QuTiP: Quantum Toolbox in Python

Release 4.2.0

P.D. Nation, J.R. Johansson, A.J.G. Pitchford, C. Granade, and A

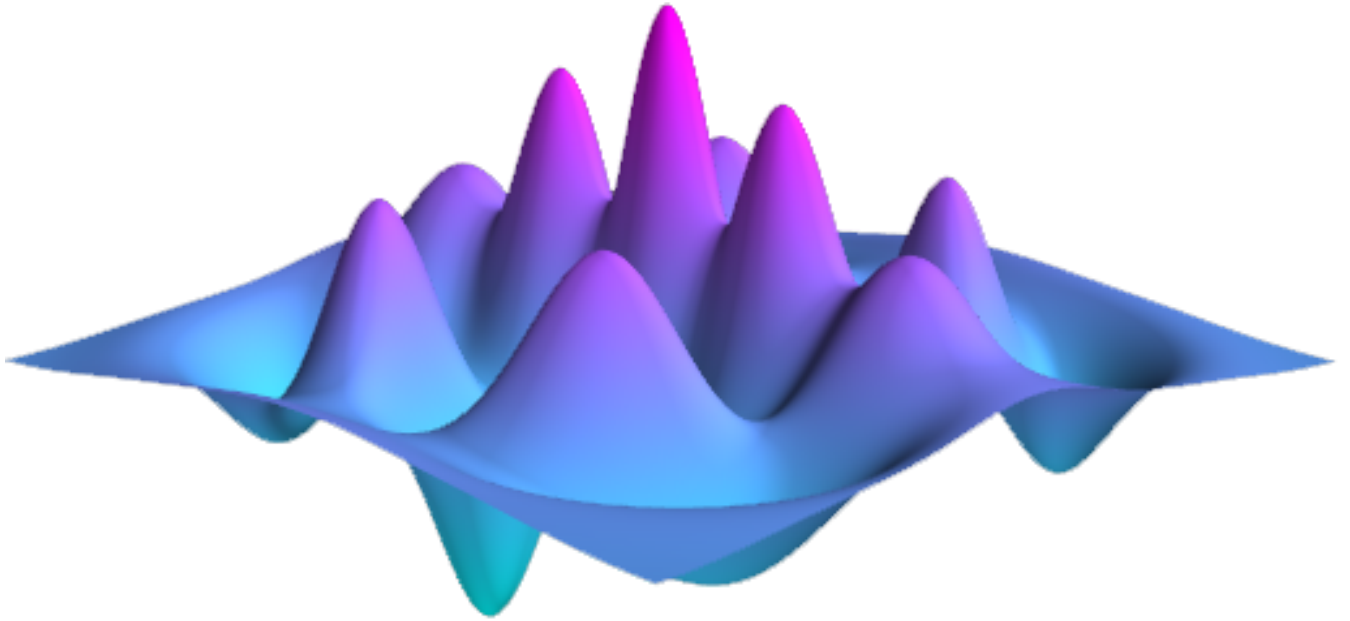**Jan 17, 2018**

Contents

Contents

Chapter 1

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 documentation page.

For more information see the QuTiP project web page.

Author P.D. Nation

Author Alexander Pitchford

Author Arne Grimsmo

Author J.R. Johansson

Author Chris Grenade

version 4.2

status Released (July 28, 2017)

copyright This documentation is licensed under the Creative Commons Attribution 3.0 Unported License.

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

QuTiP is developed under the auspice of the non-profit organization:

1.4 About QuTiP

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 practical 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 limited number of quanta, that are amenable to classical simulation in a truncated Hilbert space.

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

- QuTiP relies entirely on open-source software. You are free to modify and use it as you wish with no licensing fees or limitations.

- 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, Scipy, and Cython libraries, and are based on many of the same algorithms used in propriety software.

- QuTiP allows for solving the dynamics of Hamiltonians with (almost) 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.

For detailed information about new features of each release of QuTiP, see the *Change Log*.

1.5 Contributing to QuTiP

We welcome anyone who is interested in helping us make QuTiP the best package for simulating quantum 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.

Chapter 2

Installation

2.1 General Requirements

QuTiP depends on several open-source libraries for scientific computing in the Python programming language. The following packages are currently required:

Package	Version	Details
Python	2.7+	Version 3.5+ is highly recommended.
NumPy	1.8+	Not tested on lower versions.
SciPy	0.15+	Lower versions have missing features.
Matplotlib	1.2.1+	Some plotting does not work on lower versions.
Cython	0.21+	Needed for compiling some time-dependent Hamiltonians.
C++ Compiler	GCC 4.7+, MS VS 2015	Needed for compiling Cython files.
Python Headers	2.7+	Linux only. Needed for compiling Cython files.

In addition, there are several optional packages that provide additional functionality:

Package	Version	Details
LaTeX	TexLive 2009+	Needed if using LaTeX in matplotlib figures.
nose	1.1.2+	For running the test suite.

We would not recommend installation into the system Python on Linux platforms, as it is likely that the required libraries will be difficult to update to sufficiently recent versions. The system Python on Linux is used for system things, changing its configuration could lead to highly undesirable results. We are recommending and supporting Anaconda / Miniconda Python environments for QuTiP on all platformsx [It is also possible to install the Intel Python Distribution via the conda installer in Anaconda].

2.2 Platform-independent Installation

QuTiP is designed to work best when using the Anaconda or Intel Python distributions that support the conda package management system.

If you aleady have your conda environment set up, and have the `conda-forge` channel available, then you can install QuTiP using:

```
conda install qutip
```

Otherwise refer to *building-conda-environment*

If you are using MS Windows, then you will probably want to refer to *installation-on-MS-Windows*

2.2.1 Building your Conda environment

Important: There are no working conda-forge packages for Python 2.7 on Windows. On Windows you should create a Python 3.5+ environment.

The default Anaconda environment has all the Python packages needed for running QuTiP. You may however wish to install QuTiP in a Conda environment (env) other than the default Anaconda environment. You may wish to install Miniconda instead if you need to be economical with disk space. However, if you are not familiar with conda environments and only plan to use if for QuTiP, then you should probably work with a default Anaconda / Miniconda environment.

To create a Conda env for QuTiP called `qutip-env`:

```
conda create -n qutip-env python=3
```

Note the `python=3` can be ommited if you want the default Python version for the Anaconda / Miniconda install.

If you have created a specific conda environment, or you have installed Miniconda, then you will need to install the required packages for QuTiP.

recommended:

```
conda install numpy scipy cython matplotlib nose jupyter notebook spyder
```

minimum (recommended):

```
conda install numpy scipy cython nose matplotlib
```

absolute mimimum:

```
conda install numpy scipy cython
```

The `jupyter` and `notebook` packages are for working with Jupyter notebooks (fka IPython notebooks). Spyder is an IDE for scientific development with Python.

2.2.2 Adding the conda-forge channel

If you have conda 4.1.0 or later then, add the conda-forge channel with lowest priority using:

```
conda config --append channels conda-forge
```

Otherwise you should consider reinstalling Anaconda / Miniconda. In theory:

```
conda update conda
```

will update your conda to the latest version, but this can lead to breaking your default Ananconda enviroment.

Alternatively, this will add `conda-forge` as the highest priority channel.

```
conda config --add channels conda-forge
```

It is almost certainly better to have `defaults` as the highest priority channel. You can edit your `.condarc` (user home folder) file manually, so that `conda-forge` is below `defaults` in the `channels` list.

2.3 Installing via pip

For other types of installation, it is often easiest to use the Python package manager pip.

```
pip install qutip
```

More detailed platform-dependent installation alternatives are given below.

2.4 Installing from Source

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

> http://www.qutip.org/download.html

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.

Installing QuTiP from source requires that all the dependencies are satisfied. To install QuTiP from the source code run:

```
python setup.py install
```

To install OPENMP support, if available, run:

```
python setup.py install --with-openmp
```

If you are wishing to contribute to the QuTiP project, then you will want to create your own fork of qutip, clone this to a local folder, and 'install' it into your Python env using:

```
python setup.py develop --with-openmp
```

`import qutip` in this Python env will then load the code from your local fork, enabling you to test changes interactively.

