate = gamma1 * n_th
    if rate > 0.0:
        c_op_list.append(sqrt(rate) * sm.dag())
    rate = gamma2
    if rate > 0.0:
        c_op_list.append(sqrt(rate) * sz)
    # evolve and calculate expectation values
    output = mesolve(H, psi0, tlist, c_op_list, [sx, sy, sz])
    return output.expect


def run():

    w = 1.0 * 2 * pi  # qubit angular frequency
    theta = 0.2 * pi  # qubit angle from sigma_z axis (toward sigma_x axis)
    gamma1 = 0.05     # qubit relaxation rate
    gamma2 = 1.0      # qubit dephasing rate

    # initial state
    a = .5
    psi0 = (a * basis(2, 0) + (1 - a) * basis(2, 1)).unit()

    tlist = linspace(0, 3, 500)
    sx, sy, sz = qubit_integrate(w, theta, gamma1, gamma2, psi0, tlist)

    sphere = Bloch()
    sphere.add_points([sx, sy, sz])
    sphere.point_color = ['r']
    sphere.vector_color = ['b']
    sphere.size = [4, 4]
    sphere.font_size = 14
    sphere.add_vectors([sin(theta), 0, cos(theta)])
    sphere.show()


if __name__ == "__main__":
    run()
```



5.2.3 Monte Carlo Examples

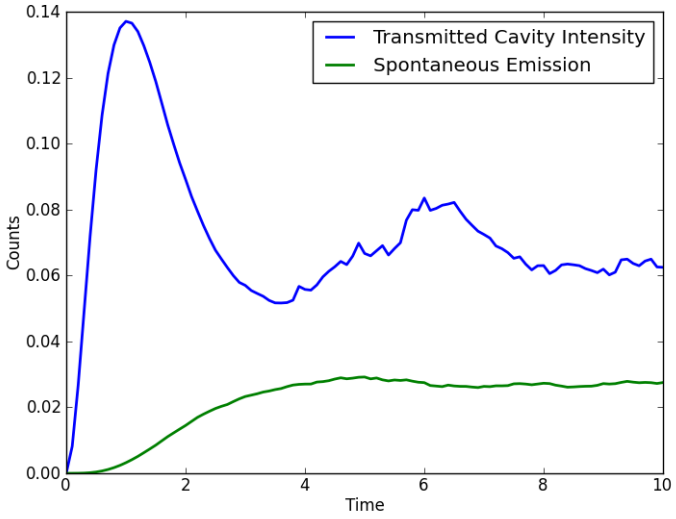
Driven Cavity+Qubit Monte Carlo

Monte Carlo evoution of a coherently driven cavity with a two-level atom initially in the ground state and no photons in the cavity.

Adapted from qotoolbox example 'probqmc3' by Sze M. Tan.

```
#load qutip and matplotlib
from qutip import *
from pylab import *

def run():
    # set system parameters
    kappa=2.0 #mirror coupling
    gamma=0.2 #spontaneous emission rate
    g=1 #atom/cavity coupling strength
    wc=0 #cavity frequency
    w0=0 #atom frequency
    wl=0 #driving frequency
    E=0.5 #driving amplitude
    N=4 #number of cavity energy levels (0->3 Fock states)
    tlist=linspace(0,10,101) #times for expectation values

    # construct Hamiltonian
    ida=qeye(N)
    idatom=qeye(2)
    a=tensor(destroy(N),idatom)
    sm=tensor(ida,sigmam())
    H=(w0-wl)*sm.dag()*sm+(wc-wl)*a.dag()*a+1j*g*(a.dag()*sm-sm.dag()*a)+E*(a.dag()+a)

    #collapse operators
    C1=sqrt(2*kappa)*a
    C2=sqrt(gamma)*sm
```

```
    C1dC1=C1.dag()*C1
    C2dC2=C2.dag()*C2

    #intial state
    psi0=tensor(basis(N,0),basis(2,1))

    #run monte-carlo solver with default 500 trajectories
    start_time=time.time()
    data=mcsolve(H,psi0,tlist,[C1,C2],[C1dC1,C2dC2])
    #plot expectation values
    plot(tlist,data.expect[0],tlist,data.expect[1],lw=2)
    legend(('Transmitted Cavity Intensity','Spontaneous Emission'))
    ylabel('Counts')
    xlabel('Time')
    show()

if __name__=='__main__':
    run()
```

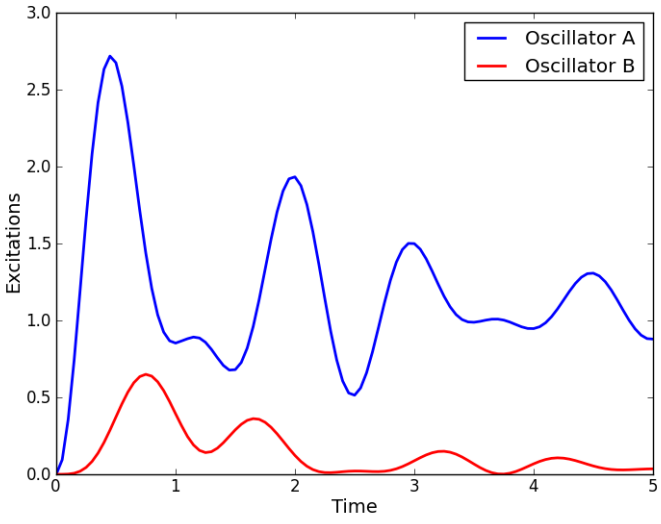


Coupled Driven Oscillators

Occupation number of two coupled osciilators with oscillator a driven by an external classical drive. Both oscillators are assumed to start in the ground state.

```
#
# Occupation number of two coupled osciilators with
# oscillator A driven by an external classical drive.
# Both oscillators are assumed to start in the ground
# state.
#
from qutip import *
from pylab import *

def run():
    wa  = 1.0 * 2 * pi    # frequency of system a
    wb  = 1.0 * 2 * pi    # frequency of system a
    wab = 0.2 * 2 * pi    # coupling frequency
    ga = 0.2 * 2 * pi     # dissipation rate of system a
    gb = 0.1 * 2 * pi     # dissipation rate of system b
    Na = 10               # number of states in system a
```

```
    Nb = 10                 # number of states in system b
    E = 1.0 * 2 * pi        # Oscillator A driving strength

    a = tensor(destroy(Na), qeye(Nb))
    b = tensor(qeye(Na), destroy(Nb))
    na = a.dag() * a
    nb = b.dag() * b
    H = wa * na + wb * nb + wab * (a.dag() * b + a * b.dag()) + E * (a.dag() + a)

    # start with both oscillators in ground state
    psi0 = tensor(basis(Na), basis(Nb))

    c_op_list = []
    c_op_list.append(sqrt(ga) * a)
    c_op_list.append(sqrt(gb) * b)

    tlist = linspace(0, 5, 101)

    #run simulation
    data = mcsolve(H, psi0, tlist, c_op_list, [na, nb])

    #plot results
    plot(tlist, data.expect[0], 'b', tlist, data.expect[1],'r', lw=2)
    xlabel('Time', fontsize=14)
    ylabel('Excitations', fontsize=14)
    legend(('Oscillator A', 'Oscillator B'))
    show()

if __name__=='__main__':
    run()
```



Averaging of Monte Carlo Trajectories to Master Equation Solution

This is a Monte Carlo simulation showing the decay of a cavity Fock state $|1\rangle$ in a thermal environment with an average occupation number of $n = 0.063$. Here, the coupling strength is given by the inverse of the cavity ring-down time $T_c = 0.129$.

The parameters chosen here correspond to those from S. Gleyzes, et al., Nature **446**, 297 (2007).

```
#load qutip and matplotlib
from qutip import *
from pylab import *

def run():
    # define parameters
    N=4              # number of basis states to consider
    kappa=1.0/0.129 # coupling to heat bath
    nth= 0.063       # temperature with <n>=0.063

    # create operators and initial |1> state
    a=destroy(N)     # cavity destruction operator
    H=a.dag()*a      # harmonic oscillator Hamiltonian
    psi0=basis(N,1) # initial Fock state with one photon

    # collapse operators
    c_op_list = []
    # decay operator
    c_op_list.append(sqrt(kappa * (1 + nth)) * a)
    # excitation operator
    c_op_list.append(sqrt(kappa * nth) * a.dag())

    # run monte carlo simulation
    ntraj=[1,5,15,904] # list of number of trajectories to avg. over
    tlist=linspace(0,0.6,100)
    mc = mcsolve(H,psi0,tlist,c_op_list,[a.dag()*a],ntraj)
    # get expectation values from mc data (need extra index since ntraj is list)
    ex1=mc.expect[0][0]      #for ntraj=1
    ex5=mc.expect[1][0]      #for ntraj=5
    ex15=mc.expect[2][0]     #for ntraj=15
    ex904=mc.expect[3][0]    #for ntraj=904

    ## run master equation to get ensemble average expectation values ##
    me = mesolve(H,psi0,tlist,c_op_list, [a.dag()*a])

    #  calulate final state using steadystate solver
    final_state=steadystate(H,c_op_list) # find steady-state
    fexpt=expect(a.dag()*a,final_state)  # find expectation value for particle number

    #
    # plot results using vertically stacked plots
    #

    # set legend fontsize
    import matplotlib.font_manager
    leg_prop = matplotlib.font_manager.FontProperties(size=10)

    f = figure(figsize=(6,9))
    subplots_adjust(hspace=0.001) #no space between plots

    # subplot 1 (top)
    ax1 = subplot(411)
    ax1.plot(tlist,ex1,'b',lw=2)
    ax1.axhline(y=fexpt,color='k',lw=1.5)
    yticks(linspace(0,2,5))
    ylim([-0.1,1.5])
    ylabel('$\left< N \\right>$',fontsize=14)
    title("Ensemble Averaging of Monte Carlo Trajectories")
    legend(('Single trajectory','steady state'),prop=leg_prop)

    # subplot 2
    ax2=subplot(412,sharex=ax1) #share x-axis of subplot 1
    ax2.plot(tlist,ex5,'b',lw=2)
```

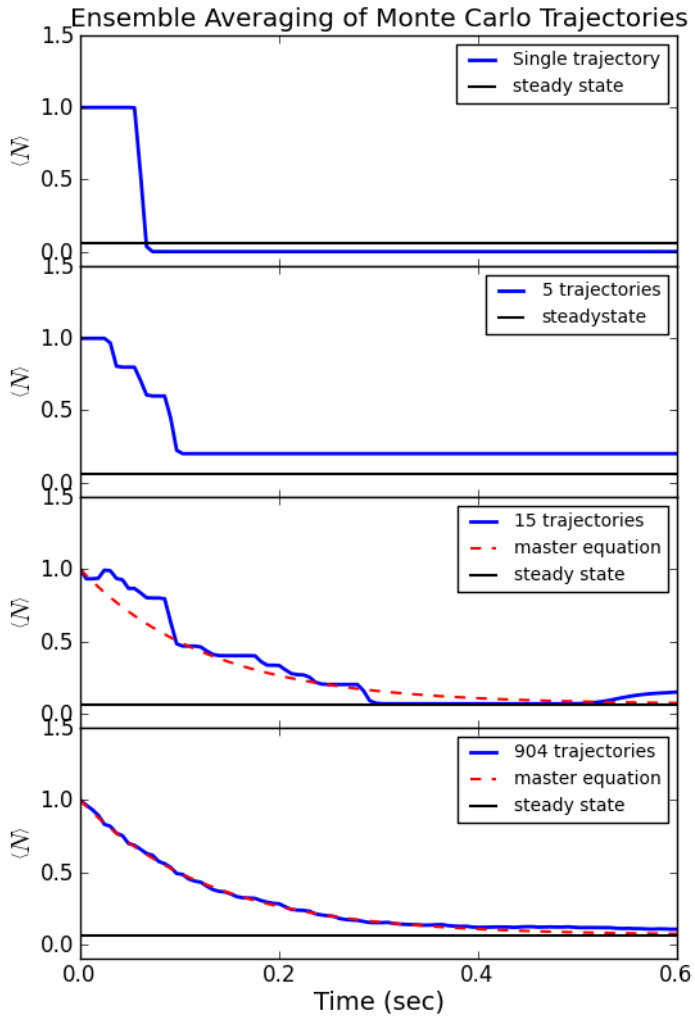
```
    ax2.axhline(y=fexpt,color='k',lw=1.5)
    yticks(linspace(0,2,5))
    ylim([-0.1,1.5])
    ylabel('$\left< N \\right>$',fontsize=14)
    legend(('5 trajectories','steadystate'),prop=leg_prop)

    # subplot 3
    ax3=subplot(413,sharex=ax1) #share x-axis of subplot 1
    ax3.plot(tlist,ex15,'b',lw=2)
    ax3.plot(tlist,me.expect[0],'r--',lw=1.5)
    ax3.axhline(y=fexpt,color='k',lw=1.5)
    yticks(linspace(0,2,5))
    ylim([-0.1,1.5])
    ylabel('$\left< N \\right>$',fontsize=14)
    legend(('15 trajectories','master equation','steady state'),prop=leg_prop)

    # subplot 4 (bottom)
    ax4=subplot(414,sharex=ax1) #share x-axis of subplot 1
    ax4.plot(tlist,ex904,'b',lw=2)
    ax4.plot(tlist,me.expect[0],'r--',lw=1.5)
    ax4.axhline(y=fexpt,color='k',lw=1.5)
    yticks(linspace(0,2,5))
    ylim([-0.1,1.5])
    ylabel('$\left< N \\right>$',fontsize=14)
    legend(('904 trajectories','master equation','steady state'),prop=leg_prop)

    #remove x-axis tick marks from top 3 subplots
    xticklabels = ax1.get_xticklabels()+ax2.get_xticklabels()+ax3.get_xticklabels()
    setp(xticklabels, visible=False)

    ax1.xaxis.set_major_locator(MaxNLocator(4))
    xlabel('Time (sec)',fontsize=14)
    show()


if __name__=="__main__":
    run()
```

Trilinear Hamiltonian: Deviation from Thermal State

Demonstrates the deviation from a thermal distribution for a single oscillator mode of the trilinear Hamiltonian.

Adapted from Nation & Blencowe, NJP 12 095013 (2010).

```
#
# Demonstrate the deviation from a thermal distribution
# for the trilinear Hamiltonian.
#
# Adapted from Nation & Blencowe, NJP 12 095013 (2010)
#
from qutip import *
from pylab import *

def run():
    #number of states for each mode
    N0=15
    N1=15
    N2=15

    #define operators
    a0=tensor(destroy(N0),qeye(N1),qeye(N2))
    a1=tensor(qeye(N0),destroy(N1),qeye(N2))
```

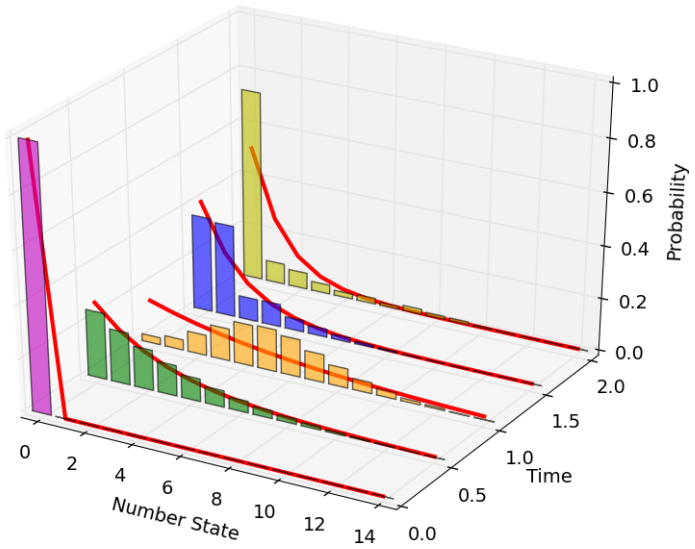

Ensemble Averaging of Monte Carlo Trajectories
Single trajectory
steady state
5 trajectories
steadystate
15 trajectories
master equation
steady state
904 trajectories
master equation
steady state
⟨N⟩
Time (sec)
0.0
0.2
0.4
0.6
0.5
1.0
1.5

```
    a2=tensor(qeye(N0),qeye(N1),destroy(N2))

    #number operators for each mode
    num0=a0.dag()*a0
    num1=a1.dag()*a1
    num2=a2.dag()*a2

    #initial state: coherent mode 0 & vacuum for modes #1 & #2
    alpha=sqrt(7)#initial coherent state param for mode 0
    psi0=tensor(coherent(N0,alpha),basis(N1,0),basis(N2,0))

    #trilinear Hamiltonian
    H=1.0j*(a0*a1.dag()*a2.dag()-a0.dag()*a1*a2)

    #run Monte-Carlo
    tlist=linspace(0,2.5,50)
    output=mcsolve(H,psi0,tlist,[],[],ntraj=1)

    #extrace mode 1 using ptrace
    mode1=[psi.ptrace(1) for psi in output.states]
    #get diagonal elements
    diags1=[k.diag() for k in mode1]
    #calculate num of particles in mode 1
    num1=[expect(num1,k) for k in output.states]
    #generate thermal state with same # of particles
    thermal=[thermal_dm(N1,k).diag() for k in num1]

    #plot results
    from mpl_toolkits.mplot3d import Axes3D
    from matplotlib import cm
    colors=['m', 'g','orange','b', 'y','pink']
    x=arange(N1)
    params = {'axes.labelsize': 14,'text.fontsize': 14,'legend.fontsize': 12,'xtick.labelsize': 1
    rcParams.update(params)
    fig = plt.figure()
    ax = Axes3D(fig)
    for j in range(5):
        ax.bar(x, diags1[10*j], zs=tlist[10*j], zdir='y',color=colors[j],linewidth=1.0,alpha=0.6,
        ax.plot(x,thermal[10*j],zs=tlist[10*j],zdir='y',color='r',linewidth=3,alpha=1)
    ax.set_zlabel(r'Probability')
    ax.set_xlabel(r'Number State')
    ax.set_ylabel(r'Time')
    ax.set_zlim3d(0,1)
    show()

if __name__=='__main__':
    run()
```

Visualizing Monte Carlo Collapse Times and Operators

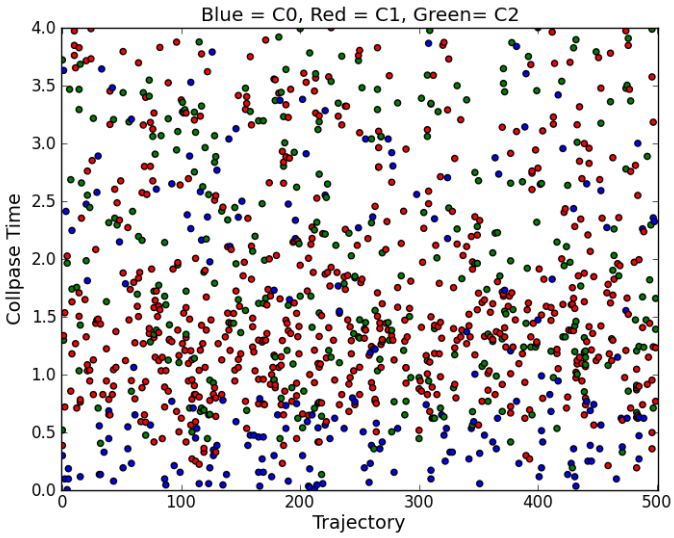
Example showing which times and operators were responsible for wave function collapse in a Monte Carlo simulation of a dissipative trilinear Hamiltonian. Operators are color coded for clarity.

```
#
# Example showing which times and operators
# were responsible for wave function collapse
# in the monte-carlo simulation of a dissipative
# trilinear Hamiltonian.
#

from qutip import *
from pylab import *
```




```
import matplotlib.pyplot as plt

def run():
    #number of states for each mode
    N0=6
    N1=6
    N2=6
    #damping rates
    gamma0=0.1
    gamma1=0.4
    gamma2=0.1
    alpha=sqrt(2)#initial coherent state param for mode 0
    tlist=linspace(0,4,200)
    ntraj=500#number of trajectories

    #define operators
    a0=tensor(destroy(N0),qeye(N1),qeye(N2))
    a1=tensor(qeye(N0),destroy(N1),qeye(N2))
    a2=tensor(qeye(N0),qeye(N1),destroy(N2))

    #number operators for each mode
    num0=a0.dag()*a0
    num1=a1.dag()*a1
    num2=a2.dag()*a2

    #dissipative operators for zero-temp. baths
    C0=sqrt(2.0*gamma0)*a0
    C1=sqrt(2.0*gamma1)*a1
    C2=sqrt(2.0*gamma2)*a2

    #initial state: coherent mode 0 & vacuum for modes #1 & #2
    psi0=tensor(coherent(N0,alpha),basis(N1,0),basis(N2,0))

    #trilinear Hamiltonian
    H=1j*(a0*a1.dag()*a2.dag()-a0.dag()*a1*a2)

    #run Monte-Carlo
    data=mcsolve(H,psi0,tlist,[C0,C1,C2],[num0,num1,num2])

    #plot results
    fig = plt.figure()
```

```
    ax = fig.add_subplot(111)
    cs=['b','r','g'] #set three colors, one for each operator
    for k in xrange(ntraj):
        if len(data.col_times[k])>0:#just in case no collapse
            colors=[cs[j] for j in data.col_which[k]]#set color
            xdat=[k for x in xrange(len(data.col_times[k]))]
            ax.scatter(xdat,data.col_times[k],marker='o',c=colors)
    ax.set_xlim([-1,ntraj+1])
    ax.set_ylim([0,tlist[-1]])
    ax.set_xlabel('Trajectory',fontsize=14)
    ax.set_ylabel('Collpase Time',fontsize=14)
    ax.set_title('Blue = C0, Red = C1, Green= C2')
    show()

if __name__=='__main__':
    run()
```



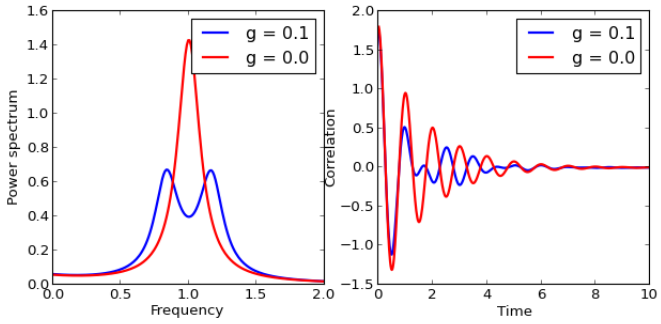
5.2.4 Correlation + Spectrum Examples

Spectrum of cavity coupled to an atom

Calculate the correlation and power spectrum of a cavity, with and without coupling to a two-level atom.