The `sudo` pre-command is typically not needed when installing into Anaconda type environments, as Anaconda is usually installed in the users home directory. `sudo` will be needed (on Linux and OSX) for installing into Python environments where the user does not have write access.

2.5 Installation on MS Windows

Important: Installation on Windows has changed substantially as of QuTiP 4.1. The only supported installation configuration is using the Conda environment with Python 3.5+ and Visual Studio 2015.

We are recommending and supporting installation of QuTiP into a Conda environment. Other scientific Python implementations such as Python-xy may also work, but are not supported.

As of QuTiP 4.1, recommended installation on Windows requires Python 3.5+, as well as Visual Studio 2015. With this configuration, one can install QuTiP using any of the above mentioned receipes. Visual Studio 2015 is not required for the install of the conda-forge package, but it is required at runtime for the string format time-dependence solvers. When installing Visual Studio 2015 be sure to select options for the C++ compiler.

The 'Community' edition of Visual Studio 2015 is free to download use, however it does require approx 10GB of disk space, much of which does have to be on the system drive. If this is not feasible, then it is possible to run QuTiP under Python 2.7.

2.5.1 Windows and Python 2.7

Important: Running QuTiP under Python 2.7 on Windows is not recommended or supported. However, it is currently possible. There are no working conda-forge packages for Python 2.7 on Windows. You will have to install via pip or from source in Python 2.7 on Windows. The 'MS Visual C for Python 2.7' compiler will not work with QuTiP. You will have to use the g++ compiler in mingw32.

If you need to create a Python 2.7 conda environment see *building-conda-environment*, including *adding-conda-forge*

Then run:

```
conda install mingwpy
```

To specify the use of the mingw compiler you will need to create the following file:

```
<path to my Python env>/Lib/distutils/distutils.cfg
```

with the following contents:

```
[build]
compiler=mingw32
[build_ext]
compiler=mingw32
```

`<path to my Python env>` will be something like `C:\Ananconda2\` or `C:\Ananconda2\envs\qutip-env\` depending on where you installed Anaconda or Miniconda, and whether you created a specific environment.

You can then install QuTiP using either the *install-via_pip* or *install-get-it* method.

2.6 Verifying the Installation

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

```python
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 you 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.

2.7 Checking Version Information using the About Function

QuTiP includes an "about" function for viewing information about QuTiP and the important dependencies installed on your system. To view this information:

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

In [2]: about()
```

Chapter 3

Users Guide

3.1 Guide Overview

The goal of this guide is to introduce you to the basic structures and functions that make up QuTiP. This guide is divided up into several sections, each highlighting a specific set of functionalities. In combination with the examples that can be found on the project web page http://qutip.org/tutorials.html, this guide should provide a more or less complete overview. In addition, the *API documentation* for each function is located at the end of this guide.

3.1.1 Organization

QuTiP is designed to be a general framework for solving quantum mechanics problems such as systems composed of few-level quantum systems and harmonic oscillators. To this end, QuTiP is built from a large (and ever growing) library of functions and classes; from `qutip.states.basis` to `qutip.wigner`. The general organization of QuTiP, highlighting the important API available to the user, is shown in the *Tree-diagram of the 306 user accessible functions and classes in QuTiP 3.2.*

Fig. 3.1: Tree-diagram of the 306 user accessible functions and classes in QuTiP 3.2.

3.2 Basic Operations on Quantum Objects

3.2.1 First things first

> **Warning:** 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. Often, we also need to import the NumPy and Matplotlib libraries with:

```
In [2]: import numpy as np
```

```
In [3]: import matplotlib.pyplot as plt
```

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: *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 interpreter prompt, Python script, or Jupyter notebook.

3.2.2 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 [4]: Qobj()
Out[4]:
Quantum object: dims = [[1], [1]], shape = (1, 1), type = bra
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 [5]: Qobj([[1],[2],[3],[4],[5]])
Out[5]:
Quantum object: dims = [[5], [1]], shape = (5, 1), type = ket
Qobj data =
[[ 1.]
 [ 2.]
 [ 3.]
 [ 4.]
 [ 5.]]
```

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

Manually specifying the data for each quantum 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)	`coherent_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
Charge operator	`charge(N,M=-N)`	Diagonal operator with entries from M..0..N.
Commutator	`commutator(A, B, kind)`	Kind = 'normal' or 'anti'.
Diagonals operator	`qdiags(N)`	Quantum object created from arrays of diagonals at given offsets.
Displacement operator (Single-mode)	`displace(N, alpha)`	N=number of levels in Hilbert space, alpha = complex displacement amplitude.
Higher spin operators	`jmat(j,#s)`	j = integer or half-integer representing spin, s = 'x', 'y', 'z', '+', or '-'
Identity	`qeye(N)`	N = number of levels in Hilbert space.
Lowering (destruction) operator	`destroy(N)`	same as above
Momentum operator	`momentum(N)`	same as above
Number operator	`num(N)`	same as above
Phase operator (Single-mode)	`phase(N, phi0)`	Single-mode Pegg-Barnett phase operator with ref phase phi0.
Position operator	`position(N)`	same as above
Raising (creation) operator	`create(N)`	same as above
Squeezing operator (Single-mode)	`squeeze(N, sp)`	N=number of levels in Hilbert space, sp = squeezing parameter.
Squeezing operator (Generalized)	`squeezing(q1, q2, sp)`	q1,q2 = Quantum operators (Qobj) sp = squeezing parameter.
Sigma-X	`sigmax()`	
Sigma-Y	`sigmay()`	
Sigma-Z	`sigmaz()`	
Sigma plus	`sigmap()`	
Sigma minus	`sigmam()`	
Tunneling operator	`tunneling(N,m)`	Tunneling operator with elements of the form $\|N><N+m\| + \|N+m><N\|$.

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

```
In [6]: basis(5,3)
Out[6]:
Quantum object: dims = [[5], [1]], shape = (5, 1), type = ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 0.]
 [ 1.]
 [ 0.]]
```

```
In [7]: coherent(5,0.5-0.5j)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
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 [8]: destroy(4)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
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 [9]: sigmaz()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 1.  0.]
 [ 0. -1.]]

In [10]: jmat(5/2.0,'+')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
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 [11]: q = destroy(4)

In [12]: q.dims
Out[12]: [[4], [4]]

In [13]: q.shape
\\\\\\\\\\\\\\\\\\\\\\\\\Out[13]: (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'?
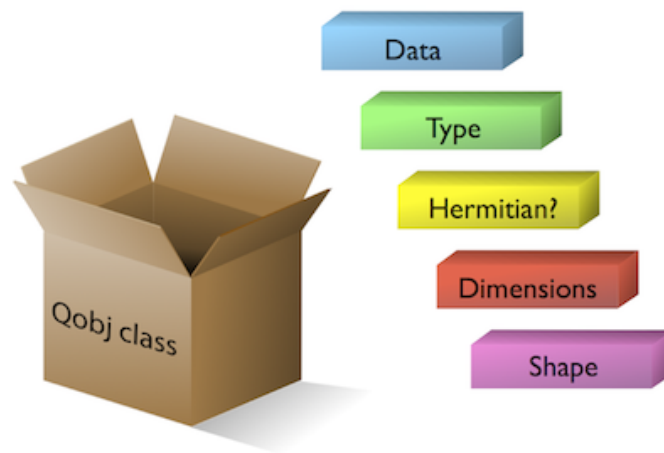
Fig. 3.2: The `Qobj` Class viewed as a container for the properties need to characterize a quantum operator or state vector.

For the destruction operator above:

```
In [14]: q.type
Out[14]: 'oper'

In [15]: q.isherm
\\\\\\\\\\\\\\\\\\Out[15]: False

In [16]: q.data
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[16]:
<4x4 sparse matrix of type '<class '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 `Qobj` instances store their data as a sparse matrix to save memory. To access the underlying dense matrix one needs to use the *`qutip.Qobj.full`* function as described below.

Qobj Math

The rules for mathematical operations on `Qobj` instances are similar to standard matrix arithmetic:

```
In [17]: q = destroy(4)

In [18]: x = sigmax()

In [19]: q + 5
Out[19]:
```

```
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 [20]: x * x
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 1.  0.]
 [ 0.  1.]]

In [21]: q ** 3
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
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 [22]: x / np.sqrt(2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
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:

```
In [23]: q * x
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-23-c5138e004127> in <module>()
----> 1 q * x

/home/agp1/anaconda3/lib/python3.6/site-packages/qutip/qobj.py in __mul__(self,
↪other)
    513
    514             else:
--> 515                 raise TypeError("Incompatible Qobj shapes")
    516
    517         elif isinstance(other, np.ndarray):

TypeError: Incompatible Qobj shapes
```

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

3.2.3 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
Check Hermicity	`Q.check_herm()`	Check if quantum object is Hermitian
Conjugate	`Q.conj()`	Conjugate of quantum object.
Cosine	`Q.cosm()`	Cosine of quantum object.
Dagger (adjoint)	`Q.dag()`	Returns adjoint (dagger) of object.
Diagonal	`Q.diag()`	Returns the diagonal elements.
Diamond Norm	`Q.dnorm()`	Returns the diamond norm.
Eigenenergies	`Q.eigenenergies()`	Eigenenergies (values) of operator.
Eigenstates	`Q.eigenstates()`	Returns eigenvalues and eigenvectors.
Eliminate States	`Q.eliminate_states(inds)`	Returns quantum object with states in list inds removed.
Exponential	`Q.expm()`	Matrix exponential of operator.
Extract States	`Q.extract_states(inds)`	Qobj with states listed in inds only.
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.
Overlap	`Q.overlap(state)`	Overlap between current Qobj and a given state.
Partial Trace	`Q.ptrace(sel)`	Partial trace returning components selected using 'sel' parameter.
Permute	`Q.permute(order)`	Permutes the tensor structure of a composite object in the given order.
Projector	`Q.proj()`	Form projector operator from given ket or bra vector.
Sine	`Q.sinm()`	Sine of quantum operator.
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.
Truncate Neg	`Q.trunc_neg()`	Truncates negative eigenvalues
Unit	`Q.unit()`	Returns normalized (unit) vector Q/Q.norm().

```
In [24]: basis(5, 3)
Out[24]:
Quantum object: dims = [[5], [1]], shape = (5, 1), type = ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 0.]
 [ 1.]
 [ 0.]]

In [25]: basis(5, 3).dag()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
Quantum object: dims = [[1], [5]], shape = (1, 5), type = bra
Qobj data =
[[ 0.  0.  0.  1.  0.]]

In [26]: coherent_dm(5, 1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
```

```
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 [27]: coherent_dm(5, 1).diag()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪array([ 0.36791117,  0.36757705,  0.18523331,  0.05810197,  0.0211765 ])

In [28]: coherent_dm(5, 1).full()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
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 [29]: coherent_dm(5, 1).norm()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪1.0

In [30]: coherent_dm(5, 1).sqrtm()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
Quantum object: dims = [[5], [5]], shape = (5, 5), type = oper, isherm = True
Qobj data =
[[ 0.36791119  0.36774406  0.2610544   0.14620658  0.08826704]
 [ 0.36774406  0.36757705  0.26093584  0.14614018  0.08822695]
 [ 0.2610544   0.26093584  0.18523332  0.10374209  0.0626306 ]
 [ 0.14620658  0.14614018  0.10374209  0.05810197  0.03507701]
 [ 0.08826704  0.08822695  0.0626306   0.03507701  0.0211765 ]]

In [31]: coherent_dm(5, 1).tr()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪1.0

In [32]: (basis(4, 2) + basis(4, 1)).unit()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
Quantum object: dims = [[4], [1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.        ]
 [ 0.70710678]
 [ 0.70710678]
 [ 0.        ]]
```

3.3 Manipulating States and Operators

3.3.1 Introduction

In the previous guide section *Basic Operations on Quantum Objects*, we saw how to create states and operators, 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 examples on the tutorials web page.

3.3.2 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]: vac = basis(5, 0)

In [2]: vac
Out[2]:
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 [3]: a = destroy(5)

In [4]: a
Out[4]:
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 `vac`,

```
In [5]: a * vac
Out[5]:
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 [6]: a.dag() * vac
Out[6]:
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 [7]: c = create(5)

In [8]: c * vac
Out[8]:
Quantum object: dims = [[5], [1]], shape = (5, 1), type = ket
Qobj data =
[[ 0.]
 [ 1.]
 [ 0.]
 [ 0.]
 [ 0.]]
```

which does the same thing. We can raise the vacuum state more than once by successively apply the raising operator:

```
In [9]: c * c * vac
Out[9]:
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 [10]: c ** 2 * vac
Out[10]:
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 `vac`:

```
In [11]: c * a * vac
Out[11]:
Quantum object: dims = [[5], [1]], shape = (5, 1), type = ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]]
```

or on the $|1\rangle$ state:

3.3. Manipulating States and Operators

```
In [12]: c * a * (c * vac)
Out[12]:
Quantum object: dims = [[5], [1]], shape = (5, 1), type = ket
Qobj data =
[[ 0.]
 [ 1.]
 [ 0.]
 [ 0.]
 [ 0.]]
```

or the $|2\rangle$ state:

```
In [13]: c * a * (c**2 * vac)
Out[13]:
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 [14]: c * a * (c**2 * vac).unit()
Out[14]:
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 [15]: ket = basis(5, 2)

In [16]: print(ket)
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 [17]: n = num(5)

In [18]: 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 * vac).unit()` we have:

```
In [19]: n * ket
Out[19]:
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 [20]: ket = (basis(5, 0) + basis(5, 1)).unit()

In [21]: print(ket)
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 [22]: n * ket
Out[22]:
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.squeeze* functions to the vacuum state:

```
In [23]: vac = basis(5, 0)

In [24]: d = displace(5, 1j)

In [25]: s = squeeze(5, 0.25 + 0.25j)

In [26]: d * vac
Out[26]:
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 [27]: d * s * vac
Out[27]:
```

```
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.

3.3.3 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 [28]: ket = basis(5, 2)

In [29]: ket * ket.dag()
Out[29]:
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 [30]: fock_dm(5, 2)
Out[30]:
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 [31]: ket2dm(ket)
Out[31]:
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 [32]: 0.5 * ket2dm(basis(5, 4)) + 0.5 * ket2dm(basis(5, 2))
Out[32]:
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 [33]: coherent_dm(5, 1.25)
Out[33]:
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 [34]: thermal_dm(5, 1.25)
Out[34]:
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*, fidelity *qutip.metrics.fidelity*, Hilbert-Schmidt distance *qutip.metrics.hilbert_dist*, Bures distance *qutip.metrics.bures_dist*, and Bures angle *qutip.metrics.bures_angle*.

```
In [35]: x = coherent_dm(5, 1.25)

In [36]: y = coherent_dm(5, 1.25j)   # <-- note the 'j'

In [37]: z = thermal_dm(5, 0.125)

In [38]: fidelity(x, x)
Out[38]: 1.0000000102501865

In [39]: tracedist(y, y)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[39]: 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 [40]: tracedist(y, x)
Out[40]: 0.97715657979973
```

```
In [41]: np.sqrt(1 - fidelity(y, x) ** 2)
Out[41]: 0.97715656996219658
```

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

```
In [42]: 1 - fidelity(x, z) ** 2
Out[42]: 0.7782890516928219
```

```
In [43]: tracedist(x, z)
Out[43]: 0.8559028328862572
```

3.3.4 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 [44]: 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 [45]: vac = 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 `vac` 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 [46]: vac
Out[46]:
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[ 1.]
 [ 0.]]
```

```
In [47]: c = create(2)

In [48]: c * vac
Out[48]:
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 [49]: spin
Out[49]:
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[ 1.]
 [ 0.]]

In [50]: sigmap() * spin
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
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 [51]: sigmaz()
Out[51]:
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 1.   0.]
 [ 0. -1.]]