```
#
# Calculate the correlation and power spectrum of a cavity,
# with and without coupling to a two-level atom.
#
from qutip import *
from pylab import *

def calc_spectrum(N, wc, wa, g, kappa, gamma, tlist, wlist):

    # Hamiltonian
    a = tensor(destroy(N), qeye(2))
    sm = tensor(qeye(N), destroy(2))
    H = wc * a.dag() * a + wa * sm.dag() * sm + \
        g * (a.dag() * sm + a * sm.dag())

    # collapse operators
    c_op_list = []
```

```
    n_th_a = 0.5
    rate = kappa * (1 + n_th_a)
    if rate > 0.0:
        c_op_list.append(sqrt(rate) * a)

    rate = kappa * n_th_a
    if rate > 0.0:
        c_op_list.append(sqrt(rate) * a.dag())

    rate = gamma
    if rate > 0.0:
        c_op_list.append(sqrt(rate) * sm)

    A = a.dag() + a
    B = A

    # calculate the power spectrum
    corr = correlation_ss(H, tlist, c_op_list, A, B)

    # calculate the power spectrum
    spec = spectrum_ss(H, wlist, c_op_list, A, B)

    return corr, spec

def run():
    #
    # setup the calcualtion
    #
    N = 4               # number of cavity fock states
    wc = 1.00 * 2 * pi  # cavity frequency
    wa = 1.00 * 2 * pi  # atom frequency
    g = 0.20 * 2 * pi  # coupling strength
    kappa = 1.0         # cavity dissipation rate
    gamma = 0.2         # atom dissipation rate

    wlist = linspace(0, 2 * pi * 2, 200)
    tlist = linspace(0, 10, 500)

    corr1, spec1 = calc_spectrum(N, wc, wa, g, kappa, gamma, tlist, wlist)
    corr2, spec2 = calc_spectrum(N, wc, wa, 0, kappa, gamma, tlist, wlist)

    # plot results side-by-side
    figure(figsize=(9, 4))
    subplot(1, 2, 1)
    plot(wlist / (2 * pi), abs(spec1), 'b', lw=2)
    plot(wlist / (2 * pi), abs(spec2), 'r', lw=2)
    xlabel('Frequency')
    ylabel('Power spectrum')
    legend(("g = 0.1", "g = 0.0"))
    subplot(1, 2, 2)
    plot(tlist, real(corr1), 'b', tlist, real(corr2), 'r', lw=2)
    xlabel('Time')
    ylabel('Correlation')
    legend(("g = 0.1", "g = 0.0"))
    show()


if __name__ == '__main__':
    run()
```



5.2.5 Dynamics of time-dependent systems

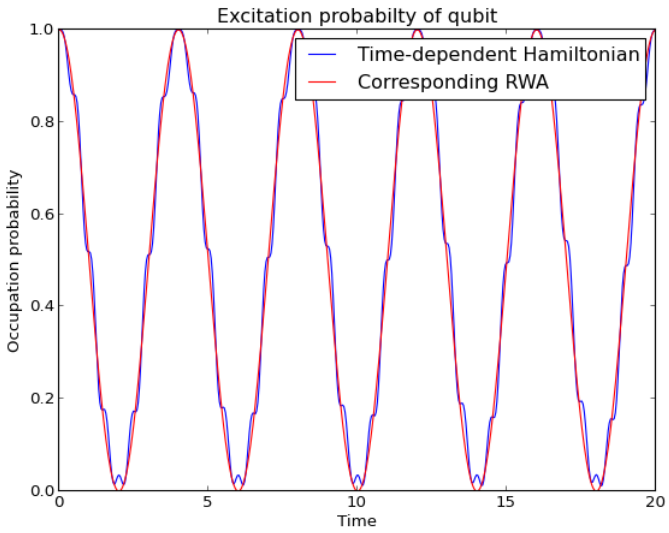
Rabi oscillations of an atom subject to a time-dependent classical driving field

The state of an atom subject to a classical driving field oscillates between its ground and excited states, a phenomena known as Rabi oscillations. This example gives a numerical demonstration of this effect by solving for the dynamics of the time-dependent two-level Hamiltonian. This example also demonstrates how the *list-string format* is used to define a time-dependent Hamiltonian.

The time-depdendent hamiltonian of the atom with a classical coherent driving becomes time-independent in the rotating-frame approximation (RWA). The results from evolving the system using the RWA Hamiltonian is also plotted.

```
#
# Rabi oscillations of qubit subject to a classical driving field.
#
from qutip import *
from pylab import *


def run():

    #
    # problem parameters:
    #
    delta = 0.0 * 2 * pi  # qubit sigma_x coefficient
    eps0 = 1.0 * 2 * pi   # qubit sigma_z coefficient
    A = 0.25 * 2 * pi     # drive amplitude (reducing -> RWA more accurate)
    w = 1.0 * 2 * pi      # drive frequency
    gamma1 = 0.0          # relaxation rate
    n_th = 0.0            # average number of excitations ("temperature")
    psi0 = basis(2, 1)    # initial state

    #
    # Hamiltonian
    #
    sx = sigmax()
    sz = sigmaz()
    sm = destroy(2)

    H0 = - delta / 2.0 * sx - eps0 / 2.0 * sz
    H1 = - A * sx

    # define the time-dependence of the hamiltonian using the list-string
    # format
    args = {'w': w}
    Ht = [H0, [H1, 'sin(w*t)']]

    #
```

```
    # collapse operators
    #
    c_op_list = []

    rate = gamma1 * (1 + n_th)
    if rate > 0.0:
        c_op_list.append(sqrt(rate) * sm)       # relaxation

    rate = gamma1 * n_th
    if rate > 0.0:
        c_op_list.append(sqrt(rate) * sm.dag())  # excitation

    #
    # evolve and system subject to the time-dependent hamiltonian
    #
    tlist = linspace(0, 5.0 * 2 * pi / A, 500)
    output1 = mesolve(Ht, psi0, tlist, c_op_list, [sm.dag() * sm], args)

    # Alternative: write the hamiltonian in a rotating frame, and neglect the
    # the high frequency component (rotating wave approximation), so that the
    # resulting Hamiltonian is time-independent.
    H_rwa = - delta / 2.0 * sx - A * sx / 2
    output2 = mesolve(H_rwa, psi0, tlist, c_op_list, [sm.dag() * sm])

    #
    # Plot the solution
    #
    plot(tlist, real(output1.expect[0]), 'b',
         tlist, real(output2.expect[0]), 'r')
    xlabel('Time')
    ylabel('Occupation probability')
    title('Excitation probabilty of qubit')
    legend(("Time-dependent Hamiltonian", "Corresponding RWA"))
    show()


if __name__ == '__main__':
    run()
```

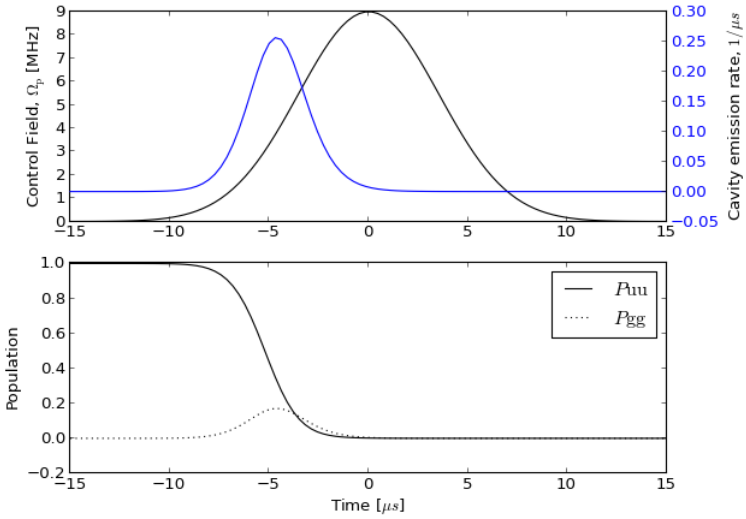


Single photon source based on a three level atom strongly coupled to a cavity

This example demonstrates a single photon source based on a three level atom strongly coupled to a cavity. Contributed by Markus Baden.

```
#
# Single photon source based on a three level atom strongly coupled to a cavity
#
# We follow the treatment presented in Kuhn et al.,
# Appl. Phys. B 69, 373-377 (1999),
# http://www.mpq.mpg.de/qdynamics/publications/library/APB69p373_Kuhn.pdf,
# for more details see M. Hennrich's thesis,
# http://mediatum2.ub.tum.de/node?id=602970.
#
# We study the following lambda system,
#
#                |e>
#             --------
#             /      \
#      Omega /        \ g
#           /          \
#          /         -------
#      -------         |g>
#       |u>
#
# where |u> and |g> are the ground states and |e> is the exicted state.
# |u> and |e> are coupled by a classical laser field with Rabi frequency
# Omega, and |g> and |e> by a cavity field with 2g being the single-photon
# Rabi frequency.
#
from qutip import *
from pylab import *


def run():

    # Define atomic states. Use ordering from paper
    ustate = basis(3, 0)
    excited = basis(3, 1)
    ground = basis(3, 2)

    # Set where to truncate Fock state for cavity
    N = 2

    # Create the atomic operators needed for the Hamiltonian
    sigma_ge = tensor(qeye(N), ground * excited.dag())  # |g><e|
    sigma_ue = tensor(qeye(N), ustate * excited.dag())  # |u><e|

    # Create the photon operator
    a = tensor(destroy(N), qeye(3))
    ada = tensor(num(N), qeye(3))

    # Define collapse operators
    c_op_list = []
    # Cavity decay rate
    kappa = 1.5
    c_op_list.append(sqrt(kappa) * a)

    # Atomic decay rate
    gamma = 6
    # Use Rb branching ratio of 5/9 e->u, 4/9 e->g
    c_op_list.append(sqrt(5 * gamma / 9) * sigma_ue)
    c_op_list.append(sqrt(4 * gamma / 9) * sigma_ge)
```

```
    # Define time vector
    t = linspace(-15, 15, 100)
    # Define pump strength as a function of time
    wp = lambda t: 9 * exp(-(t / 5) ** 2)

    # Set up the time varying Hamiltonian
    g = 5
    H0 = -g * (sigma_ge.dag() * a + a.dag() * sigma_ge)
    H1 = (sigma_ue.dag() + sigma_ue)

    def Hfunc(t, args):
        H0 = args[0]
        H1 = args[1]
        w = wp(t)
        return H0 - w * H1

    # Define initial state
    psi0 = tensor(basis(N, 0), ustate)

    # Define states onto which to project (same as in paper)
    state_GG = tensor(basis(N, 1), ground)
    sigma_GG = state_GG * state_GG.dag()
    state_UU = tensor(basis(N, 0), ustate)
    sigma_UU = state_UU * state_UU.dag()

    output = mesolve(Hfunc, psi0, t, c_op_list,
                     [ada, sigma_UU, sigma_GG], [H0, H1])

    exp_ada, exp_uu, exp_gg = (output.expect[0], output.expect[1],
                               output.expect[2])

    # Plot the results
    figure()
    subplot(211)
    plot(t, wp(t), 'k')
    ylabel('Control Field, $\Omega_\mathrm{p}$ [MHz]')
    ax = twinx()
    plot(t, kappa * exp_ada, 'b')
    ylabel('Cavity emission rate, $1/\mu s$')
    for tl in ax.get_yticklabels():
        tl.set_color('b')

    subplot(212)
    plot(t, exp_uu, 'k-', label='$P{\mathrm{uu}}$')
    plot(t, exp_gg, 'k:', label='$P{\mathrm{gg}}$')
    ylabel('Population')
    xlabel('Time [$\mu s$]')
    legend()
    show()

if __name__ == '__main__':
    run()
```



Landau-Zener transitions in a quantum two-level system

The Landau-Zener problem is a simple yet nontrivial example of a time-dependent problem in quantum mechanics. It concerns the occupation probabilities of the states of a two-level atom when its energy bias is linearly swept from negative to positive infinity, through an avoided-level crossing. The Hamiltonian for the problem is

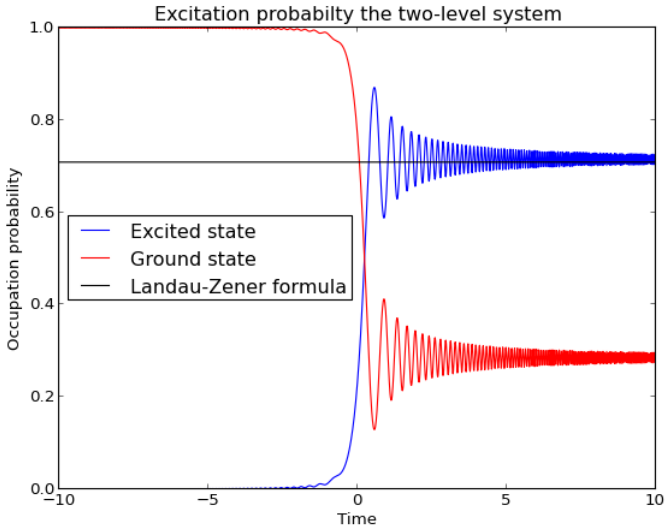
$$H(t) = \frac{1}{2}\Delta\sigma_x + \frac{1}{2}vt\sigma_z,$$

where Δ is the tunneling rate at $t = 0$, v is the sweep rate of the bare energy splitting, and time t goes from $-\infty$ to ∞. The Landau-Zener formula gives the final occupation probabilities at $t \to \infty$, e.g., for the final ground state: $P = 1 - \exp(-\pi\Delta^2/2v)$. However, there is no analytic formala for the occupation probabilities at intermediate times.

In QuTiP it is easy to calculate the time-evolution of the Landau-Zener problem numerically, which is demonstrated here. This example also shows how to use the function-callback format to define a time-dependent Hamiltonian.

```
#
# Landau-Zener transitions in a quantum two-level system
#
from qutip import *
from pylab import *


def run():

    def hamiltonian_t(t, args):
        """ evaluate the hamiltonian at time t. """
        H0 = args[0]
        H1 = args[1]
        return H0 + t * H1

    #
    # set up the parameters
    #
    delta = 0.5 * 2 * pi  # qubit sigma_x coefficient
    eps0 = 0.0 * 2 * pi   # qubit sigma_z coefficient
    A = 2.0 * 2 * pi      # sweep rate
    gamma1 = 0.0          # relaxation rate
    n_th = 0.0            # average number of thermal photons
    psi0 = basis(2, 0)    # initial state
```

```
    #
    # Hamiltonian
    #
    sx = sigmax()
    sz = sigmaz()
    sm = destroy(2)

    H0 = - delta / 2.0 * sx - eps0 / 2.0 * sz
    H1 = - A / 2.0 * sz
    args = (H0, H1)

    #
    # collapse operators, only active if gamma1 > 0
    #
    c_ops = []

    rate = gamma1 * (1 + n_th)
    if rate > 0.0:
        c_ops.append(sqrt(rate) * sm)       # relaxation

    rate = gamma1 * n_th
    if rate > 0.0:
        c_ops.append(sqrt(rate) * sm.dag())  # excitation

    #
    # evolve and calculate expectation values
    #
    tlist = linspace(-10.0, 10.0, 1500)
    output = mesolve(hamiltonian_t, psi0, tlist, c_ops, [sm.dag() * sm], args)

    #
    # Plot the results
    #
    plot(tlist, real(output.expect[0]), 'b',
         tlist, real(1 - output.expect[0]), 'r')
    plot(tlist, 1 - exp(- pi * delta ** 2 / (2 * A)) * ones(shape(tlist)), 'k')
    xlabel('Time')
    ylabel('Occupation probability')
    title('Excitation probabilty the two-level system')
    legend(("Excited state", "Ground state", "Landau-Zener formula"), loc=0)
    show()

if __name__ == '__main__':
    run()
```



Using the propagator to find the steady state of a driven system

In this example we consider a strongly driven two-level system where the driving field couples to the σ_z operator. The system is subject to repeated Landau-Zener-like transitions:

$$H(t) = -\frac{1}{2}\Delta\sigma_x - \frac{1}{2}\epsilon\sigma_z - \frac{1}{2}A\cos(\omega t)\sigma_z.$$

Here, Δ is the tunneling rate, ϵ is the energy-bias in the absense of the driving field, $A \gg \epsilon$ is the (strong) driving amplitude, and ω is the driving frequency.

In the following code we evolve the system for a few driving periods and plot the results, to get an idea of how the state of the two-level system changes at the avoided-level crossing points (where the σ_z coefficient in the Hamiltonian is zero).

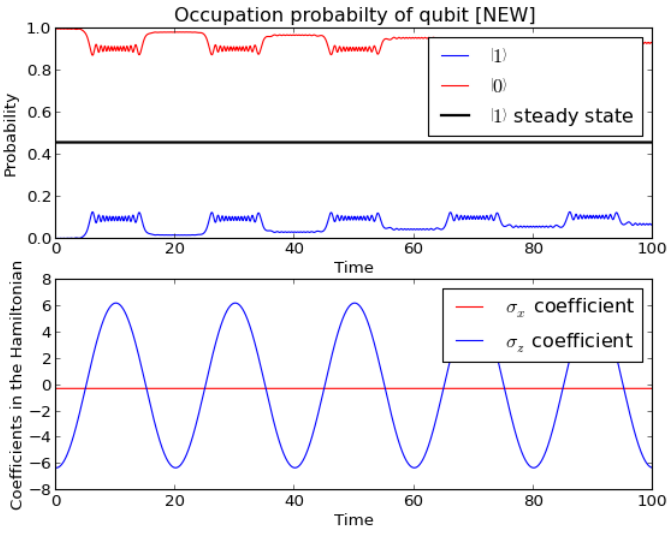
Next, we use the `qutip.propagator` function to find the propagator for the system for one driving period, and then we use the `qutip.propagator_steadystate` function to find the pseudo steady state density matrix that follows from infinitely many applications of the one-period propagotor.

This examples demonstrates how to use the list-callback format to define a time-dependent Hamiltonian.

```
#
# Using the propagator to find the steady state of a driven system.
#
from qutip import *
from pylab import *


def run():

    #
    # configure the parameters
    #
    delta = 0.075 * 2 * pi  # qubit sigma_x coefficient
    eps0 = 0.0 * 2 * pi     # qubit sigma_z coefficient
    A = 2.0 * 2 * pi        # sweep rate
    gamma1 = 0.0001         # relaxation rate
    gamma2 = 0.005          # dephasing  rate
    psi0 = basis(2, 0)      # initial state
    omega = 0.05 * 2 * pi   # driving frequency
    T = (2 * pi) / omega    # driving period

    #
```

```
# Hamiltonian
#
sx = sigmax()
sz = sigmaz()
sm = destroy(2)

H0 = - delta / 2.0 * sx - eps0 / 2.0 * sz
H1 = - A / 2.0 * sz

# alternative 1: using function callback format (H_func_t)
# args = [H0, H1, omega]
# def hamiltonian_t(t, args):
#    H0 = args[0]
#    H1 = args[1]
#    w  = args[2]
#    return H0 + cos(w * t) * H1

# alternative 2: using list-callback format
args = {'w': omega}

def H1_coeff_t(t, args):
    return cos(args['w'] * t)

hamiltonian_t = [H0, [H1, H1_coeff_t]]

# alternative 3: using list-string format
# args = {'w': omega}
# hamiltonian_t = [H0, [H1, 'cos(w * t)']]

#
# collapse operators
#
c_ops = []

n_th = 0.0  # temperature in terms of the bath excitation number

rate = gamma1 * (1 + n_th)
if rate > 0.0:
    c_ops.append(sqrt(rate) * sm)       # relaxation

rate = gamma1 * n_th
if rate > 0.0:
    c_ops.append(sqrt(rate) * sm.dag())  # excitation

rate = gamma2
if rate > 0.0:
    c_ops.append(sqrt(rate) * sz)       # dephasing

#
# evolve for five driving periods
#
tlist = linspace(0.0, 5 * T, 1500)
output = mesolve(hamiltonian_t, psi0, tlist, c_ops, [sm.dag() * sm], args)

#
# find the propagator for one driving period
#
T = 2 * pi / omega
U = propagator(hamiltonian_t, T, c_ops, args)

#
# find the steady state of repeated applications of the propagator
# (i.e., t -> inf)
```

```
    #
    rho_ss = propagator_steadystate(U)
    p_ex_ss = expect(sm.dag() * sm, rho_ss)

    #
    # plot the results
    #
    figure(1)

    subplot(211)
    plot(tlist, output.expect[0], 'b')
    plot(tlist, 1 - output.expect[0], 'r')
    plot(tlist, ones(shape(tlist)) * p_ex_ss, 'k', linewidth=2)
    xlabel('Time')
    ylabel('Probability')
    title('Occupation probabilty of qubit [NEW]')
    legend((r"$\left|1\right>$", r"$\left|0\right>$",
           r"$\left|1\right>$ steady state"), loc=0)

    subplot(212)
    plot(tlist, -delta / 2.0 * ones(shape(tlist)), 'r')
    plot(tlist, -(eps0 / 2.0 + A / 2.0 * cos(omega * tlist)), 'b')
    legend(("$\sigma_x$ coefficient", "$\sigma_z$ coefficient"))
    xlabel('Time')
    ylabel('Coefficients in the Hamiltonian')

    show()


if __name__ == '__main__':
    run()
```



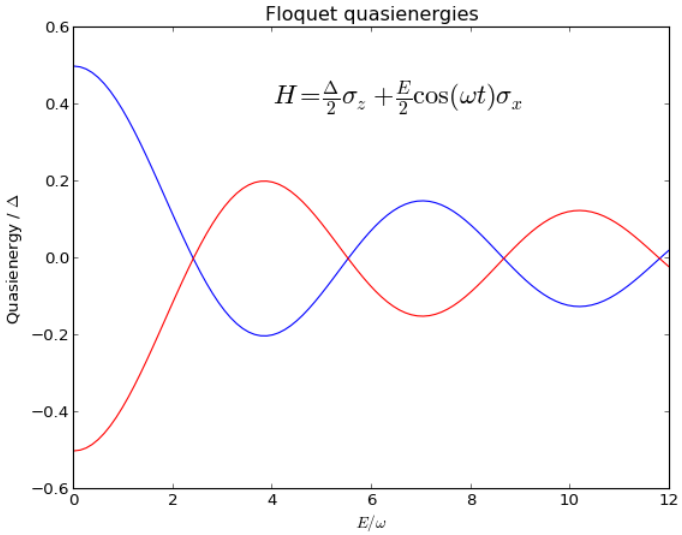
Floquet quasienergy levels for a driven two-level system

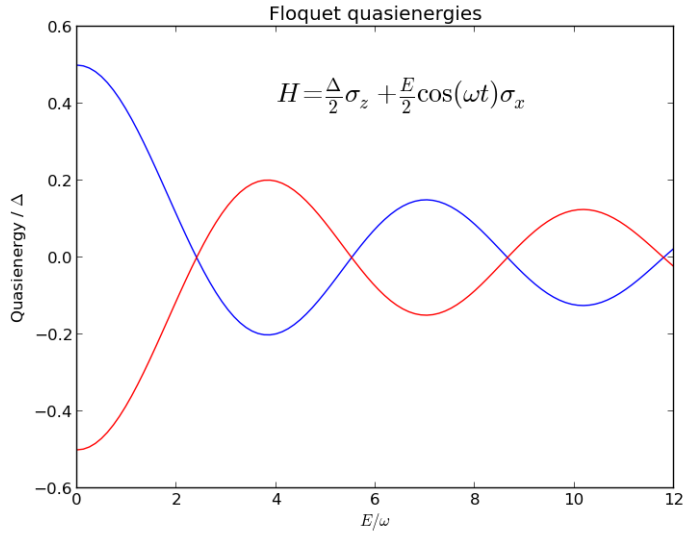
This example demonstrates how to calculate the Floquet quasienergies for a driven system. The example is taken from Creffield et al., Phys. Rev. B 67, 165301 (2003), see Fig. 1(a) in that paper. The Hamiltonian is

$$H(t) = \frac{\Delta}{2}\sigma_z + \frac{E}{2}\cos(\omega t)\sigma_z,$$

and we use the QuTiP function `qutip.floquet.floquet_modes` to obtain the Floquet modes and the quasienergies.