In [52]: sigmaz() * spin
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[ 1.]
 [ 0.]]

In [53]: spin2 = basis(2, 1)

In [54]: spin2
Out[54]:
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[ 0.]
 [ 1.]]

In [55]: sigmaz() * spin2
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
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 $|\uparrow\rangle$ state of a two-level spin system while `spin2` gives the $|\downarrow\rangle$ 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 $|\uparrow\rangle$ state:

```
In [56]: sigmaz() * spin
Out[56]:
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 $|\downarrow\rangle$ 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 $|\uparrow\rangle$ state is label as $|0\rangle$, and the $|\downarrow\rangle$ state by $|1\rangle$.

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

3.3.5 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 [57]: vac = basis(5, 0)

In [58]: one = basis(5, 1)

In [59]: c = create(5)

In [60]: N = num(5)

In [61]: expect(N, vac)
Out[61]: 0.0

In [62]: expect(N, one)
\\\\\\\\\\\\\\Out[62]: 1.0
```

```
In [63]: coh = coherent_dm(5, 1.0j)

In [64]: expect(N, coh)
Out[64]: 0.9970555745806596
```

```
In [65]: cat = (basis(5, 4) + 1.0j * basis(5, 3)).unit()

In [66]: expect(c, cat)
Out[66]: 0.999999999999998j
```

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

```
In [67]: states = [(c**k * vac).unit() for k in range(5)]  # must normalize

In [68]: expect(N, states)
Out[68]: array([ 0.,  1.,  2.,  3.,  4.])
```

```
In [69]: cat_list = [(basis(5, 4) + x * basis(5, 3)).unit()
   ....:              for x in [0, 1.0j, -1.0, -1.0j]]
   ....:

In [70]: expect(c, cat_list)
Out[70]: 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 [71]: up = basis(2, 0)

In [72]: down = basis(2, 1)

In [73]: expect(sigmaz(), up)
Out[73]: 1.0

In [74]: expect(sigmaz(), down)
\\\\\\\\\\\\\\Out[74]: -1.0
```

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

```
In [75]: spin1 = basis(2, 0)

In [76]: spin2 = basis(2, 1)
```

```
In [77]: two_spins = tensor(spin1, spin2)

In [78]: sz1 = tensor(sigmaz(), qeye(2))

In [79]: sz2 = tensor(qeye(2), sigmaz())

In [80]: expect(sz1, two_spins)
Out[80]: 1.0

In [81]: expect(sz2, two_spins)
\\\\\\\\\\\\\Out[81]: -1.0
```

3.3.6 Superoperators and Vectorized Operators

In addition to state vectors and density operators, QuTiP allows for representing maps that act linearly on density operators using the Kraus, Liouville supermatrix and Choi matrix formalisms. This support is based on the correspondance between linear operators acting on a Hilbert space, and vectors in two copies of that Hilbert space, vec $: \mathcal{L}(\mathcal{H}) \rightarrow \mathcal{H} \otimes \mathcal{H}$ *[Hav03]*, *[Wat13]*.

This isomorphism is implemented in QuTiP by the *operator_to_vector* and *vector_to_operator* functions:

```
In [82]: psi = basis(2, 0)

In [83]: rho = ket2dm(psi)

In [84]: rho
Out[84]:
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 1.  0.]
 [ 0.  0.]]

In [85]: vec_rho = operator_to_vector(rho)

In [86]: vec_rho
Out[86]:
Quantum object: dims = [[[2], [2]], [1]], shape = (4, 1), type = operator-ket
Qobj data =
[[ 1.]
 [ 0.]
 [ 0.]
 [ 0.]]

In [87]: rho2 = vector_to_operator(vec_rho)

In [88]: (rho - rho2).norm()
Out[88]: 0.0
```

The `type` attribute indicates whether a quantum object is a vector corresponding to an operator (`operator-ket`), or its Hermitian conjugate (`operator-bra`).

Note that QuTiP uses the *column-stacking* convention for the isomorphism between $\mathcal{L}(\mathcal{H})$ and $\mathcal{H} \otimes \mathcal{H}$:

```
In [89]: import numpy as np

In [90]: A = Qobj(np.arange(4).reshape((2, 2)))

In [91]: A
Out[91]:
```

```
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = False
Qobj data =
[[ 0.  1.]
 [ 2.  3.]]

In [92]: operator_to_vector(A)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
Quantum object: dims = [[[2], [2]], [1]], shape = (4, 1), type = operator-ket
Qobj data =
[[ 0.]
 [ 2.]
 [ 1.]
 [ 3.]]
```

Since $\mathcal{H} \otimes \mathcal{H}$ is a vector space, linear maps on this space can be represented as matrices, often called *super-operators*. Using the `Qobj`, the `spre` and `spost` functions, supermatrices corresponding to left- and right-multiplication respectively can be quickly constructed.

```
In [93]: X = sigmax()

In [94]: S = spre(X) * spost(X.dag()) # Represents conjugation by X.
```

Note that this is done automatically by the `to_super` function when given `type='oper'` input.

```
In [95]: S2 = to_super(X)

In [96]: (S - S2).norm()
Out[96]: 0.0
```

Quantum objects representing superoperators are denoted by `type='super'`:

```
In [97]: S
Out[97]:
Quantum object: dims = [[[2], [2]], [[2], [2]]], shape = (4, 4), type = super,
↪isherm = True
Qobj data =
[[ 0.  0.  0.  1.]
 [ 0.  0.  1.  0.]
 [ 0.  1.  0.  0.]
 [ 1.  0.  0.  0.]]
```

Information about superoperators, such as whether they represent completely positive maps, is exposed through the `iscp`, `istp` and `iscptp` attributes:

```
In [98]: S.iscp, S.istp, S.iscptp
Out[98]: (True, True, True)
```

In addition, dynamical generators on this extended space, often called *Liouvillian superoperators*, can be created using the `liouvillian` function. Each of these takes a Hamilonian along with a list of collapse operators, and returns a `type="super"` object that can be exponentiated to find the superoperator for that evolution.

```
In [99]: H = 10 * sigmaz()

In [100]: c1 = destroy(2)

In [101]: L = liouvillian(H, [c1])

In [102]: L
Out[102]:
Quantum object: dims = [[[2], [2]], [[2], [2]]], shape = (4, 4), type = super,
↪isherm = False
```

```
Qobj data =
[[ 0.0 +0.j   0.0 +0.j   0.0 +0.j   1.0 +0.j]
 [ 0.0 +0.j  -0.5+20.j   0.0 +0.j   0.0 +0.j]
 [ 0.0 +0.j   0.0 +0.j  -0.5-20.j   0.0 +0.j]
 [ 0.0 +0.j   0.0 +0.j   0.0 +0.j  -1.0 +0.j]]

In [103]: S = (12 * L).expm()
```