```
#
# Calculate the quasienergies for a driven two-level system as a function of
# driving amplitude. See Creffield et al., Phys. Rev. B 67, 165301 (2003).
#

from qutip import *
from pylab import *


def run():

    delta = 1.0 * 2 * pi    # bare qubit sigma_z coefficient
    epsilon = 0.0 * 2 * pi  # bare qubit sigma_x coefficient
    omega = 8.0 * 2 * pi    # driving frequency
    T = (2 * pi) / omega    # driving period

    E_vec = linspace(0.0, 12.0, 100) * omega

    sx = sigmax()
    sz = sigmaz()

    q_energies = zeros((len(E_vec), 2))

    H0 = delta / 2.0 * sz - epsilon / 2.0 * sx
    for idx, A in enumerate(E_vec):
        H1 = A / 2.0 * sx

        # H = H0 + H1 * sin(w * t) in the 'list-string' format
        args = {'w': omega}
        H = [H0, [H1, 'cos(w * t)']]

        # find the floquet modes
        f_modes, f_energies = floquet_modes(H, T, args)

        q_energies[idx, :] = f_energies

    # plot the results
    plot(E_vec / omega, real(q_energies[:, 0]) / delta, 'b', E_vec /
         omega, real(q_energies[:, 1]) / delta, 'r')
    xlabel(r'$E/\omega$')
    ylabel(r'Quasienergy / $\Delta$')
    title(r'Floquet quasienergies')
    text(4, 0.4,
         r'$H = \frac{\Delta}{2}\sigma_z + \frac{E}{2}\cos(\omega t)\sigma_x$',
         fontsize=20)
    show()

if __name__ == '__main__':
    run()
```





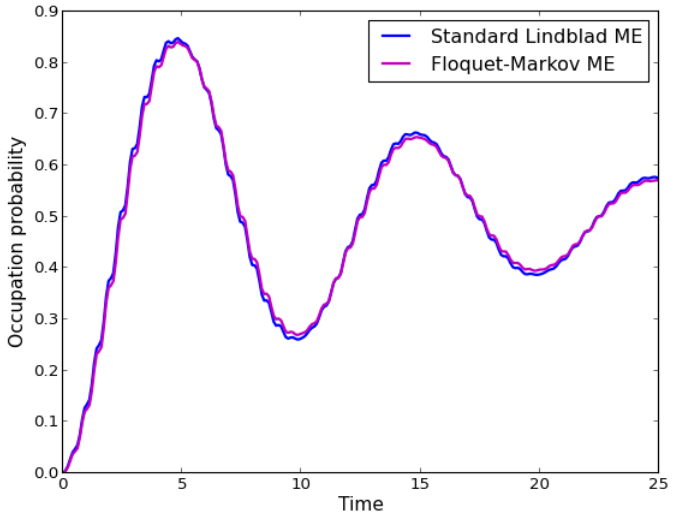
Floquet-Markov master equation

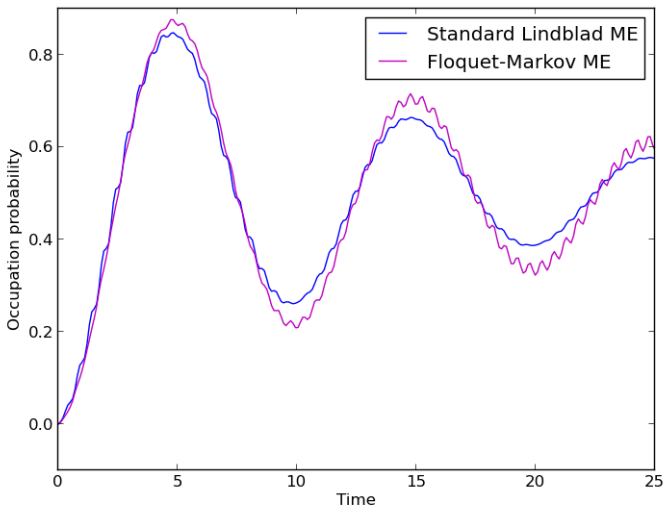
In this example we demonstrate how to use the Floquet-Markov master equation solver in QuTiP by revisting the vacuum Rabi oscillation problem: i.e., a simple two level system subject to a driving field (classical in this example) and dissipation due to its interaction with the environment. We use the QuTiP function `qutip.floquet.fmmesolve` to obtain the system dynamics. For comparison we also calculate the dynamics using the standard Lindblad master eqaution. For weak driving and dissipation the two solvers should give similar results, but not necessarily when the driving amplitude or dissipation rates are large compared to the two-level energy splitting.

```
#
# Calculate the dynamics of a driven two-level system with according to the
# Floquet-Markov master equation. For compari
#
from qutip import *
from pylab import *

gamma1 = 0.05  # relaxation rate
gamma2 = 0.0   # dephasing  rate
```

```
def J_cb(omega):
    """ Noise spectral density """
    return 0.5 * gamma1 * omega / (2 * pi)


def run():

    delta = 0.0 * 2 * pi  # qubit sigma_x coefficient
    eps0 = 1.0 * 2 * pi   # qubit sigma_z coefficient
    A = 0.1 * 2 * pi      # driving amplitude
    w = 1.0 * 2 * pi      # driving frequency
    T = 2 * pi / w        # driving period
    psi0 = basis(2, 0)    # initial state
    tlist = linspace(0, 25.0, 250)

    # Hamiltonian: list-string format
    args = {'w': w}
    H0 = - delta / 2.0 * sigmax() - eps0 / 2.0 * sigmaz()
    H1 = - A * sigmax()
    H = [H0, [H1, 'sin(w * t)']]

    #
    # Standard lindblad master equation with time-dependent hamiltonian
    #
    c_op_list = [sqrt(gamma1) * sigmax(), sqrt(gamma2) * sigmaz()]
    p_ex_me = mesolve(H, psi0, tlist, c_op_list, [num(2)], args=args).expect[0]

    #
    # Floquet markov master equation dynamics
    #
    rhs_clear()  # clears previous time-dependent Hamiltonian data

    # find initial floquet modes and quasienergies
    f_modes_0, f_energies = floquet_modes(H, T, args, False)

    # precalculate floquet modes for the first driving period
    f_modes_table_t = floquet_modes_table(f_modes_0, f_energies,
                                          linspace(0, T, 500 + 1), H, T, args)

    # solve the floquet-markov master equation
    rho_list = fmmesolve(H, psi0, tlist,
                         [sigmax()], [], [J_cb], T, args).states

    # calculate expectation values in the computational basis
    p_ex_fmme = zeros(shape(p_ex_me))
    for idx, t in enumerate(tlist):
        f_modes_t = floquet_modes_t_lookup(f_modes_table_t, t, T)
        p_ex_fmme[idx] = expect(num(2),
                                rho_list[idx].transform(f_modes_t, False))

    # plot the results
    figure()
    plot(tlist, p_ex_me, 'b', lw=2)     # standard lindblad with time-dependence
    plot(tlist, p_ex_fmme, 'm-', lw=2)  # floquet markov
    xlabel('Time', fontsize=14)
    ylabel('Occupation probability', fontsize=14)
    legend(("Standard Lindblad ME", "Floquet-Markov ME"))
    show()


if __name__ == '__main__':
```

```
    run()
```





5.2.6 Advanced topics and examples

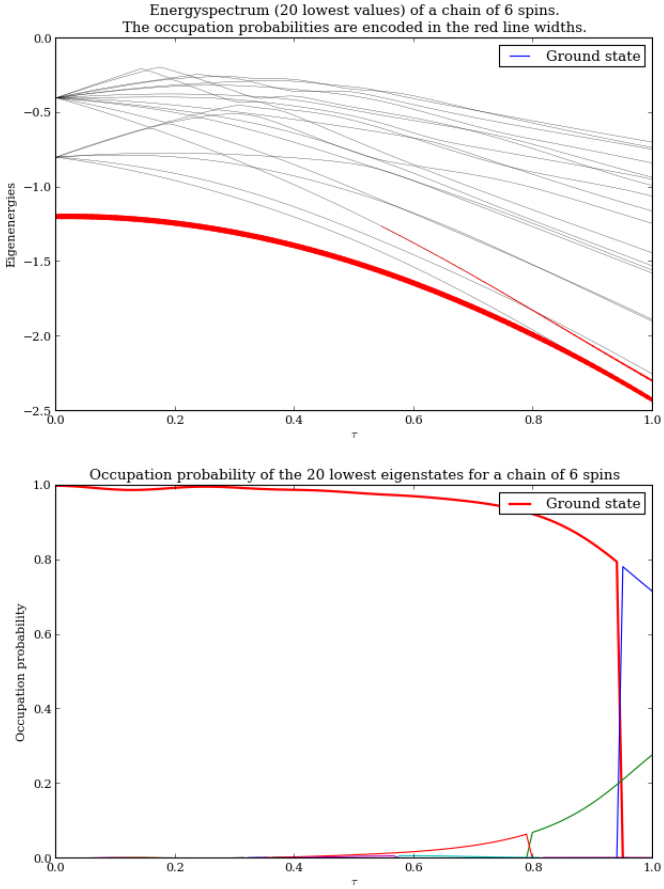
Nonadiabatic transformation of a spin-chain Hamiltonian

```
#
# Nonadiabatic sweep: Gradually transform a simple decoupled spin chain
# hamiltonian to a complicated interacting spin chain.
#
from qutip import *
from pylab import *


def compute(N, M, h, Jx, Jy, Jz, taulist):

    # pre-allocate operators
    si = qeye(2)
    sx = sigmax()
```

```
sy = sigmay()
sz = sigmaz()

sx_list = []
sy_list = []
sz_list = []

for n in range(N):
    op_list = [si] * N

    op_list[n] = sx
    sx_list.append(tensor(op_list))

    op_list[n] = sy
    sy_list.append(tensor(op_list))

    op_list[n] = sz
    sz_list.append(tensor(op_list))

#
# Construct the initial hamiltonian and state vector
#
psi_list = [basis(2, 0) for n in range(N)]
psi0 = tensor(psi_list)
H0 = 0
for n in range(N):
    H0 += - 0.5 * 2.5 * sz_list[n]

#
# Construct the target hamiltonian
#

# energy splitting terms
H1 = 0
for n in range(N):
    H1 += - 0.5 * h[n] * sz_list[n]

H1 = 0
for n in range(N - 1):
    # interaction terms
    H1 += - 0.5 * Jx[n] * sx_list[n] * sx_list[n + 1]
    H1 += - 0.5 * Jy[n] * sy_list[n] * sy_list[n + 1]
    H1 += - 0.5 * Jz[n] * sz_list[n] * sz_list[n + 1]

# the time-dependent hamiltonian in list-string format
args = max(taulist)
h_t = [[H0, lambda t, t_max: (t_max - t) / t_max],
       [H1, lambda t, t_max: t / t_max]]

#
# callback function for each time-step
#
evals_mat = zeros((len(taulist), M))
occupation_mat = zeros((len(taulist), M))

idx = [0]

def process_rho(tau, psi):

    # evaluate the Hamiltonian with gradually switched on interaction
    H = qobj_list_evaluate(h_t, tau, args)

    # find the M lowest eigenvalues of the system
```

```
        evals, ekets = H.eigenstates(eigvals=M)

        evals_mat[idx[0], :] = real(evals)

        # find the overlap between the eigenstates and psi
        for n, eket in enumerate(ekets):
            occupation_mat[idx[0], n] = abs((eket.dag().data *
                                             psi.data)[0, 0]) ** 2

        idx[0] += 1

    #
    # Evolve the system, request the solver to call process_rho at each time
    # step.
    #
    mesolve(h_t, psi0, taulist, [], process_rho, args)

    return evals_mat, occupation_mat


def run():

    #
    # set up the paramters
    #
    N = 6            # number of spins
    M = 20           # number of eigenenergies to solve for

    # array of spin energy splittings and coupling strengths (random values).
    h = 1.0 * 2 * pi * (1 - 2 * rand(N))
    Jz = 1.0 * 2 * pi * (1 - 2 * rand(N))
    Jx = 1.0 * 2 * pi * (1 - 2 * rand(N))
    Jy = 1.0 * 2 * pi * (1 - 2 * rand(N))

    # increase taumax to get make the sweep more adiabatic
    taumax = 5.0
    taulist = linspace(0, taumax, 100)

    evals_mat, occ_mat = compute(N, M, h, Jx, Jy, Jz, taulist)

    #
    # plots
    #
    rc('font', family='serif')
    rc('font', size='10')

    figure(figsize=(9, 12))

    #
    # plot the energy eigenvalues
    #
    subplot(2, 1, 1)

    # first draw thin lines outlining the energy spectrum
    for n in range(len(evals_mat[0, :])):
        if n == 0:
            ls = 'b'
            lw = 1
        else:
            ls = 'k'
            lw = 0.25
        plot(taulist / max(taulist), evals_mat[:, n] / (2 * pi), ls,
             linewidth=lw)
```

```
    # second, draw line that encode the occupation probability of each state in
    # its linewidth. thicker line => high occupation probability.
    for idx in range(len(taulist) - 1):
        for n in range(len(occ_mat[0, :])):
            lw = 0.5 + 4 * occ_mat[idx, n]
            if lw > 0.55:
                plot(array([taulist[idx], taulist[idx + 1]]) / taumax,
                     array([evals_mat[idx, n], evals_mat[idx + 1, n]])/(2*pi),
                     'r', linewidth=lw)

    xlabel(r'$\tau$')
    ylabel('Eigenenergies')
    title("Energyspectrum (%d lowest values) of a chain of %d spins.\n" % (M, N)
          + "The occupation probabilities are encoded in the red line widths.")
    legend(("Ground state",))

    #
    # plot the occupation probabilities for the few lowest eigenstates
    #
    subplot(2, 1, 2)
    for n in range(len(occ_mat[0, :])):
        if n == 0:
            plot(taulist / max(taulist), 0 + occ_mat[:, n], 'r', linewidth=2)
        else:
            plot(taulist / max(taulist), 0 + occ_mat[:, n])

    xlabel(r'$\tau$')
    ylabel('Occupation probability')
    title("Occupation probability of the %d lowest " % M +
          "eigenstates for a chain of %d spins" % N)
    legend(("Ground state",))

    show()

if __name__ == '__main__':
    run()
```



Comparing the Lindblad and Bloch-Redfield master equations

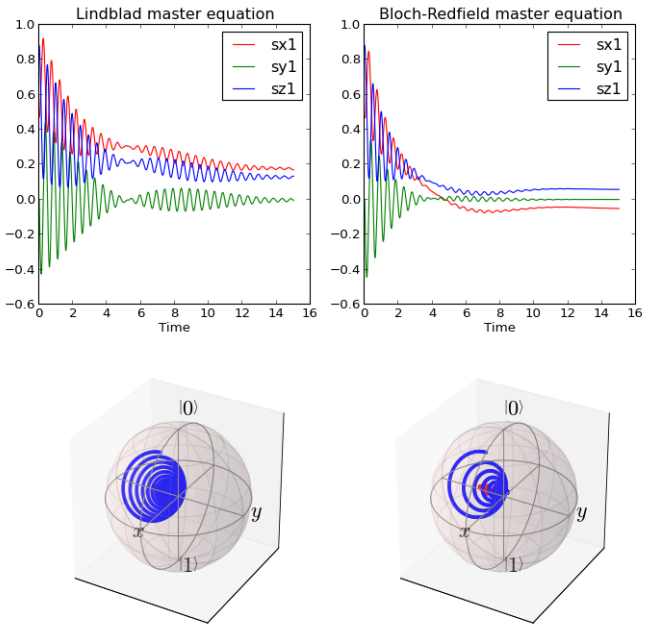
This example compares the results of simulating a two-qubit system with the Lindblad master equation and the Bloch-Redfield master equation. The two models are not intended to be identical, and the results are not similar to each other, but the example demonstrates how a similar physical situation can be modelled in different way and simulated with the two solvers.

This example also demonstrates how to stack Bloch spheres in sub figures.

```
#
# Comparing the Lindblad and Bloch-Redfield master equations
#
from qutip import *
from pylab import *


def qubit_integrate(w, theta, g, gamma1, gamma2, psi0, tlist):
    #
    # Hamiltonian
    #
    sx1 = tensor(sigmax(), qeye(2))
    sy1 = tensor(sigmay(), qeye(2))
    sz1 = tensor(sigmaz(), qeye(2))
    sm1 = tensor(sigmam(), qeye(2))

    sx2 = tensor(qeye(2), sigmax())
```

```
    sy2 = tensor(qeye(2), sigmay())
    sz2 = tensor(qeye(2), sigmaz())
    sm2 = tensor(qeye(2), sigmam())

    H = w[0] * (cos(theta[0]) * sz1 + sin(theta[0]) * sx1)    # qubit 1
    H += w[1] * (cos(theta[1]) * sz2 + sin(theta[1]) * sx2)   # qubit 2
    H += g * sx1 * sx2                                        # interaction

    #
    # Lindblad master equation
    #
    c_op_list = []
    n_th = 0.0  # zero temperature
    rate = gamma1[0] * (n_th + 1)
    if rate > 0.0:
        c_op_list.append(sqrt(rate) * sm1)
    rate = gamma1[1] * (n_th + 1)
    if rate > 0.0:
        c_op_list.append(sqrt(rate) * sm2)

    lme_results = mesolve(H, psi0, tlist, c_op_list, [sx1, sy1, sz1]).expect

    #
    # Bloch-Redfield tensor
    #
    def ohmic_spectrum1(w):
        if w == 0.0:
            # dephasing inducing noise
            return gamma2[0]
        else:
            # relaxation inducing noise
            return gamma1[0] * w / (2 * pi) * (w > 0.0)

    def ohmic_spectrum2(w):
        if w == 0.0:
            # dephasing inducing noise
            return gamma2[1]
        else:
            # relaxation inducing noise
            return gamma1[1] * w / (2 * pi) * (w > 0.0)

    brme_results = brmesolve(H, psi0, tlist, [sx1, sx2], [sx1, sy1, sz1],
                             [ohmic_spectrum1, ohmic_spectrum2]).expect

    return lme_results, brme_results


def run():
    #
    # set up the calculation
    #
    w = array([1.0, 1.0]) * 2 * pi        # qubit angular frequency
    theta = array([0.15, 0.45]) * 2 * pi  # qubit angle from sigma_z axis
    gamma1 = [0.25, 0.35]                 # qubit relaxation rate
    gamma2 = [0.0, 0.0]                   # qubit dephasing rate
    g = 0.1 * 2 * pi
    # initial state
    a = 0.8
    psi1 = (a * basis(2, 0) + (1 - a) * basis(2, 1)).unit()
    psi2 = ((1 - a) * basis(2, 0) + a * basis(2, 1)).unit()
    psi0 = tensor(psi1, psi2)
    # run simulation
    tlist = linspace(0, 15, 5000)
```

```
    lme_results, brme_results = qubit_integrate(
        w, theta, g, gamma1, gamma2, psi0, tlist)

    fig = figure(figsize=(10, 10))
    ax = fig.add_subplot(2, 2, 1)
    title('Lindblad master equation')
    ax.plot(tlist, lme_results[0], 'r')
    ax.plot(tlist, lme_results[1], 'g')
    ax.plot(tlist, lme_results[2], 'b')
    ax.legend(("sx1", "sy1", "sz1"))
    xlabel('Time')

    ax = fig.add_subplot(2, 2, 2)
    title('Bloch-Redfield master equation')
    ax.plot(tlist, brme_results[0], 'r')
    ax.plot(tlist, brme_results[1], 'g')
    ax.plot(tlist, brme_results[2], 'b')
    ax.legend(("sx1", "sy1", "sz1"))
    xlabel('Time')

    # Bloch sphere for Lindblad
    sphere = Bloch(axes=fig.add_subplot(2, 2, 3, projection='3d'))
    sphere.add_vectors([sin(theta[0]), 0, cos(theta[0])])
    sphere.add_points([lme_results[0], lme_results[1], lme_results[2]])
    sphere.vector_color = ['r']
    sphere.point_size = [12]
    sphere.make_sphere()

    # Bloch sphere for Bloch-Redfield
    sphere = Bloch(axes=fig.add_subplot(2, 2, 4, projection='3d'))
    sphere.add_vectors([sin(theta[0]), 0, cos(theta[0])])
    sphere.add_points([brme_results[0], brme_results[1], brme_results[2]])
    sphere.vector_color = ['r']
    sphere.point_size = [12]
    sphere.make_sphere()

    show()

if __name__ == "__main__":
    run()
```



Landau-Zener-Stückelberg interferometry

This is an other example of how to use the propagator of a driven system to calculate its steadystate. The system is the same as considered in *Using the propagator to find the steady state of a driven system*, for which the Hamiltonian is

$$H(t) = -\frac{1}{2}\Delta\sigma_x - \frac{1}{2}\epsilon\sigma_z - \frac{1}{2}A\cos(\omega t)\sigma_z.$$

This example also illustrates how to use the `qutip.parfor` to parallelize the loop over the elements of a matrix.

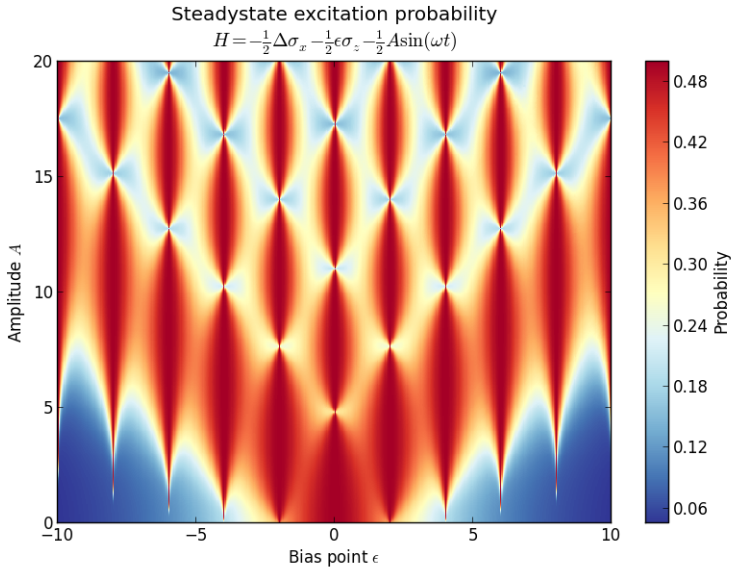
Note: This example takes quite long time to run.