For qubits, a particularly useful way to visualize superoperators is to plot them in the Pauli basis, such that $S_{\mu,\nu} = \langle\langle\sigma_\mu|S[\sigma_\nu]\rangle\rangle$. Because the Pauli basis is Hermitian, $S_{\mu,\nu}$ is a real number for all Hermitian-preserving superoperators S, allowing us to plot the elements of S as a Hinton diagram. In such diagrams, positive elements are indicated by white squares, and negative elements by black squares. The size of each element is indicated by the size of the corresponding square. For instance, let $S[\rho] = \sigma_x\rho\sigma_x^\dagger$. Then $S[\sigma_\mu] = \sigma_\mu \cdot \begin{cases} +1 & \mu = 0, x \\ -1 & \mu = y, z \end{cases}$. We can quickly see this by noting that the Y and Z elements of the Hinton diagram for S are negative:
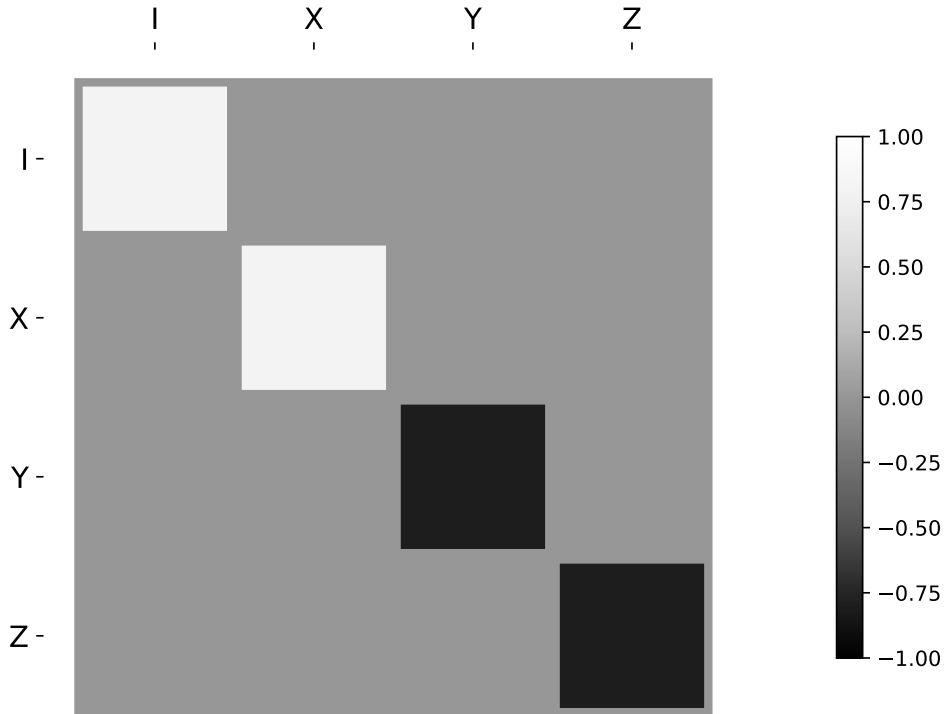
3.3.7 Choi, Kraus, Stinespring and χ Representations

In addition to the superoperator representation of quantum maps, QuTiP supports several other useful representations. First, the Choi matrix $J(\Lambda)$ of a quantum map Λ is useful for working with ancilla-assisted process tomography (AAPT), and for reasoning about properties of a map or channel. Up to normalization, the Choi matrix is defined by acting Λ on half of an entangled pair. In the column-stacking convention,

$$J(\Lambda) = (\mathbb{1} \otimes \Lambda)[|\mathbb{1}\rangle\rangle \langle\langle\mathbb{1}|].$$

In QuTiP, $J(\Lambda)$ can be found by calling the `to_choi` function on a `type="super"` Qobj.

```
In [104]: X = sigmax()

In [105]: S = sprepost(X, X)

In [106]: J = to_choi(S)

In [107]: print(J)
Quantum object: dims = [[[2], [2]], [[2], [2]]], shape = (4, 4), type = super,
→isherm = True, superrep = choi
Qobj data =
[[ 0.  0.  0.  0.]
 [ 0.  1.  1.  0.]
 [ 0.  1.  1.  0.]
 [ 0.  0.  0.  0.]]

In [108]: print(to_choi(spre(qeye(2))))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
→object: dims = [[[2], [2]], [[2], [2]]], shape = (4, 4), type = super, isherm =
→True, superrep = choi
Qobj data =
[[ 1.  0.  0.  1.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 1.  0.  0.  1.]]
```

If a Qobj instance is already in the Choi `superrep`, then calling `to_choi` does nothing:

```
In [109]: print(to_choi(J))
Quantum object: dims = [[[2], [2]], [[2], [2]]], shape = (4, 4), type = super,
→isherm = True, superrep = choi
Qobj data =
[[ 0.  0.  0.  0.]
 [ 0.  1.  1.  0.]
 [ 0.  1.  1.  0.]
 [ 0.  0.  0.  0.]]
```

To get back to the superoperator representation, simply use the `to_super` function. As with `to_choi`, `to_super` is idempotent:

```
In [110]: print(to_super(J) - S)
Quantum object: dims = [[[2], [2]], [[2], [2]]], shape = (4, 4), type = super,
→isherm = True
Qobj data =
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]

In [111]: print(to_super(S))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
→object: dims = [[[2], [2]], [[2], [2]]], shape = (4, 4), type = super, isherm =
→True
Qobj data =
[[ 0.  0.  0.  1.]
 [ 0.  0.  1.  0.]
 [ 0.  1.  0.  0.]
 [ 1.  0.  0.  0.]]
```

We can quickly obtain another useful representation from the Choi matrix by taking its eigendecomposition. In particular, let $\{A_i\}$ be a set of operators such that $J(\Lambda) = \sum_i |A_i\rangle\!\rangle\langle\!\langle A_i|$. We can write $J(\Lambda)$ in this way for any hermicity-preserving map; that is, for any map Λ such that $J(\Lambda) = J^\dagger(\Lambda)$. These operators then form the Kraus

representation of Λ. In particular, for any input ρ,

$$\Lambda(\rho) = \sum_i A_i \rho A_i^\dagger.$$

Notice using the column-stacking identity that $(C^{\mathrm{T}} \otimes A)|B\rangle\rangle = |ABC\rangle\rangle$, we have that

$$\sum_i (1\!\!\!/ \otimes A_i)(1\!\!\!/ \otimes A_i)^\dagger |1\!\!\!/\rangle\rangle\langle\langle 1\!\!\!/| = \sum_i |A_i\rangle\rangle\langle\langle A_i| = J(\Lambda).$$

The Kraus representation of a hermicity-preserving map can be found in QuTiP using the `to_kraus` function.

```
In [112]: I, X, Y, Z = qeye(2), sigmax(), sigmay(), sigmaz()

In [113]: S = sum(sprepost(P, P) for P in (I, X, Y, Z)) / 4
   .....: print(S)
   .....:
Quantum object: dims = [[[2], [2]], [[2], [2]]], shape = (4, 4), type = super,␣
→isherm = True
Qobj data =
[[ 0.5  0.   0.   0.5]
 [ 0.   0.   0.   0. ]
 [ 0.   0.   0.   0. ]
 [ 0.5  0.   0.   0.5]]

In [115]: J = to_choi(S)
   .....: print(J)
   .....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
→object: dims = [[[2], [2]], [[2], [2]]], shape = (4, 4), type = super, isherm =␣
→True, superrep = choi
Qobj data =
[[ 0.5  0.   0.   0. ]
 [ 0.   0.5  0.   0. ]
 [ 0.   0.   0.5  0. ]
 [ 0.   0.   0.   0.5]]

In [117]: print(J.eigenstates()[1])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
→Quantum object: dims = [[[2], [2]], [1, 1]], shape = (4, 1), type = operator-ket
Qobj data =
[[ 1.]
 [ 0.]
 [ 0.]
 [ 0.]]
 Quantum object: dims = [[[2], [2]], [1, 1]], shape = (4, 1), type = operator-ket
Qobj data =
[[ 0.]
 [ 1.]
 [ 0.]
 [ 0.]]
 Quantum object: dims = [[[2], [2]], [1, 1]], shape = (4, 1), type = operator-ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 1.]
 [ 0.]]
 Quantum object: dims = [[[2], [2]], [1, 1]], shape = (4, 1), type = operator-ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 0.]
 [ 1.]]]
```

```
In [118]: K = to_kraus(S)
   .....: print(K)
   .....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 0.70710678  0.        ]
 [ 0.          0.        ]], Quantum object: dims = [[2], [2]], shape = (2, 2),␣
↪type = oper, isherm = False
Qobj data =
[[ 0.          0.        ]
 [ 0.70710678  0.        ]], Quantum object: dims = [[2], [2]], shape = (2, 2),␣
↪type = oper, isherm = False
Qobj data =
[[ 0.          0.70710678]
 [ 0.          0.        ]], Quantum object: dims = [[2], [2]], shape = (2, 2),␣
↪type = oper, isherm = True
Qobj data =
[[ 0.          0.        ]
 [ 0.          0.70710678]]]
```

As with the other representation conversion functions, `to_kraus` checks the `superrep` attribute of its input, and chooses an appropriate conversion method. Thus, in the above example, we can also call `to_kraus` on J.

```
In [120]: KJ = to_kraus(J)
   .....: print(KJ)
   .....:
[Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 0.70710678  0.        ]
 [ 0.          0.        ]], Quantum object: dims = [[2], [2]], shape = (2, 2),␣
↪type = oper, isherm = False
Qobj data =
[[ 0.          0.        ]
 [ 0.70710678  0.        ]], Quantum object: dims = [[2], [2]], shape = (2, 2),␣
↪type = oper, isherm = False
Qobj data =
[[ 0.          0.70710678]
 [ 0.          0.        ]], Quantum object: dims = [[2], [2]], shape = (2, 2),␣
↪type = oper, isherm = True
Qobj data =
[[ 0.          0.        ]
 [ 0.          0.70710678]]]

In [122]: for A, AJ in zip(K, KJ):
   .....:     print(A - AJ)
   .....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 0.  0.]
 [ 0.  0.]]
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 0.  0.]
 [ 0.  0.]]
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 0.  0.]
 [ 0.  0.]]
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
```

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

The Stinespring representation is closely related to the Kraus representation, and consists of a pair of operators A and B such that for all operators X acting on $\mathcal{H}$,

$$\Lambda(X) = \text{Tr}_2(AXB^\dagger),$$

where the partial trace is over a new index that corresponds to the index in the Kraus summation. Conversion to Stinespring is handled by the `to_stinespring` function.

```
In [123]: a = create(2).dag()

In [124]: S_ad = sprepost(a * a.dag(), a * a.dag()) + sprepost(a, a.dag())
   .....: S = 0.9 * sprepost(I, I) + 0.1 * S_ad
   .....: print(S)
   .....:
Quantum object: dims = [[[2], [2]], [[2], [2]]], shape = (4, 4), type = super,␣
→isherm = False
Qobj data =
[[ 1.   0.   0.   0.1]
 [ 0.   0.9  0.   0. ]
 [ 0.   0.   0.9  0. ]
 [ 0.   0.   0.   0.9]]

In [127]: A, B = to_stinespring(S)
   .....: print(A)
   .....: print(B)
   .....:
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
→object: dims = [[2, 3], [2]], shape = (6, 2), type = other
Qobj data =
[[-0.98845443  0.        ]
 [ 0.          0.31622777]
 [ 0.15151842  0.        ]
 [ 0.         -0.93506452]
 [ 0.          0.        ]
 [ 0.         -0.16016975]]
Quantum object: dims = [[2, 3], [2]], shape = (6, 2), type = other
Qobj data =
[[-0.98845443  0.        ]
 [ 0.          0.31622777]
 [ 0.15151842  0.        ]
 [ 0.         -0.93506452]
 [ 0.          0.        ]
 [ 0.         -0.16016975]]
```

Notice that a new index has been added, such that A and B have dimensions `[[2, 3], [2]]`, with the length-3 index representing the fact that the Choi matrix is rank-3 (alternatively, that the map has three Kraus operators).

```
In [130]: to_kraus(S)
Out[130]:
[Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
 Qobj data =
 [[ 0.98845443  0.        ]
  [ 0.          0.93506452]],
 Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
 Qobj data =
 [[-0.15151842  0.        ]
  [ 0.          0.16016975]],
 Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
 Qobj data =
```

```
 [[ 0.   0.]
  [ 0.   0.]],
 Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = False
 Qobj data =
 [[ 0.          0.31622777]
  [ 0.          0.         ]]]

In [131]: print(to_choi(S).eigenenergies())
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪0.          0.04861218  0.1          1.85138782]
```

Finally, the last superoperator representation supported by QuTiP is the χ-matrix representation,

$$\Lambda(\rho) = \sum_{\alpha,\beta} \chi_{\alpha,\beta} B_\alpha \rho B_\beta^\dagger,$$

where $\{B_\alpha\}$ is a basis for the space of matrices acting on $\mathcal{H}$. In QuTiP, this basis is taken to be the Pauli basis $B_\alpha = \sigma_\alpha/\sqrt{2}$. Conversion to the χ formalism is handled by the `to_chi` function.

```
In [132]: chi = to_chi(S)
   .....: print(chi)
   .....:
Quantum object: dims = [[[2], [2]], [[2], [2]]], shape = (4, 4), type = super,
↪isherm = True, superrep = chi
Qobj data =
[[ 3.7+0.j   0.0+0.j   0.0+0.j   0.1+0.j ]
 [ 0.0+0.j   0.1+0.j   0.0+0.1j  0.0+0.j ]
 [ 0.0+0.j   0.0-0.1j  0.1+0.j   0.0+0.j ]
 [ 0.1+0.j   0.0+0.j   0.0+0.j   0.1+0.j ]]
```

One convenient property of the χ matrix is that the average gate fidelity with the identity map can be read off directly from the χ_{00} element:

```
In [134]: print(average_gate_fidelity(S))
0.95

In [135]: print(chi[0, 0] / 4)
\\\\\(0.925+0j)
```

Here, the factor of 4 comes from the dimension of the underlying Hilbert space $\mathcal{H}$. As with the superoperator and Choi representations, the χ representation is denoted by the `superrep`, such that *`to_super`*, *`to_choi`*, *`to_kraus`*, `to_stinespring` and `to_chi` all convert from the χ representation appropriately.

3.3.8 Properties of Quantum Maps

In addition to converting between the different representations of quantum maps, QuTiP also provides attributes to make it easy to check if a map is completely positive, trace preserving and/or hermicity preserving. Each of these attributes uses `superrep` to automatically perform any needed conversions.

In particular, a quantum map is said to be positive (but not necessarily completely positive) if it maps all positive operators to positive operators. For instance, the transpose map $\Lambda(\rho) = \rho^\mathrm{T}$ is a positive map. We run into problems, however, if we tensor Λ with the identity to get a partial transpose map.

```
In [136]: rho = ket2dm(bell_state())

In [137]: rho_out = partial_transpose(rho, [0, 1])
   .....: print(rho_out.eigenenergies())
   .....:
[-0.5  0.5  0.5  0.5]
```

Notice that even though we started with a positive map, we got an operator out with negative eigenvalues. Complete positivity addresses this by requiring that a map returns positive operators for all positive operators, and does so even under tensoring with another map. The Choi matrix is very useful here, as it can be shown that a map is completely positive if and only if its Choi matrix is positive *[Wat13]*. QuTiP implements this check with the `iscp` attribute. As an example, notice that the snippet above already calculates the Choi matrix of the transpose map by acting it on half of an entangled pair. We simply need to manually set the `dims` and `superrep` attributes to reflect the structure of the underlying Hilbert space and the chosen representation.

```
In [139]: J = rho_out

In [140]: J.dims = [[[2], [2]], [[2], [2]]]
    .....: J.superrep = 'choi'
    .....:

In [142]: print(J.iscp)
False
```

This confirms that the transpose map is not completely positive. On the other hand, the transpose map does satisfy a weaker condition, namely that it is hermicity preserving. That is, $\Lambda(\rho) = (\Lambda(\rho))^\dagger$ for all ρ such that $\rho = \rho^\dagger$. To see this, we note that $(\rho^T)^\dagger = \rho^*$, the complex conjugate of ρ. By assumption, $\rho = \rho^\dagger = (\rho^*)^T$, though, such that $\Lambda(\rho) = \Lambda(\rho^\dagger) = \rho^*$. We can confirm this by checking the `ishp` attribute:

```
In [143]: print(J.ishp)
True
```

Next, we note that the transpose map does preserve the trace of its inputs, such that $\text{Tr}(\Lambda[\rho]) = \text{Tr}(\rho)$ for all ρ. This can be confirmed by the `istp` attribute:

```
In [144]: print(J.ishp)
True
```

Finally, a map is called a quantum channel if it always maps valid states to valid states. Formally, a map is a channel if it is both completely positive and trace preserving. Thus, QuTiP provides a single attribute to quickly check that this is true.

```
In [145]: print(J.iscptp)
False

In [146]: print(to_super(qeye(2)).iscptp)
\\\\\\True
```