```
#
# Landau-Zener-Stuckelberg interferometry: steady state of repeated Landau-Zener
# like avoided-level crossing, as a function of driving amplitude and bias.
#
from qutip import *

# set up the parameters and start calculation
delta    = 0.1  * 2 * pi  # qubit sigma_x coefficient
w        = 2.0  * 2 * pi  # driving frequency
T        = 2 * pi / w     # driving period
gamma1   = 0.00001        # relaxation rate
gamma2   = 0.005          # dephasing  rate
eps_list = linspace(-10.0, 10.0, 501) * 2 * pi
A_list   = linspace(0.0, 20.0, 501) * 2 * pi

# pre-calculate the necessary operators
sx = sigmax(); sz = sigmaz(); sm = destroy(2); sn = num(2)
# collapse operators
c_op_list = [sqrt(gamma1) * sm, sqrt(gamma2) * sz]  # relaxation and dephasing

# setup time-dependent Hamiltonian (list-string format)
```

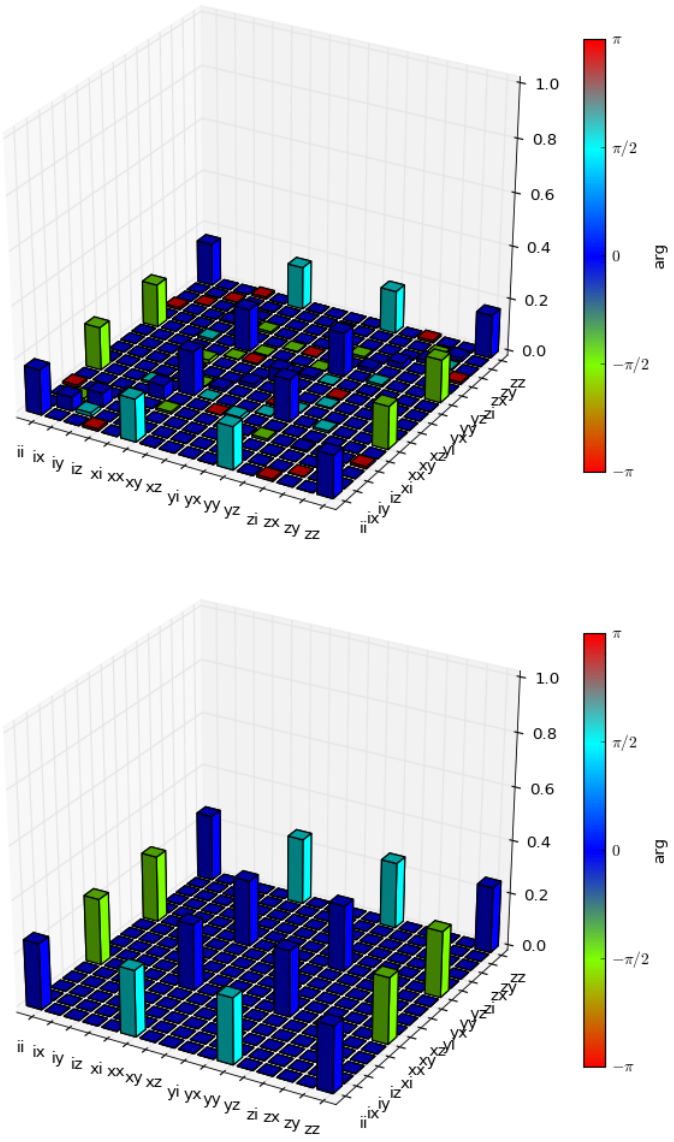
```
H0 = -delta/2.0 * sx
H1 = [sz,'-eps/2.0+A/2.0*sin(w * t)']
H_td = [H0,H1]
Hargs = {'w': w,'eps':eps_list[0],'A':A_list[0]}

# ODE settings (for reusing list-str format Hamiltonian)
opts = Odeoptions(rhs_reuse = True)
#pre-generate RHS so we can use parfor
rhs_generate(H_td,c_op_list,Hargs,name='lz_func')

# a task function for the for-loop parallelization:
# the m-index is parallelized in loop over the elements of p_mat[m,n]
def task(args):
    m, eps = args
    p_mat_m = zeros(len(A_list))
    for n, A in enumerate(A_list):
        # change args sent to solver, w is really a constant though.
        Hargs = {'w': w, 'eps': eps,'A': A}
        U = propagator(H_td, T, c_op_list, Hargs, opts)
        rho_ss = propagator_steadystate(U)
        p_mat_m[n] = expect(sn, rho_ss)
    return [m, p_mat_m]

# start a parallel for loop over bias point values (eps_list)
p_mat_list = parfor(task, enumerate(eps_list))

# assemble a matrix p_mat from list of (index,array) tuples returned by parfor
p_mat = zeros((len(eps_list),len(A_list)))
for m, p_mat_m in p_mat_list:
    p_mat[m,:] = p_mat_m

# Plot the results
from pylab import *
A_mat, eps_mat = meshgrid(A_list/(2*pi), eps_list/(2*pi))
fig = figure()
ax = fig.add_axes([0.1, 0.1, 0.9, 0.8])
c = ax.pcolor(eps_mat,A_mat,p_mat)
c.set_cmap('RdYlBu_r')
cbar = fig.colorbar(c)
cbar.set_label("Probability")
ax.set_xlabel(r'Bias point $\epsilon$')
ax.set_ylabel(r'Amplitude $A$')
ax.autoscale(tight=True)
title('Steadystate excitation probability\n' +
      '$H = -\\frac{1}{2}\\Delta\\sigma_x -\\frac{1}{2}\\epsilon\\sigma_z'+
      ' - \\frac{1}{2}A\\sin(\\omega t)$\n')
show()
```

Process tomography matrix for a 2-qubit iSWAP gate

```
#
# Plot the process tomography matrix for a 2-qubit iSWAP gate.
#
from qutip import *
from pylab import *

def run():
    g = 1.0 * 2 * pi  # coupling strength
    g1 = 0.75         # relaxation rate
    g2 = 0.25         # dephasing rate
    n_th = 1.5        # bath temperature
```




```
    T = pi / (4 * g)
    H = g * (tensor(sigmax(), sigmax()) + tensor(sigmay(), sigmay()))

    c_ops = []
    # qubit 1 collapse operators
    sm1 = tensor(sigmam(), qeye(2))
    sz1 = tensor(sigmaz(), qeye(2))
    c_ops.append(sqrt(g1 * (1 + n_th)) * sm1)
    c_ops.append(sqrt(g1 * n_th) * sm1.dag())
    c_ops.append(sqrt(g2) * sz1)
    # qubit 2 collapse operators
    sm2 = tensor(qeye(2), sigmam())
    sz2 = tensor(qeye(2), sigmaz())
    c_ops.append(sqrt(g1 * (1 + n_th)) * sm2)
    c_ops.append(sqrt(g1 * n_th) * sm2.dag())
    c_ops.append(sqrt(g2) * sz2)

    # process tomography basis
    op_basis = [[qeye(2), sigmax(), sigmay(), sigmaz()]] * 2
    op_label = [["i", "x", "y", "z"]] * 2

    # dissipative gate
    U_diss = propagator(H, T, c_ops)
    chi = qpt(U_diss, op_basis)
    qpt_plot_combined(chi, op_label)

    # ideal gate
    U_psi = (-1j * H * T).expm()
    U_ideal = spre(U_psi) * spost(U_psi.dag())
    chi = qpt(U_ideal, op_basis)
    qpt_plot_combined(chi, op_label)

    show()

if __name__ == '__main__':
    run()
```


1.0
0.8
0.6
0.4
0.2
0.0
ii ix iy iz xi xx xy xz yi yx yy yz zi zx zy zz
ii ix iy iz xi xx xy xz yi yx yy yz zi zx zy zz
π
π/2
0
−π/2
−π
arg

1.0
0.8
0.6
0.4
0.2
0.0
ii ix iy iz xi xx xy xz yi yx yy yz zi zx zy zz
ii ix iy iz xi xx xy xz yi yx yy yz zi zx zy zz
π
π/2
0
−π/2
−π
arg

CHAPTER

SIX

API DOCUMENTATION

This chapter contains automatically generated API documentation, including a complete list of QuTiP's public classes and functions.

6.1 QuTiP Classes

6.1.1 The Qobj class

class `Qobj`(*inpt=array([[0]])*, *dims=*[[][]], *shape=*[], *type=None*, *isherm=None*, *fast=False*)

A class for representing quantum objects, such as quantum operators and states.

The Qobj class is the QuTiP representation of quantum operators and state vectors. This class also implements math operations +,-,* between Qobj instances (and / by a C-number), as well as a collection of common operator/state operations. The Qobj constructor optionally takes a dimension `list` and/or shape `list` as arguments.

Parameters **inpt** : array_like

Data for vector/matrix representation of the quantum object.

dims : list

Dimensions of object used for tensor products.

shape : list

Shape of underlying data structure (matrix shape).

fast : bool

Flag for fast qobj creation when running ode solvers. This parameter is used internally only.

Attributes **data** : array_like

Sparse matrix characterizing the quantum object.

dims : list

List of dimensions keeping track of the tensor structure.

shape : list

Shape of the underlying *data* array.

isherm : bool

Indicates if quantum object represents Hermitian operator.

type : str

Type of quantum object: 'bra', 'ket', 'oper', or 'super'.

Methods conj() :

Conjugate of quantum object.

dag() :

Adjoint (dagger) of quantum object.

eigenenergies(sparse=False, sort='low', eigvals=0, tol=0, maxiter=100000) :

Returns eigenenergies (eigenvalues) of a quantum object.

eigenstates(sparse=False, sort='low', eigvals=0, tol=0, maxiter=100000) :

Returns eigenenergies and eigenstates of quantum object.

expm() :

Matrix exponential of quantum object.

full() :

Returns dense array of quantum object *data* attribute.

groundstate(sparse=False,tol=0,maxiter=100000) :

Returns eigenvalue and eigenket for the groundstate of a quantum object.

matrix_element(bra,ket) :

Returns the matrix element of operator between *bra* and *ket* vectors.

norm(oper_norm='tr',sparse=False,tol=0,maxiter=100000) :

Returns norm of operator.

ptrace(sel) :

Returns quantum object for selected dimensions after performing partial trace.

sqrtm() :

Matrix square root of quantum object.

tidyup(atol=1e-12) :

Removes small elements from quantum object.

tr() :

Trace of quantum object.

trans() :

Transpose of quantum object.

transform(inpt,inverse=False) :

Performs a basis transformation defined by *inpt* matrix.

unit(oper_norm='tr',sparse=False,tol=0,maxiter=100000) :

Returns normalized quantum object.

checkherm()

Explicitly check if the Qobj is hermitian

Returns isherm: bool :

Returns the new value of isherm property.

conj()

Returns the conjugate operator of quantum object.

dag()

Returns the adjont operator of quantum object.

diag()
Diagonal elements of Qobj.

Returns **diags: array** :

Returns array of `real` values if operators is Hermitian, otherwise `complex` values are returned.

eigenenergies(*sparse=False*, *sort='low'*, *eigvals=0*, *tol=0*, *maxiter=100000*)
Finds the Eigenenergies (Eigenvalues) of a quantum object.

Eigenenergies are defined for operators or superoperators only.

Parameters **sparse** : bool

Use sparse Eigensolver

sort : str

Sort eigenvalues 'low' to high, or 'high' to low.

eigvals : int

Number of requested eigenvalues. Default is all eigenvalues.

tol : float

Tolerance used by sparse Eigensolver (0=machine precision). The sparse solver may not converge if the tolerance is set too low.

maxiter : int

Maximum number of iterations performed by sparse solver (if used).

Returns **eigvals: array** :

Array of eigenvalues for operator.

Notes

The sparse eigensolver is much slower than the dense version. Use sparse only if memory requirements demand it.

eigenstates(*sparse=False*, *sort='low'*, *eigvals=0*, *tol=0*, *maxiter=100000*)
Find the eigenstates and eigenenergies.

Eigenstates and Eigenvalues are defined for operators and superoperators only.

Parameters **sparse** : bool

Use sparse Eigensolver

sort : str

Sort eigenvalues (and vectors) 'low' to high, or 'high' to low.

eigvals : int

Number of requested eigenvalues. Default is all eigenvalues.

tol : float

Tolerance used by sparse Eigensolver (0 = machine precision). The sparse solver may not converge if the tolerance is set too low.

maxiter : int

Maximum number of iterations performed by sparse solver (if used).

Returns **eigvals** : array

Array of eigenvalues for operator.

eigvecs : array

Array of quantum operators representing the oprator eigenkets. Order of eigenkets is determined by order of eigenvalues.

Notes

The sparse eigensolver is much slower than the dense version. Use sparse only if memory requirements demand it.

expm()
Matrix exponential of quantum operator.

Input operator must be square.

Returns **oper** : qobj

Exponentiated quantum operator.

Raises **TypeError** :

Quantum operator is not square.

full()
Returns a dense array from quantum object.

Returns **data** : array

Array of complex data from quantum objects *data* attribute.

groundstate(*sparse=False*, *tol=0*, *maxiter=100000*)
Finds the ground state Eigenvalue and Eigenvector.

Defined for quantum operators or superoperators only.

Parameters **sparse** : bool

Use sparse Eigensolver

tol : float

Tolerance used by sparse Eigensolver (0 = machine precision). The sparse solver may not converge if the tolerance is set too low.

maxiter : int

Maximum number of iterations performed by sparse solver (if used).

Returns **eigval** : float

Eigenvalue for the ground state of quantum operator.

eigvec : qobj

Eigenket for the ground state of quantum operator.

Notes

The sparse eigensolver is much slower than the dense version. Use sparse only if memory requirements demand it.

matrix_element(*bra*, *ket*)
Calculates a matrix element.

Gives matrix for the Qobj sandwiched between a *bra* and *ket* vector.

Parameters **bra** : qobj

Quantum object of type 'bra'.

ket : qobj

Quantum object of type 'ket'.

Returns **elem** : complex

Complex valued matrix element.

Raises **TypeError** :

Can only calculate matrix elements between a bra and ket quantum object.

norm(*oper_norm='tr'*, *sparse=False*, *tol=0*, *maxiter=100000*)

Returns the norm of a quantum object.

Norm is L2-norm for kets and trace-norm (by default) for operators. Other operator norms may be specified using the *oper_norm* argument.

Parameters **oper_norm** : str

Which norm to use for operators: trace 'tr', Frobius 'fro', one 'one', or max 'max'. This parameter does not affect the norm of a state vector.

sparse : bool

Use sparse eigenvalue solver for trace norm. Other norms are not affected by this parameter.

tol : float

Tolerance for sparse solver (if used) for trace norm. The sparse solver may not converge if the tolerance is set too low.

maxiter : int

Maximum number of iterations performed by sparse solver (if used) for trace norm.

Returns **norm** : float

The requested norm of the operator or state quantum object.

Notes

The sparse eigensolver is much slower than the dense version. Use sparse only if memory requirements demand it.

ptrace(*sel*)

Partial trace of the Qobj with selected components remaining.

Parameters **sel** : int/list

An `int` or `list` of components to keep after partial trace.

Returns **oper: qobj** :

Quantum object representing partial trace with selected components remaining.

Notes

This function is identical to the `qutip.qobj.ptrace` function that has been deprecated.

sqrtm(*sparse=False*, *tol=0*, *maxiter=100000*)

The sqrt of a quantum operator.

Operator must be square.

Parameters **sparse** : bool

Use sparse eigenvalue/vector solver.

tol : float

Tolerance used by sparse solver (0 = machine precision).

maxiter : int

Maximum number of iterations used by sparse solver.

Returns **oper: qobj** :

Matrix square root of operator.

Raises **TypeError** :

Quantum object is not square.

Notes

The sparse eigensolver is much slower than the dense version. Use sparse only if memory requirements demand it.

tidyup(*atol=1e-12*)

Removes small elements from a quantum object.

Parameters **atol** : float

Absolute tolerance used by tidyup. Default is set via qutip global settings parameters.

Returns **oper: qobj** :

Quantum object with small elements removed.

tr()

The trace of a quantum object.

Returns **trace: float** :

Returns `real` if operator is Hermitian, returns `complex` otherwise.

trans()

Transposed operator.

Returns **oper: qobj** :

Transpose of input operator.

transform(*inpt*, *inverse=False*)

Performs a basis transformation defined by an input array.

Input array can be a `matrix` defining the transformation, or a `list` of kets that defines the new basis.

Parameters **inpt** : array_like

A `matrix` or `list` of kets defining the transformation.

inverse : bool

Whether to return inverse transformation.

Returns **oper** : qobj

Operator in new basis.

Notes

This function is still in development.

unit (*oper_norm='tr'*, *sparse=False*, *tol=0*, *maxiter=100000*)
Returns the operator or state normalized to unity.

Uses norm from Qobj.norm().

Parameters **oper_norm** : str

Requested norm for operator. Norm is always L2 for states.

sparse : bool

Use sparse eigensolver for trace norm. Does not affect other norms.

tol : float

Tolerance used by sparse eigensolver.

maxiter: int :

Number of maximum iterations performed by sparse eigensolver.

Returns **oper** : qobj

Normalized quantum object.

6.1.2 The eseries class

class eseries (*q=array([ ], dtype=float64)*, *s=array([ ], dtype=float64)*)
Class representation of an exponential-series expansion of time-dependent quantum objects.

Attributes **ampl** : ndarray

Array of amplitudes for exponential series.

rates : ndarray

Array of rates for exponential series.

dims : list

Dimensions of exponential series components

shape : list

Shape corresponding to exponential series components

Methods **value(tlist)** :

Evaluate an exponential series at the times listed in tlist

spec(wlist) :

Evaluate the spectrum of an exponential series at frequencies in wlist.

tidyup() :

Returns a tidier version of the exponential series

spec (*wlist*)
Evaluate the spectrum of an exponential series at frequencies in `wlist`.

Parameters **wlist** : array_like

Array/list of frequenies.

Returns **val_list** : ndarray

Values of exponential series at frequencies in `wlist`.

tidyup (**args*)
Returns a tidier version of exponential series.

value(*tlist*)

Evaluates an exponential series at the times listed in `tlist`.

Parameters **tlist** : ndarray

Times at which to evaluate exponential series.

Returns **val_list** : ndarray

Values of exponential at times in `tlist`.

6.1.3 The Bloch class

class Bloch(*fig=None*, *axes=None*)

Class for plotting data on the Bloch sphere. Valid data can be either points, vectors, or qobj objects.

Attributes **axes** : instance {None}

User supplied Matplotlib axes for Bloch sphere animation.

fig : instance {None}

User supplied Matplotlib Figure instance for plotting Bloch sphere.

font_color : str {'black'}

Color of font used for Bloch sphere labels.

font_size : int {20}

Size of font used for Bloch sphere labels.

frame_alpha : float {0.1}

Sets transparency of Bloch sphere frame.

frame_color : str {'gray'}

Color of sphere wireframe.

frame_width : int {1}

Width of wireframe.

point_color : list {["b","r","g","#CC6600"]}

List of colors for Bloch sphere point markers to cycle through. i.e. By default, points 0 and 4 will both be blue ('b').

point_marker : list {["o","s","d","^"]}

List of point marker shapes to cycle through.

point_size : list {[25,32,35,45]}

List of point marker sizes. Note, not all point markers look the same size when plotted!

sphere_alpha : float {0.2}

Transparency of Bloch sphere itself.

sphere_color : str {'#FFDDDD'}

Color of Bloch sphere.

size : list {[7,7]}

Size of Bloch sphere plot. Best to have both numbers the same; otherwise you will have a Bloch sphere that looks like a football.

vector_color : list {["g","#CC6600","b","r"]}

List of vector colors to cycle through.

vector_width : int {5}

Width of displayed vectors.

vector_style : str {'-|>', 'simple', 'fancy', ''}

Vector arrowhead style (from matplotlib's arrow style).

vector_mutation : int {20}

Width of vectors arrowhead.

view : list {[-60,30]}

Azimuthal and Elevation viewing angles.

xlabel : list {["x","]}

List of strings corresponding to +x and -x axes labels, respectively.

xlpos : list {[1.1,-1.1]}

Positions of +x and -x labels respectively.

ylabel : list {["y","]}

List of strings corresponding to +y and -y axes labels, respectively.

ylpos : list {[1.2,-1.2]}

Positions of +y and -y labels respectively.

zlabel : list {['$left|0right>$','$left|1right>$']}

List of strings corresponding to +z and -z axes labels, respectively.

zlpos : list {[1.2,-1.2]}

Positions of +z and -z labels respectively.

add_points(*points*, *meth='s'*)

Add a list of data points to bloch sphere.

Parameters **points** : array/list

Collection of data points.

meth : str {'s','m'}

Type of points to plot, use 'm' for multicolored.

add_states(*state*, *kind='vector'*)

Add a state vector Qobj to Bloch sphere.

Parameters **state** : qobj

Input state vector.

kind : str {'vector','point'}

Type of object to plot.

add_vectors(*vectors*)

Add a list of vectors to Bloch sphere.

Parameters **vectors** : array/list

Array with vectors of unit length or smaller.

clear()

Resets Bloch sphere data sets to empty.

make_sphere()

Plots Bloch sphere and data sets.

save(*name=None*, *format='png'*, *dirc=None*)
Saves Bloch sphere to file of type `format` in directory `dirc`.

Parameters **name** : str

Name of saved image. Must include path and format as well. i.e. '/Users/Paul/Desktop/bloch.png' This overrides the 'format' and 'dirc' arguments.

format : str

Format of output image.

dirc : str

Directory for output images. Defaults to current working directory.

Returns **File containing plot of Bloch sphere.** :

show()
Display Bloch sphere and corresponding data sets.

vector_mutation = None
Sets the width of the vectors arrowhead

vector_style = None
Style of Bloch vectors, default = '-|>' (or 'simple')

vector_width = None
Width of Bloch vectors, default = 5

6.1.4 The Bloch3d class

class Bloch3d(*fig=None*)
Class for plotting data on a 3D Bloch sphere using mayavi. Valid data can be either points, vectors, or qobj objects corresponding to state vectors or density matricies for a two-state system (or subsystem).

Attributes **fig** : instance {None}

User supplied Matplotlib Figure instance for plotting Bloch sphere.

font_color : str {'black'}

Color of font used for Bloch sphere labels.

font_scale : float {0.08}

Scale for font used for Bloch sphere labels.

frame : bool {True}

Draw frame for Bloch sphere

frame_alpha : float {0.05}

Sets transparency of Bloch sphere frame.

frame_color : str {'gray'}

Color of sphere wireframe.

frame_num : int {8}

Number of frame elements to draw.

frame_radius : floats {0.005}

Width of wireframe.

point_color : list {['r', 'g', 'b', 'y']}

List of colors for Bloch sphere point markers to cycle through. i.e. By default, points 0 and 4 will both be blue ('r').

point_mode : string {'sphere','cone','cube','cylinder','point'}

Point marker shapes.

point_size : float {0.075}

Size of points on Bloch sphere.

sphere_alpha : float {0.1}

Transparency of Bloch sphere itself.

sphere_color : str {'#808080'}

Color of Bloch sphere.

size : list {[500,500]}

Size of Bloch sphere plot in pixels. Best to have both numbers the same otherwise you will have a Bloch sphere that looks like a football.

vector_color : list {['r', 'g', 'b', 'y']}

List of vector colors to cycle through.

vector_width : int {3}

Width of displayed vectors.

view : list {[45,65]}

Azimuthal and Elevation viewing angles.

xlabel : list {['|x>', '']}

List of strings corresponding to +x and -x axes labels, respectively.

xlpos : list {[1.07,-1.07]}

Positions of +x and -x labels respectively.

ylabel : list {['|y>', '']}

List of strings corresponding to +y and -y axes labels, respectively.

ylpos : list {[1.07,-1.07]}

Positions of +y and -y labels respectively.

zlabel : list {['|0>', '|1>']}

List of strings corresponding to +z and -z axes labels, respectively.

zlpos : list {[1.07,-1.07]}

Positions of +z and -z labels respectively.

Notes

The use of mayavi for 3D rendering of the Bloch sphere comes with a few limitations: I) You can not embed a Bloch3d figure into a matplotlib window. II) The use of LaTex is not supported by the mayavi rendering engine. Therefore all labels must be defined using standard text. Of course you can post-process the generated figures later to add LaTeX using other software if needed.

add_points(*points*, *meth='s'*)

Add a list of data points to bloch sphere.

Parameters **points** : array/list

Collection of data points.

meth : str {'s','m'}

Type of points to plot, use 'm' for multicolored.

add_states(*state*, *kind='vector'*)
Add a state vector Qobj to Bloch sphere.

Parameters **state** : qobj

Input state vector.

kind : str {'vector','point'}

Type of object to plot.

add_vectors(*vectors*)
Add a list of vectors to Bloch sphere.

Parameters **vectors** : array/list

Array with vectors of unit length or smaller.

clear()
Resets the Bloch sphere data sets to empty.

make_sphere()
Plots Bloch sphere and data sets.

plot_points()
Plots points on the Bloch sphere.

plot_vectors()
Plots vectors on the Bloch sphere.

save(*name=None*, *format='png'*, *dirc=None*)
Saves Bloch sphere to file of type `format` in directory `dirc`.

Parameters **name** : str

Name of saved image. Must include path and format as well. i.e. '/Users/Paul/Desktop/bloch.png' This overrides the 'format' and 'dirc' arguments.

format : str

Format of output image. Default is 'png'.

dirc : str

Directory for output images. Defaults to current working directory.

Returns **File containing plot of Bloch sphere.** :

show()
Display the Bloch sphere and corresponding data sets.

6.1.5 The Odeoptions class

class Odeoptions(*atol=1e-08*, *rtol=1e-06*, *method='adams'*, *order=12*, *nsteps=1000*, *first_step=0*, *max_step=0*, *min_step=0*, *mc_avg=True*, *tidy=True*, *num_cpus=0*, *norm_tol=0.001*, *norm_steps=5*, *rhs_reuse=False*, *rhs_filename=None*, *gui=True*, *ntraj=500*)
Class of options for ODE solver used by `qutip.mesolve` and `qutip.mcsolve`. Options can be changed either inline:

```
opts=Odeoptions(gui=False,order=10,.....)
```

or by changing the class properties after creation:

```
opts=Odeoptions()
opts.gui=False
opts.order=10
```

Returns options class to be used as options in `qutip.mesolve` and `qutip.mcsolve`.

Attributes **atol** : float {1e-8}

Absolute tolerance.

rtol : float {1e-6}

Relative tolerance.

method : str {'adams','bdf'}

Integration method.

order : int {12}

Order of integrator (<=12 'adams', <=5 'bdf')

nsteps : int {2500}

Max. number of internal steps/call.

first_step : float {0}

Size of initial step (0 = automatic).

min_step : float {0}

Minimum step size (0 = automatic).

max_step : float {0}

Maximum step size (0 = automatic)

tidy : bool {True,False}

Tidyup Hamiltonian and initial state by removing small terms.

num_cpus : int

Number of cpus used by mcsolver (default = # of cpus).

norm_tol :float :

Tolerance used when finding wavefunction norm in mcsolve.

norm_steps : int

Max. number of steps used to find wavefunction norm to within norm_tol in mcsolve.

gui : bool {True,False}

Use progress bar GUI for mcsolver.

mc_avg : bool {True,False}

Avg. expectation values in mcsolver.

ntraj : int {500}

Number of trajectories in mcsolver.

rhs_reuse : bool {False,True}

Reuse Hamiltonian data.

rhs_filename : str

Name for compiled Cython file.

6.1.6 The Odedata class

class `Odedata`

Class for storing simulation results from any of the dynamics solvers.

Attributes **solver** : str

Which solver was used ['mesolve','mcsolve','brsolve','floquet']

times : list/array

Times at which simulation data was collected.

expect : list/array

Expectation values (if requested) for simulation.

states : array

State of the simulation (density matrix or ket) evaluated at `times`.

num_expect : int

Number of expectation value operators in simulation.

num_collapse : int

Number of collapse operators in simualation.

ntraj : int/list

Number of monte-carlo trajectories (if using mcsolve). List indicates that averaging of expectation values was done over a subset of total number of trajectories.

col_times : list

Times at which state collpase occurred. Only for Monte-Carlo solver.

col_which : list

Which collapse operator was responsible for each collapse in `col_times`. mcsolver only.

6.2 QuTiP Functions

6.2.1 Manipulation and Creation of States and Operators

Quantum States

basis(*N*, **args*)

Generates the vector representation of a Fock state.

Parameters **N** : int

Number of Fock states in Hilbert space.

args : int

`int` corresponding to desired number state, defaults to 0 if omitted.

Returns **state** : qobj

Qobj representing the requested number state `|args>`.

Notes

A subtle incompability with the quantum optics toolbox: In QuTiP:

```
basis(N, 0) = ground state
```

but in qotoolbox:

```
basis(N, 1) = ground state
```

Examples

```
>>> basis(5,2)
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[ 0.+0.j]
 [ 0.+0.j]
 [ 1.+0.j]
 [ 0.+0.j]
 [ 0.+0.j]]
```

coherent (*N*, *alpha*, *method='operator'*)

Generates a coherent state with eigenvalue alpha.

Constructed using displacement operator on vacuum state.

Parameters **N** : int

Number of Fock states in Hilbert space.

alpha : float/complex

Eigenvalue of coherent state.

method : string {'operator', 'analytic'}

Method for generating coherent state.

Returns **state** : qobj

Qobj quantum object for coherent state

Notes

Select method 'operator' (default) or 'analytic'. With the 'operator' method, the coherent state is generated by displacing the vacuum state using the displacement operator defined in the truncated Hilbert space of size 'N'. This method guarantees that the resulting state is normalized. With 'analytic' method the coherent state is generated using the analytical formula for the coherent state coefficients in the Fock basis. This method does not guarantee that the state is normalized if truncated to a small number of Fock states, but would in that case give more accurate coefficients.

Examples

```
>>> coherent(5,0.25j)
Quantum object: dims = [[5], [1]], shape = [5, 1], type = ket
Qobj data =
[[  9.69233235e-01+0.j        ]
 [  0.00000000e+00+0.24230831j]
 [ -4.28344935e-02+0.j        ]
 [  0.00000000e+00-0.00618204j]
 [  7.80904967e-04+0.j        ]]
```

coherent_dm(*N*, *alpha*, *method='operator'*)

Density matrix representation of a coherent state.

Constructed via outer product of `qutip.states.coherent`

Parameters **N** : int

Number of Fock states in Hilbert space.

alpha : float/complex

Eigenvalue for coherent state.

method : string {'operator', 'analytic'}

Method for generating coherent density matrix.

Returns **dm** : qobj

Density matrix representation of coherent state.

Notes

Select method 'operator' (default) or 'analytic'. With the 'operator' method, the coherent density matrix is generated by displacing the vacuum state using the displacement operator defined in the truncated Hilbert space of size 'N'. This method guarantees that the resulting density matrix is normalized. With 'analytic' method the coherent density matrix is generated using the analytical formula for the coherent state coefficients in the Fock basis. This method does not guarantee that the state is normalized if truncated to a small number of Fock states, but would in that case give more accurate coefficients.

Examples

```
>>> coherent_dm(3,0.25j)
Quantum object: dims = [[3], [3]], shape = [3, 3], type = oper, isHerm = True
Qobj data =
[[ 0.93941695+0.j          0.00000000-0.23480733j -0.04216943+0.j        ]
 [ 0.00000000+0.23480733j  0.05869011+0.j          0.00000000-0.01054025j]
 [-0.04216943+0.j          0.00000000+0.01054025j  0.00189294+0.j        ]]
```

fock(*N*, **args*)

Bosonic Fock (number) state.

Same as `qutip.states.basis`.

Parameters **N** : int

Number of states in the Hilbert space.

m : int

`int` for desired number state, defaults to 0 if omitted.

Returns **Requested number state :math:'left|mathrm{args}right>'.** :

Examples

```
>>> fock(4,3)
Quantum object: dims = [[4], [1]], shape = [4, 1], type = ket
Qobj data =
[[ 0.+0.j]
 [ 0.+0.j]
 [ 0.+0.j]
 [ 1.+0.j]]
```

fock_dm(*N*, **args*)

Density matrix representation of a Fock state

Constructed via outer product of `qutip.states.fock`.

Parameters **N** : int

Number of Fock states in Hilbert space.

m : int

`int` for desired number state, defaults to 0 if omitted.

Returns **dm** : qobj

Density matrix representation of Fock state.

Examples

```
>>> fock_dm(3,1)
Quantum object: dims = [[3], [3]], shape = [3, 3], type = oper, isHerm = True
Qobj data =
[[ 0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  1.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j]]
```

ket2dm(*Q*)

Takes input ket or bra vector and returns density matrix formed by outer product.

Parameters **Q** : qobj

Ket or bra type quantum object.

Returns **dm** : qobj

Density matrix formed by outer product of *Q*.

Examples

```
>>> x=basis(3,2)
>>> ket2dm(x)
Quantum object: dims = [[3], [3]], shape = [3, 3], type = oper, isHerm = True
Qobj data =
[[ 0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  1.+0.j]]
```

qutrit_basis()

Basis states for a three level system (qutrit)

Returns **qstates** : array

Array of qutrit basis vectors

thermal_dm(*N*, *n*, *method='operator'*)

Density matrix for a thermal state of n particles

Parameters **N** : int

Number of basis states in Hilbert space.

n : float

Expectation value for number of particles in thermal state.

method : string {'operator', 'analytic'}

`string` that sets the method used to generate the thermal state probabilities

Returns **dm** : qobj

Thermal state density matrix.

Notes

The 'operator' method (default) generates the thermal state using the truncated number operator `num(N)`. This is the method that should be used in computations. The 'analytic' method uses the analytic coefficients derived in an infinite Hilbert space. The analytic form is not necessarily normalized, if truncated too agressively.

Examples

```
>>> thermal_dm(5, 1)
Quantum object: dims = [[5], [5]], shape = [5, 5], type = oper, isHerm = True
Qobj data =
[[ 0.51612903  0.          0.          0.          0.        ]
 [ 0.          0.25806452  0.          0.          0.        ]
 [ 0.          0.          0.12903226  0.          0.        ]
 [ 0.          0.          0.          0.06451613  0.        ]
 [ 0.          0.          0.          0.          0.03225806]]

>>> thermal_dm(5, 1, 'analytic')
Quantum object: dims = [[5], [5]], shape = [5, 5], type = oper, isHerm = True
Qobj data =
[[ 0.5      0.       0.       0.       0.     ]
 [ 0.       0.25     0.       0.       0.     ]
 [ 0.       0.       0.125    0.       0.     ]
 [ 0.       0.       0.       0.0625   0.     ]
 [ 0.       0.       0.       0.       0.03125]]
```

state_number_enumerate(*dims*, *state=None*, *idx=0*)

An iterator that enumerate all the state number arrays (quantum numbers on the form [n1, n2, n3, ...]) for a system with dimensions given by dims.

Example:

```
>>> for state in state_number_enumerate([2,2]):
>>>     print state
[ 0.  0.]
[ 0.  1.]
[ 1.  0.]
[ 1.  1.]
```

Parameters **dims** : list or array

The quantum state dimensions array, as it would appear in a Qobj.

state : list

Current state in the iteration. Used internally.

idx : integer

Current index in the iteration. Used internally.

Returns **state_number** : list

Successive state number arrays that can be used in loops and other iterations, using standard state enumeration *by definition*.

state_number_index(*dims*, *state*)

Return the index of a quantum state corresponding to state, given a system with dimensions given by dims.

Example:

```
>>> state_number_index([2, 2, 2], [1, 1, 0])
6.0
```

Parameters **dims** : list or array

The quantum state dimensions array, as it would appear in a Qobj.

state : list

State number array.

Returns **idx** : list

The index of the state given by *state* in standard enumeration ordering.

state_index_number(*dims*, *index*)

Return a quantum number representation given a state index, for a system of composite structure defined by dims.

Example:

```
>>> state_index_number([2, 2, 2], 6)
[1, 1, 0]
```

Parameters **dims** : list or array

The quantum state dimensions array, as it would appear in a Qobj.

index : integer

The index of the state in standard enumeration ordering.

Returns **state** : list

The state number array corresponding to index *index* in standard enumeration ordering.

state_number_qobj(*dims*, *state*)

Return a Qobj representation of a quantum state specified by the state array *state*.

Example:

```
>>> state_number_qobj([2, 2, 2], [1, 0, 1])
Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = [8, 1], type = ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 1.]
 [ 0.]
 [ 0.]]
```

Parameters **dims** : list or array

The quantum state dimensions array, as it would appear in a Qobj.

state : list

State number array.

Returns **state** : `qutip.Qobj.qobj`

The state as a `qutip.Qobj.qobj` instance.

Quantum Operators

create(*N*)

Creation (raising) operator.

Parameters **N** : int

Dimension of Hilbert space.

Returns **oper** : qobj

Qobj for raising operator.

Examples

```
>>> create(4)
Quantum object: dims = [[4], [4]], shape = [4, 4], type = oper, isHerm = False
Qobj data =
[[ 0.00000000+0.j  0.00000000+0.j  0.00000000+0.j  0.00000000+0.j]
 [ 1.00000000+0.j  0.00000000+0.j  0.00000000+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  1.41421356+0.j  0.00000000+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.00000000+0.j  1.73205081+0.j  0.00000000+0.j]]
```

destroy(*N*)

Destruction (lowering) operator.

Parameters **N** : int

Dimension of Hilbert space.

Returns **oper** : qobj

Qobj for lowering operator.

Examples

```
>>> destroy(4)
Quantum object: dims = [[4], [4]], shape = [4, 4], type = oper, isHerm = False
Qobj data =
[[ 0.00000000+0.j  1.00000000+0.j  0.00000000+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.00000000+0.j  1.41421356+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.00000000+0.j  0.00000000+0.j  1.73205081+0.j]
 [ 0.00000000+0.j  0.00000000+0.j  0.00000000+0.j  0.00000000+0.j]]
```

displace(*N*, *alpha*)

Single-mode displacement operator.

Parameters **N** : int

Dimension of Hilbert space.

alpha : float/complex

Displacement amplitude.

Returns **oper** : qobj

Displacement operator.

Examples

```
>>> displace(4,0.25)
Quantum object: dims = [[4], [4]], shape = [4, 4], type = oper, isHerm = False
Qobj data =
[[ 0.96923323+0.j -0.24230859+0.j  0.04282883+0.j -0.00626025+0.j]
 [ 0.24230859+0.j  0.90866411+0.j -0.33183303+0.j  0.07418172+0.j]
 [ 0.04282883+0.j  0.33183303+0.j  0.84809499+0.j -0.41083747+0.j]
 [ 0.00626025+0.j  0.07418172+0.j  0.41083747+0.j  0.90866411+0.j]]
```

jmat (*j*, **args*)

Higher-order spin operators:

Parameters **j** : float

Spin of operator

args : str

Which operator to return 'x','y','z','+','-'. If no args given, then output is ['x','y','z']

Returns **jmat** : qobj/list

`qobj` for requested spin operator(s).

Notes

If no 'args' input, then returns array of ['x','y','z'] operators.

Examples

```
>>> jmat(1)
[ Quantum object: dims = [[3], [3]], shape = [3, 3], type = oper, isHerm = True
Qobj data =
[[ 0.          0.70710678  0.        ]
 [ 0.70710678  0.          0.70710678]
 [ 0.          0.70710678  0.        ]]
 Quantum object: dims = [[3], [3]], shape = [3, 3], type = oper, isHerm = True
Qobj data =
[[ 0.+0.j          0.+0.70710678j  0.+0.j        ]
 [ 0.-0.70710678j  0.+0.j          0.+0.70710678j]
 [ 0.+0.j          0.-0.70710678j  0.+0.j        ]]
 Quantum object: dims = [[3], [3]], shape = [3, 3], type = oper, isHerm = True
Qobj data =
[[ 1.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0. -1.]]]
```

num (*N*)

Quantum object for number operator.

Parameters **N** : int

The dimension of the Hilbert space.

Returns **oper: qobj** :

Qobj for number operator.

Examples

```
>>> num(4)
Quantum object: dims = [[4], [4]], shape = [4, 4], type = oper, isHerm = True
Qobj data =
[[0 0 0 0]
 [0 1 0 0]
 [0 0 2 0]
 [0 0 0 3]]
```

qeye(*N*)

Identity operator

Parameters **N** : int

Dimension of Hilbert space.

Returns **oper** : qobj

Identity operator Qobj.

Examples

```
>>> qeye(3)
Quantum object: dims = [[3], [3]], shape = [3, 3], type = oper, isHerm = True
Qobj data =
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

identity(*N*)

Identity operator. Alternative name to `qeye`.

Parameters **N** : int

Dimension of Hilbert space.

Returns **oper** : qobj

Identity operator Qobj.

qutrit_ops()

Operators for a three level system (qutrit).

Returns **opers: array** :

array of qutrit operators.

sigmam()

Annihilation operator for Pauli spins.

Examples

```
>>> sigmam()
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isHerm = False
Qobj data =
[[ 0.  0.]
 [ 1.  0.]]
```

sigmap()

Creation operator for Pauli spins.

Examples

```
>>> sigmam()
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isHerm = False
Qobj data =
[[ 0.  1.]
 [ 0.  0.]]
```

sigmax()
Pauli spin 1/2 sigma-x operator

Examples

```
>>> sigmax()
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isHerm = False
Qobj data =
[[ 0.  1.]
 [ 1.  0.]]
```

sigmay()
Pauli spin 1/2 sigma-y operator.

Examples

```
>>> sigmay()
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isHerm = True
Qobj data =
[[ 0.+0.j  0.-1.j]
 [ 0.+1.j  0.+0.j]]
```

sigmaz()
Pauli spin 1/2 sigma-z operator.

Examples

```
>>> sigmaz()
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isHerm = True
Qobj data =
[[ 1.  0.]
 [ 0. -1.]]
```

squeez(*N*, *sp*)
Single-mode Squeezing operator.

Parameters **N** : int

Dimension of hilbert space.

sp : float/complex

Squeezing parameter.

Returns **oper** : `qutip.qobj.Qobj`

Squeezing operator.

Examples

```
>>> squeez(4,0.25)
Quantum object: dims = [[4], [4]], shape = [4, 4], type = oper, isHerm = False
Qobj data =
[[ 0.98441565+0.j  0.00000000+0.j  0.17585742+0.j  0.00000000+0.j]
 [ 0.00000000+0.j  0.95349007+0.j  0.00000000+0.j  0.30142443+0.j]
 [-0.17585742+0.j  0.00000000+0.j  0.98441565+0.j  0.00000000+0.j]
 [ 0.00000000+0.j -0.30142443+0.j  0.00000000+0.j  0.95349007+0.j]]
```

Important: There is no ending 'e' for the squeezing operator!

---

squeezing(*a1*, *a2*, *z*)

Generalized squeezing operator.

$$S(z) = \exp\left(\frac{1}{2}\left(z^* a_1 a_2 - z a_1^\dagger a_2^\dagger\right)\right)$$

Parameters **a1** : `qutip.qobj.Qobj`

Operator 1.

a2 : `qutip.qobj.Qobj`

Operator 2.

z : float/complex

Squeezing parameter.

Returns **oper** : `qutip.qobj.Qobj`

Squeezing operator.

Liouvillian

liouvillian(*H*, *c_op_list*)

Assembles the Liouvillian superoperator from a Hamiltonian and a `list` of collapse operators.

Parameters **H** : qobj

System Hamiltonian.

c_op_list : array_like

A `list` or `array` of collpase operators.

Returns **L** : qobj

Louvillian superoperator.

spost(*A*)

Superoperator formed from post-multiplication by operator A

Parameters **A** : qobj

Quantum operator for post multiplication.

Returns **super** : qobj

Superoperator formed from input qauntum object.

spre(*A*)

Superoperator formed from pre-multiplication by operator A.

Parameters **A** : qobj

Quantum operator for pre-multiplication.

Returns **super :qobj** :

Superoperator formed from input quantum object.

Tensor

Module for the creation of composite quantum objects via the tensor product.

tensor(**args*)

Calculates the tensor product of input operators.

Parameters **args** : array_like

`list` or `array` of quantum objects for tensor product.

Returns **obj** : qobj

A composite quantum object.

Examples

```
>>> tensor([sigmax(), sigmax()])
Quantum object: dims = [[2, 2], [2, 2]], shape = [4, 4], type = oper, isHerm = True
Qobj data =
[[ 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]]
```

Expectation Values

expect(*oper*, *state*)

Calculates the expectation value for operator and state(s).

Parameters **oper** : qobj

Operator for expectation value.

state : qobj/list

A single or *list* of quantum states or density matricies.

Returns **expt** : float

Expectation value. `real` if *oper* is Hermitian, `complex` otherwise.

Examples

```
>>> expect(num(4),basis(4,3))
3
```

Partial transpose

partial_transpose(*rho*, *mask*, *method='dense'*)

Return the partial transpose of a Qobj instance *rho*, where *mask* is an array/list with length that equals the number of components of *rho* (that is, the length of *rho.dims[0]*), and the values in *mask* indicates whether or not the corresponding subsystem is to be transposed. The elements in *mask* can be boolean or integers *0* or *1*, where *True/1* indicates that the corresponding subsystem should be tranposed.

Parameters **rho** : `qutip.qobj`

A density matrix.

mask : *list / array*

A mask that selects which subsystems should be transposed.

method : str

choice of method, *dense* or *sparse*. The default method is *dense*. The *sparse* implementation can be faster for large and sparse systems (hundreds of quantum states).

Returns **rho_pr: :class:'qutip.qobj'** :

A density matrix with the selected subsystems transposed.

Pseudoprobability Functions

qfunc(*state*, *xvec*, *yvec*, *g=1.4142135623730951*)

Q-function of a given state vector or density matrix at points *xvec + i * yvec*.

Parameters **state** : qobj

A state vector or density matrix.

xvec : array_like

x-coordinates at which to calculate the Wigner function.

yvec : array_like

y-coordinates at which to calculate the Wigner function.

g : float

Scaling factor for *a = 0.5 * g * (x + iy)*, default *g = sqrt(2)*.

Returns **Q** : array

Values representing the Q-function calculated over the specified range [xvec,yvec].

wigner(*psi*, *xvec*, *yvec*, *g=1.4142135623730951*, *method='iterative'*, *parfor=False*)

Wigner function for a state vector or density matrix at points *xvec + i * yvec*.

Parameters **state** : qobj

A state vector or density matrix.

xvec : array_like

x-coordinates at which to calculate the Wigner function.

yvec : array_like

y-coordinates at which to calculate the Wigner function.

g : float

Scaling factor for *a = 0.5 * g * (x + iy)*, default *g = sqrt(2)*.

method : string {'iterative', 'laguerre'}

Select method 'iterative' or 'laguerre', where 'iterative' use a iterative method to evaluate the Wigner functions for density matrices $|m><n|$, while 'laguerre' uses the Laguerre polynomials in scipy for the same task. The 'iterative' method is default, and in general recommended, but the 'laguerre' method is more efficient for very sparse density matrices (e.g., superpositions of Fock states in a large Hilbert space).

parfor : bool {False, True}

Flag for calculating the Laguerre polynomial based Wigner function method='laguerre' in parallel using the parfor function.

Returns **W** : array

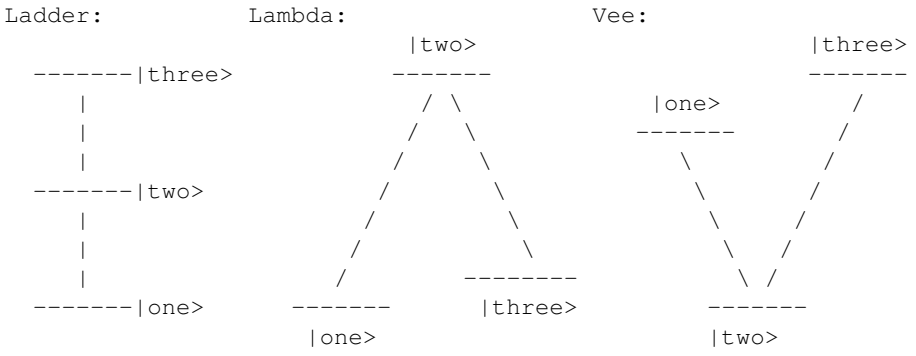
Values representing the Wigner function calculated over the specified range [xvec,yvec].

References

Ulf Leonhardt, Measuring the Quantum State of Light, (Cambridge University Press, 1997)

Three-level atoms

This module provides functions that are useful for simulating the three level atom with QuTiP. A three level atom (qutrit) has three states, which are linked by dipole transitions so that 1 <-> 2 <-> 3. Depending on there relative energies they are in the ladder, lambda or vee configuration. The structure of the relevant operators is the same for any of the three configurations:



References

The naming of qutip operators follows the convention in [1] .

Notes

Contributed by Markus Baden, Oct. 07, 2011

three_level_basis()
Basis states for a three level atom.

Returns **states** : array

array of three level atom basis vectors.

three_level_ops()
Operators for a three level system (qutrit)

Returns **ops** : array

array of three level operators.

6.2.2 Dynamics and time-evolution

Master Equation

This module provides solvers for the Lindblad master equation and von Neumann equation.

[1] Shore, B. W., "The Theory of Coherent Atomic Excitation", Wiley, 1990.

mesolve(*H*, *rho0*, *tlist*, *c_ops*, *expt_ops*, *args={}*, *options=None*)

Master equation evolution of a density matrix for a given Hamiltonian.

Evolve the state vector or density matrix (*rho0*) using a given Hamiltonian (*H*) and an [optional] set of collapse operators (*c_op_list*), by integrating the set of ordinary differential equations that define the system. In the absense of collase operators the system is evolved according to the unitary evolution of the Hamiltonian.

The output is either the state vector at arbitrary points in time (*tlist*), or the expectation values of the supplied operators (*expt_ops*). If expt_ops is a callback function, it is invoked for each time in *tlist* with time and the state as arguments, and the function does not use any return values.

Time-dependent operators

For problems with time-dependent problems *H* and *c_ops* can be callback functions that takes two arguments, time and *args*, and returns the Hamiltonian or Liuovillian for the system at that point in time (*callback format*).

Alternatively, *H* and *c_ops* can be a specified in a nested-list format where each element in the list is a list of length 2, containing an operator (`qutip.qobj`) at the first element and where the second element is either a string (*list string format*) or a callback function (*list callback format*) that evaluates to the time-dependent coefficient for the corresponding operator.

Examples

H = [[H0, 'sin(w*t)'], [H1, 'sin(2*w*t)']]

H = [[H0, f0_t], [H1, f1_t]]

where f0_t and f1_t are python functions with signature f_t(t, args).

In the *list string format* and *list callback format*, the string expression and the callback function must evaluate to a real or complex number (coefficient for the corresponding operator).

In all cases of time-dependent operators, *args* is a dictionary of parameters that is used when evaluating operators. It is passed to the callback functions as second argument

---

Note: If an element in the list-specification of the Hamiltonian or the list of collapse operators are in super-operator for it will be added to the total Liouvillian of the problem with out further transformation. This allows for using mesolve for solving master equations that are not on standard Lindblad form.

Note: On using callback function: mesolve transforms all `qutip.qobj` objects to sparse matrices before handing the problem to the integrator function. In order for your callback function to work correctly, pass all `qutip.qobj` objects that are used in constructing the Hamiltonian via args. odesolve will check for `qutip.qobj` in *args* and handle the conversion to sparse matrices. All other `qutip.qobj` objects that are not passed via *args* will be passed on to the integrator to scipy who will raise an NotImplemented exception.

Parameters **H** : `qutip.qobj`

system Hamiltonian, or a callback function for time-dependent Hamiltonians.

rho0 : `qutip.qobj`

initial density matrix or state vector (ket).

tlist : *list* / *array*

list of times for *t*.

c_ops : list of `qutip.qobj`

single collapse operator, or list of collapse operators.

expt_ops : list of `qutip.qobj` / callback function single

single operator or list of operators for which to evaluate expectation values.

args : *dictionary*

dictionary of parameters for time-dependent Hamiltonians and collapse operators.

options : `qutip.Qdeoptions`

with options for the ODE solver.

Returns **output: :class:'qutip.odedata'** :

An instance of the class `qutip.odedata`, which contains either an *array* of expectation values for the times specified by *tlist*, or an *array* or state vectors or density matrices corresponding to the times in *tlist* [if *expt_ops* is an empty list], or nothing if a callback function was given inplace of operators for which to calculate the expectation values.

odesolve(*H*, *rho0*, *tlist*, *c_op_list*, *expt_ops*, *args=None*, *options=None*)

Master equation evolution of a density matrix for a given Hamiltonian.

Evolution of a state vector or density matrix (*rho0*) for a given Hamiltonian (*H*) and set of collapse operators (*c_op_list*), by integrating the set of ordinary differential equations that define the system. The output is either the state vector at arbitrary points in time (*tlist*), or the expectation values of the supplied operators (*expt_ops*).

For problems with time-dependent Hamiltonians, *H* can be a callback function that takes two arguments, time and *args*, and returns the Hamiltonian at that point in time. *args* is a list of parameters that is passed to the callback function *H* (only used for time-dependent Hamiltonians).

Parameters **H** : `qutip.qobj`

system Hamiltonian, or a callback function for time-dependent Hamiltonians.

rho0 : `qutip.qobj`

initial density matrix or state vector (ket).

tlist : *list* / *array*

list of times for t.

c_op_list : list of `qutip.qobj`

list of collapse operators.

expt_ops : list of `qutip.qobj` / callback function

list of operators for which to evaluate expectation values.

args : *dictionary*

dictionary of parameters for time-dependent Hamiltonians and collapse operators.

options : `qutip.Qdeoptions`

with options for the ODE solver.

Returns **output :array** :

Expectation values of wavefunctions/density matrices :

for the times specified by 'tlist'. :

Notes

On using callback function: odesolve transforms all `qutip.qobj` objects to sparse matrices before handing the problem to the integrator function. In order for your callback function to work correctly, pass all `qutip.qobj` objects that are used in constructing the Hamiltonian via args. odesolve will check for `qutip.qobj` in *args* and handle the conversion to sparse matrices. All other `qutip.qobj` objects that

are not passed via *args* will be passed on to the integrator to scipy who will raise an NotImplemented exception.

Deprecated in QuTiP 2.0.0. Use `mesolve` instead.

Monte Carlo Evolution

mcsolve(*H*, *psi0*, *tlist*, *c_ops*, *e_ops*, *ntraj=None*, *args={}*, *options=<qutip.odeoptions.Odeoptions instance at 0x10e1c14d0>*)

Monte-Carlo evolution of a state vector $|\psi\rangle$ for a given Hamiltonian and sets of collapse operators, and possibly, operators for calculating expectation values. Options for the underlying ODE solver are given by the Odeoptions class.

mcsolve supports time-dependent Hamiltonians and collapse operators using either Python functions of strings to represent time-dependent coefficients. Note that, the system Hamiltonian MUST have at least one constant term.

As an example of a time-dependent problem, consider a Hamiltonian with two terms `H0` and `H1`, where `H1` is time-dependent with coefficient `sin(w*t)`, and collapse operators `C0` and `C1`, where `C1` is time-dependent with coeffcient `exp(-a*t)`. Here, w and a are constant arguments with values `W` and `A`.

Using the Python function time-dependent format requires two Python functions, one for each collapse coefficient. Therefore, this problem could be expressed as:

```
def H1_coeff(t,args):
    return sin(args['w']*t)

def C1_coeff(t,args):
    return exp(-args['a']*t)

H=[H0,[H1,H1_coeff]]

c_op_list=[C0,[C1,C1_coeff]]

args={'a':A,'w':W}
```

or in String (Cython) format we could write:

```
H=[H0,[H1,'sin(w*t)']]

c_op_list=[C0,[C1,'exp(-a*t)']]

args={'a':A,'w':W}
```

Constant terms are preferably placed first in the Hamiltonian and collapse operator lists.

Parameters **H** : qobj

System Hamiltonian.

psi0 : qobj

Initial state vector

tlist : array_like

Times at which results are recorded.

ntraj : int

Number of trajectories to run.

c_ops : array_like

single collapse operator or `list` or `array` of collapse operators.

e_ops : array_like

single operator or `list` or `array` of operators for calculating expectation values.

args : dict

Arguments for time-dependent Hamiltonian and collapse operator terms.

options : Odeoptions

Instance of ODE solver options.

Returns **results** : Odedata

Object storing all results from simulation.

mcsolve_f90(*H*, *psi0*, *tlist*, *c_ops*, *e_ops*, *ntraj=None*, *options=<qutip.odeoptions.Odeoptions instance at 0x10e1c15a8>*, *sparse_dms=True*, *serial=False*, *ptrace_sel=[]*, *calc_entropy=False*)

Monte-Carlo wave function solver with fortran 90 backend. Usage is identical to qutip.mcsolve, for problems without explicit time-dependence, and with some optional input:

Parameters **H** : qobj

System Hamiltonian.

psi0 : qobj

Initial state vector

tlist : array_like

Times at which results are recorded.

ntraj : int

Number of trajectories to run.

c_ops : array_like

`list` or `array` of collapse operators.

e_ops : array_like

`list` or `array` of operators for calculating expectation values.

options : Odeoptions

Instance of ODE solver options.

sparse_dms : boolean

If averaged density matrices are returned, they will be stored as sparse (Compressed Row Format) matrices during computation if sparse_dms = True (default), and dense matrices otherwise. Dense matrices might be preferable for smaller systems.

serial : boolean

If True (default is False) the solver will not make use of the multiprocessing module, and simply run in serial.

ptrace_sel: list :

This optional argument specifies a list of components to keep when returning a partially traced density matrix. This can be convenient for large systems where memory becomes a problem, but you are only interested in parts of the density matrix.

calc_entropy : boolean

If ptrace_sel is specified, calc_entropy=True will have the solver return the averaged entropy over trajectories in results.entropy. This can be interpreted as a measure of entanglement. See Phys. Rev. Lett. 93, 120408 (2004), Phys. Rev. A 86, 022310 (2012).

Returns **results** : Odedata

Object storing all results from simulation.

Schrödinger Equation

This module provides solvers for the unitary Schrodinger equation.

sesolve(*H*, *rho0*, *tlist*, *expt_ops*, *args={}*, *options=None*)

Schrodinger equation evolution of a state vector for a given Hamiltonian.

Evolve the state vector or density matrix (*rho0*) using a given Hamiltonian (*H*), by integrating the set of ordinary differential equations that define the system.

The output is either the state vector at arbitrary points in time (*tlist*), or the expectation values of the supplied operators (*expt_ops*). If expt_ops is a callback function, it is invoked for each time in *tlist* with time and the state as arguments, and the function does not use any return values.

Parameters **H** : `qutip.qobj`

system Hamiltonian, or a callback function for time-dependent Hamiltonians.

rho0 : `qutip.qobj`

initial density matrix or state vector (ket).

tlist : *list / array*

list of times for *t*.

expt_ops : list of `qutip.qobj` / callback function single

single operator or list of operators for which to evaluate expectation values.

args : *dictionary*

dictionary of parameters for time-dependent Hamiltonians and collapse operators.

options : `qutip.Qdeoptions`

with options for the ODE solver.

Returns **output: :class:‘qutip.odedata‘** :

An instance of the class `qutip.odedata`, which contains either an *array* of expectation values for the times specified by *tlist*, or an *array* or state vectors or density matrices corresponding to the times in *tlist* [if *expt_ops* is an empty list], or nothing if a callback function was given inplace of operators for which to calculate the expectation values.

Bloch-Redfield Master Equation

brmesolve(*H*, *psi0*, *tlist*, *c_ops*, *e_ops=[]*, *spectra_cb=[]*, *args={}*, *options=<qutip.odeoptions.Odeoptions instance at 0x10e6137e8>*)

Solve the dynamics for the system using the Bloch-Redfeild master equation.

Note: This solver does not currently support time-dependent Hamiltonian or collapse operators.

Parameters **H** : `qutip.qobj`

System Hamiltonian.

rho0 / psi0: :class:‘qutip.qobj‘ :

Initial density matrix or state vector (ket).

tlist : *list / array*

List of times for t.

c_ops : list of `qutip.qobj`

List of collapse operators.

expt_ops : list of `qutip.qobj` / callback function

List of operators for which to evaluate expectation values.

args : *dictionary*

Dictionary of parameters for time-dependent Hamiltonians and collapse operators.

options : `qutip.Qdeoptions`

Options for the ODE solver.

Returns **output: :class:'qutip.odedata'** :

An instance of the class `qutip.odedata`, which contains either an *array* of expectation values for the times specified by *tlist*.

bloch_redfield_tensor(*H*, *c_ops*, *spectra_cb*, *use_secular=True*)

Calculate the Bloch-Redfield tensor for a system given a set of operators and corresponding spectral functions that describes the system's coupling to its environment.

Parameters **H** : `qutip.qobj`

System Hamiltonian.

c_ops : list of `qutip.qobj`

List of collapse operators.

spectra_cb : list of callback functions

List of callback functions that evaluate the noise power spectrum at a given frequency.

use_secular : bool

Flag (True of False) that indicates if the secular approximation should be used.

Returns **R, kets: :class:'qutip.qobj', list of :class:'qutip.qobj'** :

R is the Bloch-Redfield tensor and kets is a list eigenstates of the Hamiltonian.

bloch_redfield_solve(*R*, *ekets*, *rho0*, *tlist*, *e_ops=[]*, *options=None*)

Evolve the ODEs defined by Bloch-Redfield master equation. The Bloch-Redfield tensor can be calculated by the function `bloch_redfield_tensor`.

Parameters **R** : `qutip.qobj`

Bloch-Redfield tensor.

ekets : array of `qutip.qobj`

Array of kets that make up a basis tranformation for the eigenbasis.

rho0 : `qutip.qobj`

Initial density matrix.

tlist : *list* / *array*

List of times for t.

e_ops : list of `qutip.qobj` / callback function

List of operators for which to evaluate expectation values.

options : `qutip.Qdeoptions`

Options for the ODE solver.

Returns **output: :class:'qutip.odedata'** :

An instance of the class `qutip.odedata`, which contains either an *array* of expectation values for the times specified by *tlist*.

Floquet States and Floquet-Markov Master Equation

fmmesolve(*H*, *rho0*, *tlist*, *c_ops*, *e_ops=[]*, *spectra_cb=[]*, *T=None*, *args={}*, *options=<qutip.odeoptions.Odeoptions instance at 0x10e613680>*, *floquet_basis=True*, *kmax=5*)

Solve the dynamics for the system using the Floquet-Markov master equation.

Note: This solver currently does not support multiple collapse operators.

Parameters **H** : `qutip.qobj`

system Hamiltonian.

rho0 / psi0 : `qutip.qobj`

initial density matrix or state vector (ket).

tlist : *list / array*

list of times for t.

c_ops : list of `qutip.qobj`

list of collapse operators.

e_ops : list of `qutip.qobj` / callback function

list of operators for which to evaluate expectation values.

spectra_cb : list callback functions

List of callback functions that compute the noise power spectrum as a function of frequency for the collapse operators in *c_ops*.

T : float

The period of the time-dependence of the hamiltonian. The default value 'None' indicates that the 'tlist' spans a single period of the driving.

args : *dictionary*

dictionary of parameters for time-dependent Hamiltonians and collapse operators.

This dictionary should also contain an entry 'w_th', which is the temperature of the environment (if finite) in the energy/frequency units of the Hamiltonian. For example, if the Hamiltonian written in units of 2pi GHz, and the temperature is given in K, use the following conversion

```
>>> temperature = 25e-3 # unit K
>>> h = 6.626e-34
>>> kB = 1.38e-23
>>> args['w_th'] = temperature * (kB / h) * 2 * pi * 1e-9
```

options : `qutip.odeoptions`

options for the ODE solver.

k_max : int

The truncation of the number of sidebands (default 5).

Returns **output** : `qutip.odedata`

An instance of the class `qutip.odedata`, which contains either an *array* of expectation values for the times specified by *tlist*.

floquet_modes (*H*, *T*, *args=None*, *sort=False*)

Calculate the initial Floquet modes Phi_alpha(0) for a driven system with period T.

Returns a list of `qutip.qobj` instances representing the Floquet modes and a list of corresponding quasienergies, sorted by increasing quasienergy in the interval [-pi/T, pi/T]. The optional parameter *sort* decides if the output is to be sorted in increasing quasienergies or not.

Parameters **H** : `qutip.qobj`

system Hamiltonian, time-dependent with period *T*

args : dictionary

dictionary with variables required to evaluate H

T : float

The period of the time-dependence of the hamiltonian. The default value 'None' indicates that the 'tlist' spans a single period of the driving.

Returns **output** : list of kets, list of quasi energies

Two lists: the Floquet modes as kets and the quasi energies.

floquet_modes_t (*f_modes_0*, *f_energies*, *t*, *H*, *T*, *args=None*)

Calculate the Floquet modes at times tlist Phi_alpha(tlist) propagting the initial Floquet modes Phi_alpha(0)

Parameters **f_modes_0** : list of `qutip.qobj` (kets)

Floquet modes at t

f_energies : list

Floquet energies.

t : float

The time at which to evaluate the floquet modes.

H : `qutip.qobj`

system Hamiltonian, time-dependent with period *T*

args : dictionary

dictionary with variables required to evaluate H

T : float

The period of the time-dependence of the hamiltonian.

Returns **output** : list of kets

The Floquet modes as kets at time t

floquet_modes_table (*f_modes_0*, *f_energies*, *tlist*, *H*, *T*, *args=None*)

Pre-calculate the Floquet modes for a range of times spanning the floquet period. Can later be used as a table to look up the floquet modes for any time.

Parameters **f_modes_0** : list of `qutip.qobj` (kets)

Floquet modes at t

f_energies : list

Floquet energies.

tlist : array

The list of times at which to evaluate the floquet modes.

H : `qutip.qobj`

system Hamiltonian, time-dependent with period *T*

T : float

The period of the time-dependence of the hamiltonian.

args : dictionary

dictionary with variables required to evaluate H

Returns **output** : nested list

A nested list of Floquet modes as kets for each time in *tlist*

floquet_modes_t_lookup (*f_modes_table_t*, *t*, *T*)

Lookup the floquet mode at time t in the pre-calculated table of floquet modes in the first period of the time-dependence.

Parameters **f_modes_table_t** : nested list of `qutip.qobj` (kets)

A lookup-table of Floquet modes at times precalculated by `qutip.floquet.floquet_modes_table`.

t : float

The time for which to evaluate the Floquet modes.

T : float

The period of the time-dependence of the hamiltonian.

Returns **output** : nested list

A list of Floquet modes as kets for the time that most closely matching the time *t* in the supplied table of Floquet modes.

floquet_states_t (*f_modes_0*, *f_energies*, *t*, *H*, *T*, *args=None*)

Evaluate the floquet states at time t given the initial Floquet modes.

Parameters **f_modes_t** : list of `qutip.qobj` (kets)

A list of initial Floquet modes (for time $t = 0$).

f_energies : array

The Floquet energies.

t : float

The time for which to evaluate the Floquet states.

H : `qutip.qobj`

System Hamiltonian, time-dependent with period *T*.

T : float

The period of the time-dependence of the hamiltonian.

args : dictionary

Dictionary with variables required to evaluate H.

Returns **output** : list

A list of Floquet states for the time t.

floquet_wavefunction_t (*f_modes_0*, *f_energies*, *f_coeff*, *t*, *H*, *T*, *args=None*)

Evaluate the wavefunction for a time t using the Floquet state decompositon, given the initial Floquet modes.

Parameters **f_modes_t** : list of `qutip.qobj` (kets)

A list of initial Floquet modes (for time $t = 0$).

f_energies : array

The Floquet energies.

f_coeff : array

The coefficients for Floquet decomposition of the initial wavefunction.

t : float

The time for which to evaluate the Floquet states.

H : `qutip.qobj`

System Hamiltonian, time-dependent with period *T*.

T : float

The period of the time-dependence of the hamiltonian.

args : dictionary

Dictionary with variables required to evaluate H.

Returns **output** : `qutip.qobj`

The wavefunction for the time t.

floquet_state_decomposition (*f_states*, *f_energies*, *psi*)

Decompose the wavefunction *psi* (typically an initial state) in terms of the Floquet states, $\psi = \sum_\alpha c_\alpha \psi_\alpha(0)$.

Parameters **f_states** : list of `qutip.qobj` (kets)

A list of Floquet modes.

f_energies : array

The Floquet energies.

psi : `qutip.qobj`

The wavefunction to decompose in the Floquet state basis.

Returns **output** : array

The coefficients c_α in the Floquet state decomposition.

fsesolve (*H*, *psi0*, *tlist*, *e_ops=[ ]*, *T=None*, *args={}*, *Tsteps=100*)

Solve the Schrodinger equation using the Floquet formalism.

Parameters **H** : `qutip.qobj.Qobj`

System Hamiltonian, time-dependent with period *T*.

psi0 : `qutip.qobj`

Initial state vector (ket).

tlist : *list* / *array*

list of times for t.

e_ops : list of `qutip.qobj` / callback function

list of operators for which to evaluate expectation values. If this list is empty, the state vectors for each time in *tlist* will be returned instead of expectation values.

T : float

The period of the time-dependence of the hamiltonian.

args : dictionary

Dictionary with variables required to evaluate H.

Tsteps : integer

The number of time steps in one driving period for which to precalculate the Floquet modes. *Tsteps* should be an even number.

Returns **output** : `qutip.odedata.Odedata`

An instance of the class `qutip.odedata.Odedata`, which contains either an *array* of expectation values or an array of state vectors, for the times specified by *tlist*.

Correlation Functions

correlation(*H*, *rho0*, *tlist*, *taulist*, *c_ops*, *a_op*, *b_op*, *solver='me'*, *reverse=False*, *options=<qutip.odeoptions.Odeoptions instance at 0x10e60ec68>*)

Calculate a two-operator two-time correlation function on the form $\langle A(t+\tau)B(t)\rangle$ or $\langle A(t)B(t+\tau)\rangle$ (if *reverse=True*), using the quantum regression theorem and the evolution solver indicated by the *solver* parameter.

Parameters **H** : `qutip.qobj.Qobj`

system Hamiltonian.

rho0 : `qutip.qobj.Qobj`

Initial state density matrix (or state vector). If 'rho0' is 'None', then the steady state will be used as initial state.

tlist : *list / array*

list of times for t.

taulist : *list / array*

list of times for τ.

c_ops : list of `qutip.qobj.Qobj`

list of collapse operators.

a_op : `qutip.qobj`

operator A.

b_op : `qutip.qobj`

operator B.

solver : str

choice of solver (*me* for master-equation, *es* for exponential series and *mc* for Monte-carlo)

Returns **corr_mat: *array*** :

An 2-dimensional *array* (matrix) of correlation values for the times specified by *tlist* (first index) and *taulist* (second index). If *tlist* is *None*, then a 1-dimensional *array* of correlation values is returned instead.

correlation_ss(*H*, *taulist*, *c_ops*, *a_op*, *b_op*, *rho0=None*, *solver='me'*, *reverse=False*, *options=<qutip.odeoptions.Odeoptions instance at 0x10e60ec20>*)

Calculate a two-operator two-time correlation function $\langle A(\tau)B(0)\rangle$ or $\langle A(0)B(\tau)\rangle$ (if *reverse=True*), using the quantum regression theorem and the evolution solver indicated by the *solver* parameter.

Parameters **H** : `qutip.qobj.Qobj`

system Hamiltonian.

rho0 : `qutip.qobj.Qobj`

Initial state density matrix (or state vector). If 'rho0' is 'None', then the steady state will be used as initial state.

taulist : *list / array*

list of times for τ.

c_ops : list of `qutip.qobj.Qobj`

list of collapse operators.

a_op : `qutip.qobj.Qobj`

operator A.

b_op : `qutip.qobj.Qobj`

operator B.

reverse : bool

If *True*, calculate $\langle A(0)B(\tau)\rangle$ instead of $\langle A(\tau)B(0)\rangle$.

solver : str

choice of solver (*me* for master-equation, *es* for exponential series and *mc* for Monte-carlo)

Returns **corr_vec: *array*** :

An *array* of correlation values for the times specified by *tlist*

correlation_2op_1t(*H*, *rho0*, *taulist*, *c_ops*, *a_op*, *b_op*, *solver='me'*, *reverse=False*, *options=<qutip.odeoptions.Odeoptions instance at 0x10e60e9e0>*)

Calculate a two-operator two-time correlation function $\langle A(\tau)B(0)\rangle$ or $\langle A(0)B(\tau)\rangle$ (if *reverse=True*), using the quantum regression theorem and the evolution solver indicated by the *solver* parameter.