3.4 Using Tensor Products and Partial Traces

3.4.1 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 [2]: tensor([basis(2, 0), basis(2, 0)])
Out[2]:
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 [3]: tensor((basis(2, 0) + basis(2, 1)).unit(),
   ...:         (basis(2, 0) + basis(2, 1)).unit(), basis(2, 0))
   ...:
Out[3]:
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 [4]: tensor(sigmax(), sigmax())
Out[4]:
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 [5]: tensor(sigmaz(), identity(2))
Out[5]:
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.]]
```

3.4.2 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 [6]: H = tensor(sigmaz(), identity(2)) + tensor(identity(2),
   ...:          sigmaz()) + 0.05 * tensor(sigmax(), sigmax())
   ...:

In [7]: H
Out[7]:
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 [8]: H = (tensor(sigmaz(), identity(2), identity(2)) +
   ...:      tensor(identity(2), sigmaz(), identity(2)) +
   ...:      tensor(identity(2), identity(2), sigmaz()) +
   ...:      0.5 * tensor(sigmax(), sigmax(), identity(2)) +
   ...:      0.25 * tensor(identity(2), sigmax(), sigmax()))
   ...:

In [9]: H
Out[9]:
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:

```
In [10]: N = 10

In [11]: omega_a = 1.0

In [12]: omega_c = 1.25

In [13]: g = 0.05

In [14]: a = tensor(identity(2), destroy(N))

In [15]: sm = tensor(destroy(2), identity(N))

In [16]: sz = tensor(sigmaz(), identity(N))

In [17]: 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.

3.4.3 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 [18]:    psi = tensor(basis(2, 0), basis(2, 1))

In [19]:    psi.ptrace(0)
Out[19]:
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 1.  0.]
 [ 0.  0.]]

In [20]:    psi.ptrace(1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
 ↪
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 [21]:    psi = tensor((basis(2, 0) + basis(2, 1)).unit(), basis(2, 0))

In [22]:    psi
Out[22]:
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.70710678]
 [ 0.        ]
 [ 0.70710678]
 [ 0.        ]]

In [23]:    psi.ptrace(0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 0.5  0.5]
 [ 0.5  0.5]]

In [24]:    rho = tensor(ket2dm((basis(2, 0) + basis(2, 1)).unit()), fock_dm(2, 0))

In [25]:    rho
Out[25]:
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 [26]:    rho.ptrace(0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm = True
Qobj data =
[[ 0.5  0.5]
 [ 0.5  0.5]]
```

3.4.4 Superoperators and Tensor Manipulations

As described in *Superoperators and Vectorized Operators*, *superoperators* are operators that act on Liouville space, the vectorspace of linear operators. Superoperators can be represented using the isomorphism $\mathrm{vec} : \mathcal{L}(\mathcal{H}) \to \mathcal{H} \otimes \mathcal{H}$ *[Hav03]*, *[Wat13]*. To represent superoperators acting on $\mathcal{L}(\mathcal{H}_1 \otimes \mathcal{H}_2)$ thus takes some tensor rearrangement to get the desired ordering $\mathcal{H}_1 \otimes \mathcal{H}_2 \otimes \mathcal{H}_1 \otimes \mathcal{H}_2$.

In particular, this means that `qutip.tensor` does not act as one might expect on the results of `qutip.to_super`:

```
In [27]: A = qeye([2])

In [28]: B = qeye([3])

In [29]: to_super(tensor(A, B)).dims
Out[29]: [[[2, 3], [2, 3]], [[2, 3], [2, 3]]]

In [30]: tensor(to_super(A), to_super(B)).dims
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[30]: [[[2], [2], [3], [3]], [[2],
↪ [2], [3], [3]]]
```

In the former case, the result correctly has four copies of the compound index with dims `[2, 3]`. In the latter case, however, each of the Hilbert space indices is listed independently and in the wrong order.

The `qutip.super_tensor` function performs the needed rearrangement, providing the most direct analog to `qutip.tensor` on the underlying Hilbert space. In particular, for any two `type="oper"` Qobjs `A` and `B`, `to_super(tensor(A, B)) == super_tensor(to_super(A), to_super(B))` and `operator_to_vector(tensor(A, B)) == super_tensor(operator_to_vector(A), operator_to_vector(B))`. Returning to the previous example:

```
In [31]: super_tensor(to_super(A), to_super(B)).dims
Out[31]: [[[2, 3], [2, 3]], [[2, 3], [2, 3]]]
```

The `qutip.composite` function automatically switches between `qutip.tensor` and `qutip.super_tensor` based on the `type` of its arguments, such that `composite(A, B)` returns an appropriate Qobj to represent the composition of two systems.

```
In [32]: composite(A, B).dims
Out[32]: [[2, 3], [2, 3]]

In [33]: composite(to_super(A), to_super(B)).dims
\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[33]: [[[2, 3], [2, 3]], [[2, 3], [2, 3]]]
```

QuTiP also allows more general tensor manipulations that are useful for converting between superoperator representations [WBC11]. In particular, the `tensor_contract` function allows for contracting one or more pairs of indices. As detailed in the channel contraction tutorial, this can be used to find superoperators that represent partial trace maps. Using this functionality, we can construct some quite exotic maps, such as a map from 3×3 operators to 2×2 operators:

```
In [34]: tensor_contract(composite(to_super(A), to_super(B)), (1, 3), (4, 6)).dims
Out[34]: [[[2], [2]], [[3], [3]]]
```

3.5 Time Evolution and Quantum System Dynamics

3.5.1 Dynamics Simulation Results

Important: In QuTiP 2, the results from all of the dynamics solvers are returned as Odedata objects. This unified and significantly simplified postprocessing of simulation results from different solvers, compared to QuTiP 1. However, this change also results in the loss of backward compatibility with QuTiP version 1.x. In QuTiP 3, the Odedata class has been renamed to Result, but for backwards compatibility an alias between Result and Odedata is provided.

The solver.Result 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.solver.Result` class that stores all the crucial data needed for analyzing and plotting the results of a simulation. Like the `qutip.Qobj` class, the `Result` 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 `Result` object `result` contains the following properties for storing simulation data:

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

Accessing Result Data

To understand how to access the data in a Result object we will use an 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 Result object, here called simply `result`. To see what is contained inside `result` we can use the print function:

```
>>> print(result)
    Result 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 *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 = result.expect[0]
>>> expt1 = result.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 = result.times
```

we can plot the resulting expectation values:

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

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 Result Objects

The main advantage in using the Result 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(result, '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:

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

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

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

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

3.5.2 Lindblad Master Equation Solver

Unitary evolution

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

$$i\hbar \frac{\partial}{\partial t} \Psi = \hat{H} \Psi, \tag{3.1}$$

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

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

where $|\psi\rangle$ 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_ops` at the points in time in the list `times`, 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

```
In [1]: H = 2 * np.pi * 0.1 * sigmax()

In [2]: psi0 = basis(2, 0)

In [3]: times = np.linspace(0.0, 10.0, 20.0)

In [4]: result = mesolve(H, psi0, times, [], [sigmaz()])
```

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.solver.Result*, as described in the previous section *Dynamics 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 times, for each operator)

```
In [5]: result = mesolve(H, psi0, times, [], [sigmaz(), sigmay()])

In [6]: result.expect
Out[6]:
[array([ 1.        ,  0.78914057,  0.24548559, -0.40169513, -0.8794735 ,
        -0.98636142, -0.67728219, -0.08258023,  0.54694721,  0.94581685,
         0.94581769,  0.54694945, -0.08257765, -0.67728015, -0.98636097,
        -0.87947476, -0.40169736,  0.24548326,  0.78913896,  1.        ]),
 array([  0.00000000e+00,  -6.14212640e-01,  -9.69400240e-01,
         -9.15773457e-01,  -4.75947849e-01,   1.64593874e-01,
          7.35723339e-01,   9.96584419e-01,   8.37167094e-01,
          3.24700624e-01,  -3.24698160e-01,  -8.37165632e-01,
         -9.96584633e-01,  -7.35725221e-01,  -1.64596567e-01,
          4.75945525e-01,   9.15772479e-01,   9.69400830e-01,
          6.14214701e-01,   2.77159958e-06])]
```

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