Parameters **H** : `qutip.qobj.Qobj`

system Hamiltonian.

rho0 : `qutip.qobj.Qobj`

Initial state density matrix (or state vector). If *rho0* is *None*, then the steady state will be used as initial state.

taulist : *list / array*

list of times for τ.

c_ops : list of `qutip.qobj.Qobj`

list of collapse operators.

a_op : `qutip.qobj.Qobj`

operator A.

b_op : `qutip.qobj.Qobj`

operator B.

reverse : bool

If *True*, calculate $\langle A(0)B(\tau)\rangle$ instead of $\langle A(\tau)B(0)\rangle$.

solver : str

choice of solver (*me* for master-equation, *es* for exponential series and *mc* for Monte-carlo)

Returns **corr_vec: *array*** :

An *array* of correlation values for the times specified by *taulist*

correlation_2op_2t(*H*, *rho0*, *tlist*, *taulist*, *c_ops*, *a_op*, *b_op*, *solver='me'*, *reverse=False*, *options=<qutip.odeoptions.Odeoptions instance at 0x10e60ea28>*)

Calculate a two-operator two-time correlation function on the form $\langle A(t+\tau)B(t)\rangle$ or $\langle A(t)B(t+\tau)\rangle$ (if *reverse=True*), using the quantum regression theorem and the evolution solver indicated by the *solver* parameter.

Parameters **H** : `qutip.qobj.Qobj`

system Hamiltonian.

rho0 : `qutip.qobj.Qobj`

Initial state density matrix $\rho(t_0)$ (or state vector). If 'rho0' is 'None', then the steady state will be used as initial state.

tlist : *list / array*

list of times for t.

taulist : *list / array*

list of times for τ.

c_ops : list of `qutip.qobj.Qobj`

list of collapse operators.

a_op : `qutip.qobj.Qobj`

operator A.

b_op : `qutip.qobj.Qobj`

operator B.

solver : str

choice of solver (*me* for master-equation, *es* for exponential series and *mc* for Monte-carlo)

reverse : bool

If *True*, calculate $\langle A(t)B(t+\tau)\rangle$ instead of $\langle A(t+\tau)B(t)\rangle$.

Returns **corr_mat: *array*** :

An 2-dimensional *array* (matrix) of correlation values for the times specified by *tlist* (first index) and *taulist* (second index). If *tlist* is *None*, then a 1-dimensional *array* of correlation values is returned instead.

correlation_4op_1t(*H*, *rho0*, *taulist*, *c_ops*, *a_op*, *b_op*, *c_op*, *d_op*, *solver='me'*, *options=<qutip.odeoptions.Odeoptions instance at 0x10e60ea70>*)

Calculate the four-operator two-time correlation function on the from $\langle A(0)B(\tau)C(\tau)D(0)\rangle$ using the quantum regression theorem and the solver indicated by the 'solver' parameter.

Parameters **H** : `qutip.qobj.Qobj`

system Hamiltonian.

rho0 : `qutip.qobj.Qobj`

Initial state density matrix (or state vector). If 'rho0' is 'None', then the steady state will be used as initial state.

taulist : *list / array*

list of times for τ.

c_ops : list of `qutip.qobj.Qobj`

list of collapse operators.

a_op : `qutip.qobj.Qobj`

operator A.

b_op : `qutip.qobj.Qobj`

operator B.

c_op : `qutip.qobj.Qobj`

operator C.

d_op : `qutip.qobj.Qobj`

operator D.

solver : str

choice of solver (currently only *me* for master-equation)

Returns **corr_vec: *array*** :

An *array* of correlation values for the times specified by *taulist*

References

See, Gardiner, Quantum Noise, Section 5.2.1.

correlation_4op_2t(*H*, *rho0*, *tlist*, *taulist*, *c_ops*, *a_op*, *b_op*, *c_op*, *d_op*, *solver='me'*, *options=<qutip.odeoptions.Odeoptions instance at 0x10e60eab8>*)

Calculate the four-operator two-time correlation function on the from $\langle A(t)B(t+\tau)C(t+\tau)D(t)\rangle$ using the quantum regression theorem and the solver indicated by the 'solver' parameter.

Parameters **H** : `qutip.qobj.Qobj`

system Hamiltonian.

rho0 : `qutip.qobj.Qobj`

Initial state density matrix (or state vector). If 'rho0' is 'None', then the steady state will be used as initial state.

tlist : *list* / *array*

list of times for t.

taulist : *list* / *array*

list of times for τ.

c_ops : list of `qutip.qobj.Qobj`

list of collapse operators.

a_op : `qutip.qobj.Qobj`

operator A.

b_op : `qutip.qobj.Qobj`

operator B.

c_op : `qutip.qobj.Qobj`

operator C.

d_op : `qutip.qobj.Qobj`

operator D.

solver : str

choice of solver (currently only *me* for master-equation)

Returns **corr_mat: *array*** :

An 2-dimensional *array* (matrix) of correlation values for the times specified by *tlist* (first index) and *taulist* (second index). If *tlist* is *None*, then a 1-dimensional *array* of correlation values is returned instead.

References

See, Gardiner, Quantum Noise, Section 5.2.1.

spectrum_ss (*H*, *wlist*, *c_ops*, *a_op*, *b_op*)

Calculate the spectrum corresponding to a correlation function $\langle A(\tau)B(0)\rangle$, i.e., the Fourier transform of the correlation function:

$$S(\omega) = \int_{-\infty}^{\infty} \langle A(\tau)B(0)\rangle\, e^{-i\omega\tau} d\tau.$$

Parameters **H** : `qutip.qobj`

system Hamiltonian.

wlist : *list / array*

list of frequencies for ω.

c_ops : list of `qutip.qobj`

list of collapse operators.

a_op : `qutip.qobj`

operator A.

b_op : `qutip.qobj`

operator B.

Returns **spectrum: *array*** :

An *array* with spectrum $S(\omega)$ for the frequencies specified in *wlist*.

spectrum_correlation_fft (*tlist*, *y*)

Calculate the power spectrum corresponding to a two-time correlation function using FFT.

Parameters **tlist** : *list / array*

list/array of times t which the correlation function is given.

y : *list / array*

list/array of correlations corresponding to time delays t.

Returns **w, S** : *tuple*

Returns an array of angular frequencies 'w' and the corresponding one-sided power spectrum 'S(w)'.

coherence_function_g1 (*H*, *rho0*, *taulist*, *c_ops*, *a_op*, *solver='me'*, *options=<qutip.odeoptions.Odeoptions instance at 0x10e60eb00>*)

Calculate the first-order quantum coherence function:

$$g^{(1)}(\tau) = \frac{\langle a^\dagger(\tau)a(0)\rangle}{\sqrt{\langle a^\dagger(\tau)a(\tau)\rangle\langle a^\dagger(0)a(0)\rangle}}$$

Parameters **H** : `qutip.qobj.Qobj`

system Hamiltonian.

rho0 : `qutip.qobj.Qobj`

Initial state density matrix (or state vector). If 'rho0' is 'None', then the steady state will be used as initial state.

taulist : *list / array*

list of times for τ.

c_ops : list of `qutip.qobj.Qobj`

list of collapse operators.

a_op : `qutip.qobj.Qobj`

The annihilation operator of the mode.

solver : str

choice of solver ('me', 'mc', 'es')

Returns **g1, G2: tuble of *array*** :

The normalized and unnormalized first-order coherence function.

coherence_function_g2(*H*, *rho0*, *taulist*, *c_ops*, *a_op*, *solver='me'*, *options=<qutip.odeoptions.Odeoptions instance at 0x10e60eb90>*)

Calculate the second-order quantum coherence function:

$$g^{(2)}(\tau) = \frac{\langle a^\dagger(0)a^\dagger(\tau)a(\tau)a(0)\rangle}{\langle a^\dagger(\tau)a(\tau)\rangle\langle a^\dagger(0)a(0)\rangle}$$

Parameters **H** : `qutip.qobj.Qobj`

system Hamiltonian.

rho0 : `qutip.qobj.Qobj`

Initial state density matrix (or state vector). If 'rho0' is 'None', then the steady state will be used as initial state.

taulist : *list / array*

list of times for τ.

c_ops : list of `qutip.qobj.Qobj`

list of collapse operators.

a_op : `qutip.qobj.Qobj`

The annihilation operator of the mode.

solver : str

choice of solver (currently only 'me')

Returns **g2, G2: tuble of *array*** :

The normalized and unnormalized second-order coherence function.

Exponential Series

essolve(*H*, *rho0*, *tlist*, *c_op_list*, *expt_op_list*)

Evolution of a state vector or density matrix (*rho0*) for a given Hamiltonian (*H*) and set of collapse operators (*c_op_list*), by expressing the ODE as an exponential series. The output is either the state vector at arbitrary points in time (*tlist*), or the expectation values of the supplied operators (*expt_op_list*).

Parameters **H** : qobj/function_type

System Hamiltonian.

rho0 : `qutip.qobj`

Initial state density matrix.

tlist : list/array

`list` of times for t.

c_op_list : list of `qutip.qobj`

`list` of `qutip.qobj` collapse operators.

expt_op_list : list of `qutip.qobj`

`list` of `qutip.qobj` operators for which to evaluate expectation values.

Returns **expt_array** : array

Expectation values of wavefunctions/density matrices for the times specified in `tlist`.

.. note:: This solver does not support time-dependent Hamiltonians. :

ode2es(*L*, *rho0*)

Creates an exponential series that describes the time evolution for the initial density matrix (or state vector) *rho0*, given the Liouvillian (or Hamiltonian) *L*.

Parameters **L** : qobj

Liouvillian of the system.

rho0 : qobj

Initial state vector or density matrix.

Returns **eseries** : `qutip.eseries`

`eseries` represention of the system dynamics.

Steady-state Solvers

Module contains functions for iteratively solving for the steady state density matrix of an open qunatum system defind by a Louvillian.

steady(*L*, *maxiter=10*, *tol=1e-06*, *itertol=1e-05*, *method='solve'*, *use_umfpack=True*, *use_precond=False*)

Steady state for the evolution subject to the supplied Louvillian.

Parameters **L** : qobj

Liouvillian superoperator.

maxiter : int

Maximum number of iterations to perform, default = 100.

tol : float

Tolerance used for terminating solver solution, default = 1e-6.

itertol : float

Tolerance used for iterative Ax=b solver, default = 1e-5.

method : str

Method for solving linear equations. Direct solver 'solve' (default) or iterative biconjugate gradient method 'bicg'.

use_umfpack: bool {True, False} :

Use the UMFpack backend for the direct solver. If 'False', the solver uses the SuperLU backend. This option does not affect the 'bicg' method.

use_precond: bool {False, True} :

Use an incomplete sparse LU decomposition as a preconditioner for the stabilized bi-conjugate gradient 'bicg' method.

Returns **ket** : qobj

Ket vector for steady state.

Notes

Uses the inverse power method. See any Linear Algebra book with an iterative methods section. Using UMFpack may result in 'out of memory' errors for some Liouvillians.

steadystate (*H*, *c_op_list*, *maxiter=10*, *tol=1e-06*, *itertol=1e-05*, *method='solve'*, *use_umfpack=True*, *use_precond=False*)

Calculates the steady state for the evolution subject to the supplied Hamiltonian and list of collapse operators.

This function builds the Louvillian from the Hamiltonaian and calls the `qutip.steady.steady` function.

Parameters **H** : qobj

Hamiltonian operator.

c_op_list : list

A `list` of collapse operators.

maxiter : int

Maximum number of iterations to perform, default = 100.

tol : float

Tolerance used for terminating solver solution, default = 1e-6.

itertol : float

Tolerance used for iterative Ax=b solver, default = 1e-5.

method : str

Method for solving linear equations. Direct solver 'solve' (default) or iterative biconjugate gradient method 'bicg'.

use_umfpack: bool, default = True :

Use the UMFpack backend for the direct solver. If 'False', the solver uses the SuperLU backend. This option does not affect the 'bicg' method.

use_precond: bool, default = False :

Use an incomplete sparse LU decomposition as a preconditioner for the stabilized bi-conjugate gradient 'bicg' method.

Returns **ket** : qobj

Ket vector for steady state.

Notes

Uses the inverse power method. See any Linear Algebra book with an iterative methods section. Using UMFpack may result in 'out of memory' errors for some Liouvillians.

Propagators

propagator (*H*, *t*, *c_op_list*, *H_args=None*, *opt=None*)

Calculate the propagator U(t) for the density matrix or wave function such that $\psi(t) = U(t)\psi(0)$ or $\rho_{vec}(t) = U(t)\rho_{vec}(0)$ where $\rho_{\rm vec}$ is the vector representation of the density matrix.

Parameters **H** : qobj or list

Hamiltonian as a Qobj instance of a nested list of Qobjs and coefficients in the list-string or list-function format for time-dependent Hamiltonians (see description in `qutip.mesolve`).

t : float or array-like

Time or list of times for which to evaluate the propagator.

c_op_list : list

List of qobj collapse operators.

H_args : list/array/dictionary

Parameters to callback functions for time-dependent Hamiltonians.

Returns **a** : qobj

Instance representing the propagator $U(t)$.

propagator_steadystate(U)

Find the steady state for successive applications of the propagator U.

Parameters **U** : qobj

Operator representing the propagator.

Returns **a** : qobj

Instance representing the steady-state density matrix.

Continous variables

This module contains a collection functions for calculating continous variable quantities from fock-basis representation of the state of multi-mode fields.

correlation_matrix(*basis*, *rho=None*)

Given a basis set of operators $\{a\}_n$, calculate the correlation matrix:

$$C_{mn} = \langle a_m a_n \rangle$$

Parameters **basis** : list of `qutip.qobj.Qobj`

List of operators that defines the basis for the correlation matrix.

rho : `qutip.qobj.Qobj`

Density matrix for which to calculate the correlation matrix. If *rho* is *None*, then a matrix of correlation matrix operators is returned instead of expectation values of those operators.

Returns **corr_mat: *array*** :

A 2-dimensional *array* of correlation values or operators.

covariance_matrix(*basis*, *rho*, *symmetrized=True*)

Given a basis set of operators $\{a\}_n$, calculate the covariance matrix:

$$V_{mn} = \frac{1}{2}\langle a_m a_n + a_n a_m \rangle - \langle a_m \rangle \langle a_n \rangle$$

or, if of the optional argument *symmetrized=False*,

$$V_{mn} = \langle a_m a_n \rangle - \langle a_m \rangle \langle a_n \rangle$$

Parameters **basis** : list of `qutip.qobj.Qobj`

List of operators that defines the basis for the covariance matrix.

rho : `qutip.qobj.Qobj`

Density matrix for which to calculate the covariance matrix.

symmetrized : *bool*

Flag indicating whether the symmetrized (default) or non-symmetrized correlation matrix is to be calculated.

Returns **corr_mat: *array*** :

A 2-dimensional *array* of covariance values.

correlation_matrix_field(*a1*, *a2*, *rho=None*)

Calculate the correlation matrix for given field operators a_1 and a_2. If a density matrix is given the expectation values are calculated, otherwise a matrix with operators is returned.

Parameters **a1** : `qutip.qobj.Qobj`

Field operator for mode 1.

a2 : `qutip.qobj.Qobj`

Field operator for mode 2.

rho : `qutip.qobj.Qobj`

Density matrix for which to calculate the covariance matrix.

Returns **cov_mat: *array* of complex numbers or :class:‘qutip.qobj.Qobj‘** :

A 2-dimensional *array* of covariance values, or, if rho=0, a matrix of operators.

correlation_matrix_quadrature(*a1*, *a2*, *rho=None*)

Calculate the quadrature correlation matrix with given field operators a_1 and a_2. If a density matrix is given the expectation values are calculated, otherwise a matrix with operators is returned.

Parameters **a1** : `qutip.qobj.Qobj`

Field operator for mode 1.

a2 : `qutip.qobj.Qobj`

Field operator for mode 2.

rho : `qutip.qobj.Qobj`

Density matrix for which to calculate the covariance matrix.

Returns **corr_mat: *array* of complex numbers or :class:‘qutip.qobj.Qobj‘** :

A 2-dimensional *array* of covariance values for the field quadratures, or, if rho=0, a matrix of operators.

wigner_covariance_matrix(*a1=None*, *a2=None*, *R=None*, *rho=None*)

Calculate the wigner covariance matrix $V_{ij} = \frac{1}{2}(R_{ij} + R_{ji})$, given the quadrature correlation matrix $R_{ij} = \langle R_i R_j \rangle - \langle R_i \rangle \langle R_j \rangle$, where $R = (q_1, p_1, q_2, p_2)^T$ is the vector with quadrature operators for the two modes.

Alternatively, if *R = None*, and if annilation operators *a1* and *a2* for the two modes are supplied instead, the quadature correlation matrix is constructed from the annihilation operators before then the covariance matrix is calculated.

Parameters **a1** : `qutip.qobj.Qobj`

Field operator for mode 1.

a2 : `qutip.qobj.Qobj`

Field operator for mode 2.

R : *array*

The quadrature correlation matrix.

rho : `qutip.qobj.Qobj`

Density matrix for which to calculate the covariance matrix.

Returns **cov_mat: *array*** :

A 2-dimensional *array* of covariance values.

logarithmic_negativity(*V*)

Calculate the logarithmic negativity given the symmetrized covariance matrix, see `qutip.continous_variables.covariance_matrix`. Note that the two-mode field state that is described by *V* must be Gaussian for this function to applicable.

Parameters **V** : *2d array*

The covariance matrix.

Returns **N: *float*, the logarithmic negativity for the two-mode Guassian state** :

that is described by the the Wigner covariance matrix V. :

Quantum Process Tomography

qpt(*U*, *op_basis_list*)

Calculate the quantum process tomography chi matrix for a given (possibly nonunitary) transformation matrix U, which transforms a density matrix in vector form according to:

vec(rho) = U * vec(rho0)

or

rho = vec2mat(U * mat2vec(rho0))

U can be calculated for an open quantum system using the QuTiP propagator function.

Parameters **U** : Qobj

Transformation operator. Can be calculated using QuTiP propagator function.

op_basis_list : list

A list of Qobj's representing the basis states.

Returns **chi** : array

QPT chi matrix

qpt_plot(*chi*, *lbls_list*, *title=None*, *fig=None*, *axes=None*)

Visualize the quantum process tomography chi matrix. Plot the real and imaginary parts separately.

Parameters **chi** : array

Input QPT chi matrix.

lbls_list : list

List of labels for QPT plot axes.

title : string

Plot title.

fig : figure instance

User defined figure instance used for generating QPT plot.

axes : list of figure axis instance

User defined figure axis instance (list of two axes) used for generating QPT plot.

Returns **An matplotlib figure instance for the plot.** :

qpt_plot_combined(*chi*, *lbls_list*, *title=None*, *fig=None*, *ax=None*)
Visualize the quantum process tomography chi matrix. Plot bars with height and color corresponding to the absolute value and phase, respectively.

Parameters **chi** : array

Input QPT chi matrix.

lbls_list : list

List of labels for QPT plot axes.

title : string

Plot title.

fig : figure instance

User defined figure instance used for generating QPT plot.

ax : figure axis instance

User defined figure axis instance used for generating QPT plot (alternative to the fig argument).

Returns **An matplotlib figure instance for the plot.** :

Graphs and visualization

hinton(*rho*, *xlabels=None*, *ylabels=None*, *title=None*, *ax=None*)
Draws a Hinton diagram for visualizing a density matrix.

Parameters **rho** : qobj

Input density matrix.

xlabels : list of strings

list of x labels

ylabels : list of strings

list of y labels

title : string

title of the plot (optional)

ax : a matplotlib axes instance

The axes context in which the plot will be drawn.

Returns **An matplotlib axes instance for the plot.** :

Raises **ValueError** :

Input argument is not a quantum object.

matrix_histogram(*M*, *xlabels=None*, *ylabels=None*, *title=None*, *limits=None*, *colorbar=True*, *fig=None*, *ax=None*)
Draw a histogram for the matrix M, with the given x and y labels and title.

Parameters **M** : Matrix of Qobj

The matrix to visualize

xlabels : list of strings

list of x labels

ylabels : list of strings

list of y labels

title : string

title of the plot (optional)

limits : list/array with two float numbers

The z-axis limits [min, max] (optional)

ax : a matplotlib axes instance

The axes context in which the plot will be drawn.

Returns **An matplotlib axes instance for the plot.** :

Raises **ValueError** :

Input argument is not valid.

matrix_histogram_complex(*M*, *xlabels=None*, *ylabels=None*, *title=None*, *limits=None*, *phase_limits=None*, *colorbar=True*, *fig=None*, *ax=None*)

Draw a histogram for the amplitudes of matrix M, using the argument of each element for coloring the bars, with the given x and y labels and title.

Parameters **M** : Matrix of Qobj

The matrix to visualize

xlabels : list of strings

list of x labels

ylabels : list of strings

list of y labels

title : string

title of the plot (optional)

limits : list/array with two float numbers

The z-axis limits [min, max] (optional)

phase_limits : list/array with two float numbers

The phase-axis (colorbar) limits [min, max] (optional)

ax : a matplotlib axes instance

The axes context in which the plot will be drawn.

Returns **An matplotlib axes instance for the plot.** :

Raises **ValueError** :

Input argument is not valid.

energy_level_diagram(*H_list*, *N=0*, *figsize=(8, 12)*, *labels=None*, *fig=None*, *ax=None*)

Plot the energy level diagrams for a list of Hamiltonians. Include up to N energy levels. For each element in H_list, the energy levels diagram for the cummulative Hamiltonian sum(H_list[0:N]) is plotted, where N is the index of an element in H_list.

Parameters **H_list** : List of Qobj

A list of Hamiltonians.

labels [List of string] A list of labels for each Hamiltonian

N [int] The number of energy levels to plot

figsize [tuple (int,int)] The size of the figure (width, height).

Returns **The figure and axes instances used for the plot.** :

Raises **ValueError** :

Input argument is not valid.

wigner_cmap(*W*, *levels=1024*, *invert=False*)

A custom colormap that emphasizes negative values by creating a nonlinear colormap.

Parameters **W** : array

Wigner function array, or any array.

levels : int

Number of color levels to create.

invert : bool

Invert the color scheme for negative values so that smaller negative values have darker color.

Returns **Returns a Matplotlib colormap instance for use in plotting.** :

fock_distribution(*rho*, *fig=None*, *ax=None*, *figsize=(8, 6)*, *title=None*)

Plot the Fock distribution for a density matrix (or ket) that describes an oscillator mode.

Parameters **rho** : `qutip.qobj.Qobj`

The density matrix (or ket) of the state to visualize.

fig : a matplotlib Figure instance

The Figure canvas in which the plot will be drawn.

ax : a matplotlib axes instance

The axes context in which the plot will be drawn.

title : string

An optional title for the figure.

figsize : (width, height)

The size of the matplotlib figure (in inches) if it is to be created (that is, if no 'fig' and 'ax' arguments are passed).

Returns **A tuple of matplotlib figure and axes instances.** :

wigner_fock_distribution(*rho*, *fig=None*, *ax=None*, *figsize=(8, 4)*, *cmap=None*, *alpha_max=7.5*, *colorbar=False*)

Plot the Fock distribution and the Wigner function for a density matrix (or ket) that describes an oscillator mode.

Parameters **rho** : `qutip.qobj.Qobj`

The density matrix (or ket) of the state to visualize.

fig : a matplotlib Figure instance

The Figure canvas in which the plot will be drawn.

ax : a matplotlib axes instance

The axes context in which the plot will be drawn.

figsize : (width, height)

The size of the matplotlib figure (in inches) if it is to be created (that is, if no 'fig' and 'ax' arguments are passed).

cmap : a matplotlib cmap instance

The colormap.

alpha_max : float

The span of the x and y coordinates (both [-alpha_max, alpha_max]).

colorbar : bool

Whether (True) or not (False) a colorbar should be attached to the Wigner function graph.

Returns **A tuple of matplotlib figure and axes instances.** :

sphereplot (*theta*, *phi*, *values*, *save=False*)

Plots a matrix of values on a sphere

Parameters **theta** : float

Angle with respect to z-axis

phi : float

Angle in x-y plane :

values : array

Data set to be plotted

save : bool {False , True}

Whether to save the figure or not

Returns **Plots figure, returns nonthing.** :

6.2.3 Other Functions

Utility functions

This module contains utility functions that are commonly needed in other qutip modules.

n_thermal (*w*, *w_th*)

Return the number of photons in thermal equilibrium for an harmonic oscillator mode with frequency 'w', at the temperature described by 'w_th' where $\omega_{\rm th} = k_B T/\hbar$.

Parameters **w** : *float* or *array*

Frequency of the oscillator.

w_th : *float*

The temperature in units of frequency (or the same units as *w*).

Returns **n_avg** : *float* or *array*

Return the number of average photons in thermal equilibrium for a an oscillator with the given frequency and temperature.

linspace_with (*start*, *stop*, *num=50*, *elems=[ ]*)

Return an array of numbers sampled over specified interval with additional elements added.

Returns *num* spaced array with elements from *elems* inserted if not already included in set.

Returned sample array is not evenly spaced if addtional elements are added.

Parameters **start** : int

The starting value of the sequence.

stop : int

The stoping values of the sequence.

num : int, optional

Number of samples to generate.

elems : list/ndarray, optional

Requested elements to include in array

Returns **samples** : ndadrray

Original equally spaced sample array with additional elements added.

clebsch(*j1*, *j2*, *j3*, *m1*, *m2*, *m3*)

Calculates the Clebsch-Gordon coefficient for coupling (j1,m1) and (j2,m2) to give (j3,m3).

Parameters **j1** : float

Total angular momentum 1.

j2 : float

Total angular momentum 2.

j3 : float

Total angular momentum 3.

m1 : float

z-component of angular momentum 1.

m2 : float

z-component of angular momentum 2.

m3 : float

z-component of angular momentum 3.

Returns **cg_coeff** : float

Requested Clebsch-Gordan coefficient.

File I/O Functions

file_data_read(*filename*, *sep=None*)

Retrieves an array of data from the requested file.

Parameters **filename** : str

Name of file containing reqested data.

sep : str

Seperator used to store data.

Returns **data** : array_like

Data from selected file.

file_data_store(*filename*, *data*, *numtype='complex'*, *numformat='decimal'*, *sep=''*, *'*)

Stores a matrix of data to a file to be read by an external program.

Parameters **filename** : str

Name of data file to be stored, including extension.

data: array_like :

Data to be written to file.

numtype : str {'complex, 'real'}

Type of numerical data.

numformat : str {'decimal','exp'}

Format for written data.

sep : str

Single-character field seperator. Usually a tab, space, comma, or semicolon.

qload(*name*)
Loads data file from file named 'filename.qu' in current directory.

Parameters **name** : str

Name of data file to be loaded.

Returns **qobject** : instance / array_like

Object retrieved from requested file.

qsave(*data*, *name='qutip_data'*)
Saves given data to file named 'filename.qu' in current directory.

Parameters **data** : instance/array_like

Input Python object to be stored.

filename : str

Name of output data file.

Entropy Functions

concurrence(*rho*)
Calculate the concurrence entanglement measure for a two-qubit state.

Parameters **rho** : qobj

Density matrix for two-qubits.

Returns **concur** : float

Concurrence

entropy_conditional(*rho*, *selB*, *base=2.718281828459045*, *sparse=False*)
Calculates the conditional entropy $S(A|B) = S(A,B) - S(B)$ of a slected density matrix component.

Parameters **rho** : qobj

Density matrix of composite object

selB : int/list

Selected components for density matrix B

base : {e,2}

Base of logarithm.

sparse : {False,True}

Use sparse eigensolver.

Returns **ent_cond** : float

Value of conditional entropy

entropy_linear(*rho*)
Linear entropy of a density matrix.

Parameters **rho** : qobj

sensity matrix or ket/bra vector.

Returns **entropy** : float

Linear entropy of rho.

Examples

```
>>> rho=0.5*fock_dm(2,0)+0.5*fock_dm(2,1)
>>> entropy_linear(rho)
0.5
```

entropy_mutual(*rho*, *selA*, *selB*, *base=2.718281828459045*, *sparse=False*)

Calculates the mutual information S(A:B) between selection components of a system density matrix.

Parameters **rho** : qobj

Density matrix for composite quantum systems

selA : int/list

int or *list* of first selected density matrix components.

selB : int/list

int or *list* of second selected density matrix components.

base : {e,2}

Base of logarithm.

sparse : {False,True}

Use sparse eigensolver.

Returns **ent_mut** : float

Mutual information between selected components.

entropy_vn(*rho*, *base=2.718281828459045*, *sparse=False*)

Von-Neumann entropy of density matrix

Parameters **rho** : qobj

Density matrix.

base : {e,2}

Base of logarithm.

sparse : {False,True}

Use sparse eigensolver.

Returns **entropy** : float

Von-Neumann entropy of *rho*.

Examples

```
>>> rho=0.5*fock_dm(2,0)+0.5*fock_dm(2,1)
>>> entropy_vn(rho,2)
1.0
```

Quantum Computing Gates

cnot()

Quantum object representing the CNOT gate.

Returns **cnot_gate** : qobj

Quantum object representation of CNOT gate

Examples

```
>>> cnot()
Quantum object: dims = [[2, 2], [2, 2]], shape = [4, 4], type = oper, isHerm = True
Qobj data =
    [[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j]
     [ 0.+0.j  1.+0.j  0.+0.j  0.+0.j]
     [ 0.+0.j  0.+0.j  0.+0.j  1.+0.j]
     [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j]]
```

fredkin()

Quantum object representing the Fredkin gate.

Returns **fred_gate** : qobj

Quantum object representation of Fredkin gate.

Examples

```
>>> fredkin()
Quantum object: dims = [[2, 2, 2], [2, 2, 2]], shape = [8, 8], type = oper, isHerm = True
Qobj data =
    [[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
     [ 0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
     [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
     [ 0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
     [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j]
     [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j]
     [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j]
     [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  1.+0.j]]
```

phasegate(*theta*)

Returns quantum object representing the phase shift gate.

Parameters **theta** : float

Phase rotation angle.

Returns **phase_gate** : qobj

Quantum object representation of phase shift gate.

Examples

```
>>> phasegate(pi/4)
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isHerm = False
Qobj data =
[[ 1.00000000+0.j          0.00000000+0.j        ]
 [ 0.00000000+0.j          0.70710678+0.70710678j]]
```

snot()

Quantum object representing the SNOT (Hadamard) gate.

Returns **snot_gate** : qobj

Quantum object representation of SNOT (Hadamard) gate.

Examples

```
>>> snot()
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isHerm = True
Qobj data =
[[ 0.70710678+0.j  0.70710678+0.j]
 [ 0.70710678+0.j -0.70710678+0.j]]
```

swap(*mask=None*)

Quantum object representing the SWAP gate.

Returns **swap_gate** : qobj

Quantum object representation of SWAP gate

Examples

```
>>> swap()
Quantum object: dims = [[2, 2], [2, 2]], shape = [4, 4], type = oper, isHerm = True
Qobj data =
[[ 1.+0.j  0.+0.j  0.+0.j  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]
 [ 0.+0.j  0.+0.j  0.+0.j  1.+0.j]]
```

toffoli()

Quantum object representing the Toffoli gate.

Returns **toff_gate** : qobj

Quantum object representation of Toffoli gate.

Examples

```
>>> toffoli()
Quantum object: dims = [[2, 2, 2], [2, 2, 2]], shape = [8, 8], type = oper, isHerm = True
Qobj data =
    [[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
     [ 0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
     [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
     [ 0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j]
     [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j  0.+0.j]
     [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j  0.+0.j]
     [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  1.+0.j]
     [ 0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  0.+0.j  1.+0.j  0.+0.j]]
```

Density Matrix Metrics

A Module containing a collection of metrics (distance measures) between density matricies.

fidelity(*A*, *B*)

Calculates the fidelity (pseudo-metric) between two density matricies. See: Nielsen & Chuang, "Quantum Computation and Quantum Information"

Parameters **A** : qobj

Density matrix

B : qobj

Density matrix with same dimensions as A.

Returns **fid** : float

Fidelity pseudo-metric between A and B.

Examples

```
>>> x=fock_dm(5,3)
>>> y=coherent_dm(5,1)
>>> fidelity(x,y)
0.24104350624628332
```

tracedist(*A*, *B*, *sparse=False*, *tol=0*)

Calculates the trace distance between two density matricies. See: Nielsen & Chuang, "Quantum Computation and Quantum Information"

Parameters **A** : qobj

Density matrix.

B : qobj:

Density matrix with same dimensions as A.

tol : float

Tolerance used by sparse eigensolver. (0=Machine precision)

sparse : {False, True}

Use sparse eigensolver.

Returns **tracedist** : float

Trace distance between A and B.

Examples

```
>>> x=fock_dm(5,3)
>>> y=coherent_dm(5,1)
>>> tracedist(x,y)
0.9705143161472971
```

Random Operators and States

This module is a collection of random state and operator generators. The sparsity of the ouput Qobj's is controlled by varing the *density* parameter.

rand_dm(*N*, *density=0.75*, *pure=False*, *dims=None*)

Creates a random NxN density matrix.

Parameters **N** : int

Shape of output density matrix.

density : float

Density etween [0,1] of output density matrix.

dims : list

Dimensions of quantum object. Used for specifying tensor structure. Default is dims=[[N],[N]].

Returns **oper** : qobj

NxN density matrix quantum operator.

Notes

For small density matricies, choosing a low density will result in an error as no diagonal elements will be generated such that $Tr(\rho) = 1$.

rand_herm(*N*, *density=0.75*, *dims=None*)
Creates a random NxN sparse Hermitian quantum object.

Uses $H = X + X^+$ where X is a randomly generated quantum operator with a given *density*.

Parameters **N** : int

Shape of output quantum operator.

density : float

Density between [0,1] of output Hermitian operator.

dims : list

Dimensions of quantum object. Used for specifying tensor structure. Default is dims=[[N],[N]].

Returns **oper** : qobj

NxN Hermitian quantum operator.

rand_ket(*N*, *density=1*, *dims=None*)
Creates a random Nx1 sparse ket vector.

Parameters **N** : int

Number of rows for output quantum operator.

density : float

Density between [0,1] of output ket state.

dims : list

Dimensions of quantum object. Used for specifying tensor structure. Default is dims=[[N],[1]].

Returns **oper** : qobj

Nx1 ket state quantum operator.

rand_unitary(*N*, *density=0.75*, *dims=None*)
Creates a random NxN sparse unitary quantum object.

Uses $\exp(-iH)$ where H is a randomly generated Hermitian operator.

Parameters **N** : int

Shape of output quantum operator.

density : float

Density between [0,1] of output Unitary operator.

dims : list

Dimensions of quantum object. Used for specifying tensor structure. Default is dims=[[N],[N]].

Returns **oper** : qobj

NxN Unitary quantum operator.

IPython notebook tools

This module contains utility functions for using QuTiP with IPython notebooks.

parfor(*task*, *task_vec*, *args=None*, *client=None*, *view=None*, *show_scheduling=False*, *show_progressbar=False*)

Call the function `tast` for each value in `task_vec` using a cluster of IPython engines. The function `task` should have the signature `task(value, args)` or `task(value)` if `args=None`.

The `client` and `view` are the IPython.parallel client and load-balanced view that will be used in the parfor execution. If these are `None`, new instances will be created.

Parameters **task: a Python function** :

The function that is to be called for each value in `task_vec`.

task_vec: array / list :

The list or array of values for which the `task` function is to be evaluated.

args: list / dictionary :

The optional additional argument to the `task` function. For example a dictionary with parameter values.

client: IPython.parallel.Client :

The IPython.parallel Client instance that will be used in the parfor execution.

view: a IPython.parallel.Client view :

The view that is to be used in scheduling the tasks on the IPython cluster. Preferably a load-balanced view, which is obtained from the IPython.parallel.Client instance client by calling, view = client.load_balanced_view().

show_scheduling: bool {False, True}, default False :

Display a graph showing how the tasks (the evaluation of `task` for for the value in `task_vec1`) was scheduled on the IPython engine cluster.

show_progressbar: bool {False, True}, default False :

Display a HTML-based progress bar duing the execution of the parfor loop.

Returns **result** : list

The result list contains the value of `task(value, args)` for each value in `task_vec`, that is, it should be equivalent to `[task(v, args) for v in task_vec]`.

version_table()

Print an HTML-formatted table with version numbers for QuTiP and its dependencies. Use it in a IPython notebook to show which versions of different packages that were used to run the notebook. This should make it possible to reproduce the environment and the calculation later on.

Returns **version_table: string** :

Return an HTML-formatted string containing version information for QuTiP dependencies.

Miscellaneous

about()

About box for qutip. Gives version numbers for QuTiP, NumPy, SciPy, and MatPlotLib. GUI version requires PySide or PyQt4.

demos()

Calls the demos scripts via a GUI window if PySide or PyQt4 are avaliable. Otherwise, a commandline interface is given in the terminal.

orbital (*theta*, *phi*, **args*)

Calculates an angular wave function on a sphere. `psi = orbital(theta,phi,ket1,ket2,...)` calculates the angular wave function on a sphere at the mesh of points defined by theta and phi which is $\sum_{lm} c_{lm} Y_{lm}(theta, phi)$ where C_{lm} are the coefficients specified by the list of kets. Each ket has 2l+1 components for some integer l.

Parameters **theta** : list/array

Polar angles

phi : list/array

Azimuthal angles

args : list/array

`list` of ket vectors.

Returns **‘‘array‘‘ for angular wave function** :

parfor (*func*, *frange*, *num_cpus=0*)

Executes a single-variable function in parallel.

Parallel execution of a for-loop over function *func* for a single variable *frange*.

Parameters **func: function_type** :

A single-variable function.

frange: array_type :

An `array` of values to be passed on to *func*.

num_cpus : int {0}

Number of CPU’s to use. Default ‘0’ uses max. number of CPU’s. Performance degrades if num_cpus is larger than the physical CPU count of your machine.

Returns **ans** : list

A `list` with length equal to number of input parameters containting the output from *func*. In general, the ordering of the output variables will not be in the same order as *frange*.

Notes

Multiple values can be passed into the parfor function using Pythons builtin ‘zip’ command, or using multidimensional *lists* or *arrays*.

rhs_generate (*H*, *c_ops*, *args={}*, *options=<qutip.odeoptions.Odeoptions instance at 0x10e221c20>*, *name=None*)

Generates the Cython functions needed for solving the dynamics of a given system using the mesolve function inside a parfor loop.

Parameters **H** : qobj

System Hamiltonian.

c_ops : list

`list` of collapse operators.

args : dict

Arguments for time-dependent Hamiltonian and collapse operator terms.

options : Odeoptions

Instance of ODE solver options.

name: str :

Name of generated RHS

Notes

Using this function with any solver other than the mesolve function will result in an error.

rhs_clear()
Resets the string-format time-dependent Hamiltonian parameters.

Returns **Nothing, just clears data from internal odeconfig module.** :

simdiag(*ops*, *evals=True*)
Simulateous diagonalization of communting Hermitian matricies.

Parameters **ops** : list/array

`list` or `array` of qobjs representing commuting Hermitian operators.

Returns **eigs** : tuple

Tuple of arrays representing eigvecs and eigvals of quantum objects corresponding to simultaneous eigenvectors and eigenvalues for each operator.

CHAPTER

SEVEN

CHANGE LOG

7.1 Version 2.2.0 (March 01, 2013):

7.1.1 New Features

- **Added Support for Windows**
- New Bloch3d class for plotting 3D Bloch spheres using Mayavi.
- Bloch sphere vectors now look like arrows.
- Added partial transpose function.
- Added continous variable functions for calculating correlation and covariance matrices, the Wigner covariance matrix and the logarithmic negativity for for multimode fields in Fock basis.
- The master-equation solver (mesolve) now accepts pre-constructed Liouvillian terms, which makes it possible to solve master equations that are not on the standard Lindblad form.
- Optional Fortran Monte Carlo solver (mcsolve_f90) by Arne Grimsmo.
- A module of tools for using QuTiP in IPython notebooks.
- Increased performance of the steady state solver.

7.1.2 Bug Fixes:

- Function based time-dependent Hamiltonians now keep the correct phase.
- mcsolve no longer prints to the command line if ntraj=1.

7.2 Version 2.1.0 [SVN-2683] (October 05, 2012):

7.2.1 New Features

- New method for generating Wigner functions based on Laguerre polynomials.
- coherent(), coherent_dm(), and thermal_dm() can now be expressed using analytic values.
- Unittests now use nose and can be run after installation.
- Added iswap and sqrt-iswap gates.
- Functions for quantum process tomography.
- Window icons are now set for Ubuntu application launcher.
- The propagator function can now take a list of times as argument, and returns a list of corresponding propagators.

7.2.2 Bug Fixes:

SVN-2571: mesolver now correctly uses the user defined rhs_filename in Odeoptions().

SVN-2566: rhs_generate() now handles user defined filenames properly.

SVN-2565: Density matrix returned by propagator_steadystate is now Hermitian.

SVN-2548: eseries_value returns real list if all imag parts are zero.

SVN-2518: mcsolver now gives correct results for strong damping rates.

SVN-2513: Odeoptions now prints mc_avg correctly.

SVN-2516: Do not check for PyObj in mcsolve when gui=False.

SVN-2514: Eseries now correctly handles purely complex rates.

SVN-2485: thermal_dm() function now uses truncated operator method.

SVN-2428: Cython based time-dependence now Python 3 compatible.

SVN-2391: Removed call to NSAutoPool on mac systems.

SVN-2389: Progress bar now displays the correct number of CPU's used.

SVN-2385: Qobj.diag() returns reals if operator is Hermitian.

SVN-2376: Text for progress bar on Linux systems is no longer cutoff.

7.3 Version 2.0.0 [SVN-2354] (June 01, 2012):

7.3.1 New Features

- QuTiP now includes solvers for both Floquet and Bloch-Redfield master equations.
- The Lindblad master equation and Monte Carlo solvers allow for time-dependent collapse operators.
- It is possible to automatically compile time-dependent problems into c-code using Cython (if installed).
- Python functions can be used to create arbitrary time-dependent Hamiltonians and collapse operators.
- Solvers now return Odedata objects containing all simulation results and parameters, simplifying the saving of simulation results.
- mesolve and mcsolve can reuse Hamiltonian data when only the initial state, or time-dependent arguments, need to be changed.
- QuTiP includes functions for creating random quantum states and operators.
- The generation and manipulation of quantum objects is now more efficient.
- Quantum objects have basis transformation and matrix element calculations as built-in methods.
- The quantum object eigensolver can use sparse solvers.
- The partial-trace (ptrace) function is up to 20x faster.
- The Bloch sphere can now be used with the Matplotlib animation function, and embedded as a subplot in a figure.
- QuTiP has built-in functions for saving quantum objects and data arrays.
- The steady-state solver has been further optimized for sparse matrices, and can handle much larger system Hamiltonians.
- The steady-state solver can use the iterative bi-conjugate gradient method instead of a direct solver.
- There are three new entropy functions for concurrence, mutual information, and conditional entropy.
- Correlation functions have been combined under a single function.

- The operator norm can now be set to trace, Frobius, one, or max norm.
- Global QuTiP settings can now be modified.
- QuTiP includes a collection of unit tests for verifying the installation.
- Demos window now lets you copy and paste code from each example.

7.4 Version 1.1.4 [fixes backported to SVN-1450] (May 28, 2012):

7.4.1 Bug Fixes:

SVN-2101: Fixed bug pointed out by Brendan Abolins.

SVN-1796: Qobj.tr() returns zero-dim ndarray instead of float or complex.

SVN-1463: Updated factorial import for scipy version 0.10+

7.5 Version 1.1.3 [svn-1450] (November 21, 2011):

7.5.1 New Functions:

SVN-1347: Allow custom naming of Bloch sphere.

7.5.2 Bug Fixes:

SVN-1450: Fixed text alignment issues in AboutBox.

SVN-1448: Added fix for SciPy V>0.10 where factorial was moved to scipy.misc module.

SVN-1447: Added tidyup function to tensor function output.

SVN-1442: Removed openmp flags from setup.py as new Mac Xcode compiler does not recognize them.

SVN-1435: Qobj diag method now returns real array if all imaginary parts are zero.

SVN-1434: Examples GUI now links to new documentation.

SVN-1415: Fixed zero-dimensional array output from metrics module.

7.6 Version 1.1.2 [svn-1218] (October 27, 2011)

7.6.1 Bug Fixes

SVN-1218: Fixed issue where Monte Carlo states were not output properly.

7.7 Version 1.1.1 [svn-1210] (October 25, 2011)

THIS POINT-RELEASE INCLUDES VASTLY IMPROVED TIME-INDEPENDENT MCSOLVE AND ODESOLVE PERFORMANCE

7.7.1 New Functions

SVN-1183: Added linear entropy function.

SVN-1179: Number of CPU's can now be changed.

7.7.2 Bug Fixes

SVN-1184: Metrics no longer use dense matrices.

SVN-1184: Fixed Bloch sphere grid issue with matplotlib 1.1.

SVN-1183: Qobj trace operation uses only sparse matrices.

SVN-1168: Fixed issue where GUI windows do not raise to front.

7.8 Version 1.1.0 [svn-1097] (October 04, 2011)

THIS RELEASE NOW REQUIRES THE GCC COMPILER TO BE INSTALLED

7.8.1 New Functions

SVN-1054: tidyup function to remove small elements from a Qobj.

SVN-1051: Added concurrence function.

SVN-1036: Added simdiag for simultaneous diagonalization of operators.

SVN-1032: Added eigenstates method returning eigenstates and eigenvalues to Qobj class.

SVN-1030: Added fileio for saving and loading data sets and/or Qobj's.

SVN-1029: Added hinton function for visualizing density matrices.

7.8.2 Bug Fixes

SVN-1091: Switched Examples to new Signals method used in PySide 1.0.6+.

SVN-1090: Switched ProgressBar to new Signals method.

SVN-1075: Fixed memory issue in expm functions.

SVN-1069: Fixed memory bug in isherm.

SVN-1059: Made all Qobj data complex by default.

SVN-1053: Reduced ODE tolerance levels in Odeoptions.

SVN-1050: Fixed bug in ptrace where dense matrix was used instead of sparse.

SVN-1047: Fixed issue where PyQt4 version would not be displayed in about box.

SVN-1041: Fixed issue in Wigner where xvec was used twice (in place of yvec).

7.9 Version 1.0.0 [svn-1021] (July 29, 2011)

Initial release.

CHAPTER

EIGHT

DEVELOPERS

8.1 Lead Developers

Robert Johansson (RIKEN)

Paul Nation (Korea University)

8.2 Contributors

Note: Anyone is welcome to contribute to QuTiP. If you are interested in helping, please let us know!

Bredan Abolins (Berkeley) - Bug hunter

Markus Baden (Centre for Quantum Technologies, Singapore) - Code contributor, documentation, **QuTiP member**

James Clemens (Miami University - Ohio) - Bug hunter

Claudia Degrandi (Yale University) - Documentation

Arne Grimsmo (University of Auckland) - Code contributor, Bug hunter

JP Hadden (University of Bristol) - Code contributor, improved Bloch sphere visualization

Myung-Joong Hwang (Pohang University of Science and Technology) - Bug hunter

Robert Jördens (NIST) - Linux packaging

Hwajung Kang (Systems Biology Institute, Tokyo) - Suggestions for improving Bloch class

Neill Lambert (RIKEN) - Windows support

Anders Lund - Bug hunting for the Monte-Carlo solver.

Jonas Neergaard-Nielsen (Technical University of Denmark) - Bug hunter, Code contributor, Windows support

Henri Nielsen (Technical University of Denmark) - Bug hunter

Per Kaer Nielsen (Technical University of Denmark) - Testing the ODE solver

Florian Ong (Institute for Quantum Computation) - Bug hunter

Denis Vasilyev (St. Petersburg State University) - Bug hunter

André Xuereb (University of Hannover) - Bug hunter

Dong Zhou (Yale University) - Bug hunter

CHAPTER

NINE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

q