```
In [7]: H = 2 * np.pi * 0.1 * sigmax()

In [8]: psi0 = basis(2, 0)

In [9]: times = np.linspace(0.0, 10.0, 100)

In [10]: result = mesolve(H, psi0, times, [], [sigmaz(), sigmay()])

In [11]: fig, ax = subplots()

In [12]: ax.plot(result.times, result.expect[0]);

In [13]: ax.plot(result.times, result.expect[1]);

In [14]: ax.set_xlabel('Time');

In [15]: ax.set_ylabel('Expectation values');

In [16]: ax.legend(("Sigma-Z", "Sigma-Y"));
```

```
In [17]: show()
```
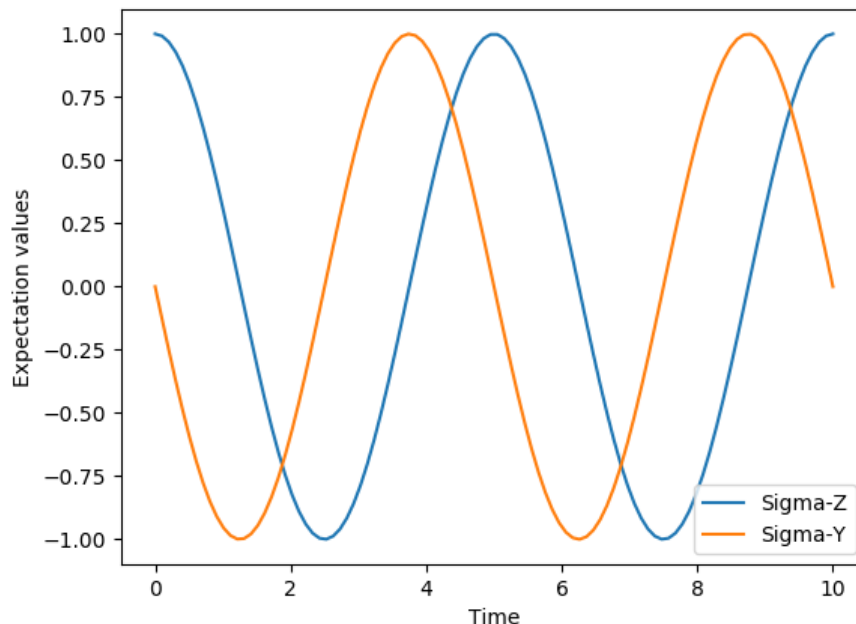

If an empty list of operators is passed as fifth parameter, the `qutip.mesolve` function returns a `qutip.solver.Result` instance that contains a list of state vectors for the times specified in `times`

```
In [18]: times = [0.0, 1.0]

In [19]: result = mesolve(H, psi0, times, [], [])

In [20]: result.states
Out[20]:
[Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
 Qobj data =
 [[ 1.]
  [ 0.]], Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
 Qobj data =
 [[ 0.80901699+0.j         ]
  [ 0.00000000-0.58778526j]]]
```

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

$$\dot{\rho}_{\text{tot}}(t) = -\frac{i}{\hbar}[H_{\text{tot}}, \rho_{\text{tot}}(t)], \tag{3.2}$$

the equivalent of the Schrödinger equation (3.1) in the density matrix formalism. Here, the total Hamiltonian

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

includes the original system Hamiltonian H_{sys}, the Hamiltonian for the environment H_{env}, and a term representing the interaction between the system and its environment H_{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. (3.2), 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 $\rho = \text{Tr}_{\text{env}}[\rho_{\text{tot}}]$

$$\dot{\rho}(t) = -\frac{i}{\hbar}[H(t), \rho(t)] + \sum_n \frac{1}{2}\left[2C_n\rho(t)C_n^+ - \rho(t)C_n^+C_n - C_n^+C_n\rho(t)\right] \tag{3.3}$$

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_{int}, and γ_n are the corresponding rates. The derivation of Eq. (3.3) 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. (3.3) 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 $\rho_{\text{tot}}^I(0) = \rho^I(0) \otimes \rho_{\text{env}}^I(0)$.

- **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, $\rho_{\text{tot}}(t) \approx \rho(t) \otimes \rho_{\text{env}}$.

- **Markov approximation** The time-scale of decay for the environment τ_{env} is much shorter than the smallest time-scale of the system dynamics $\tau_{\text{sys}} \gg \tau_{\text{env}}$. 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 $|\omega_{ab} - \omega_{cd}| \ll 1/\tau_{\text{sys}}$, 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 (3.3) which is used in `qutip.mesolve`.

For systems with environments satisfying the conditions outlined above, the Lindblad master equation (3.3) 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_ops`) 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_ops` 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 `np.sqrt(0.05) * sigmax()` to the previously empty list in the fourth parameter to the *qutip.mesolve* function:

```
In [21]: times = np.linspace(0.0, 10.0, 100)

In [22]: result = mesolve(H, psi0, times, [np.sqrt(0.05) * sigmax()], [sigmaz(),
→sigmay()])

In [23]: fig, ax = subplots()

In [24]: ax.plot(times, result.expect[0]);

In [25]: ax.plot(times, result.expect[1]);

In [26]: ax.set_xlabel('Time');

In [27]: ax.set_ylabel('Expectation values');

In [28]: ax.legend(("Sigma-Z", "Sigma-Y"));

In [29]: show(fig)
```
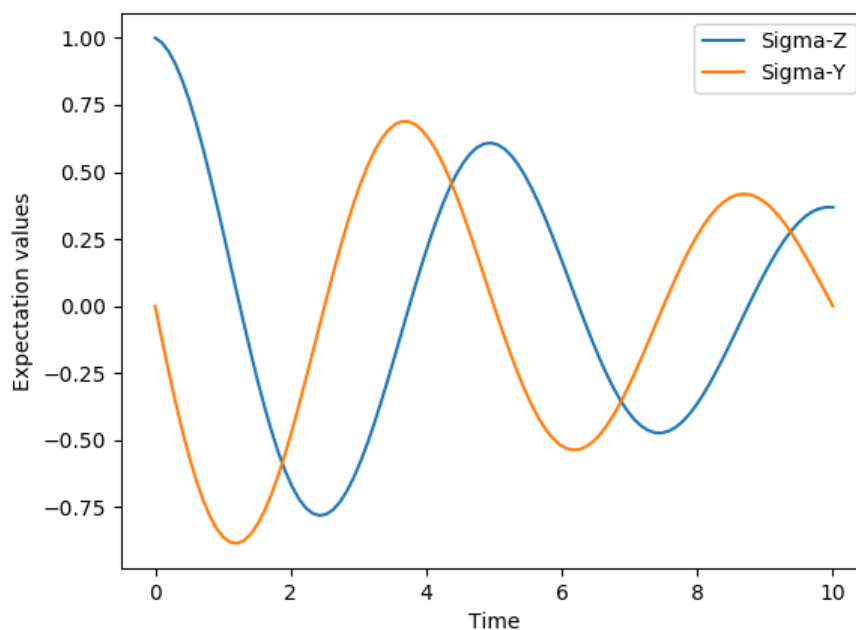

Here, 0.05 is the rate and the operator σ_x (*qutip.operators.sigmax*) describes the dissipation process.

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

```
In [30]: times = np.linspace(0.0, 10.0, 200)

In [31]: psi0 = tensor(fock(2,0), fock(10, 5))
```

```
In [32]: a  = tensor(qeye(2), destroy(10))

In [33]: sm = tensor(destroy(2), qeye(10))

In [34]: H = 2 * np.pi * a.dag() * a + 2 * np.pi * sm.dag() * sm + \
    ....: 2 * np.pi * 0.25 * (sm * a.dag() + sm.dag() * a)
    ....:

In [35]: result = mesolve(H, psi0, times, [np.sqrt(0.1)*a], [a.dag()*a, sm.
→dag()*sm])

In [36]: figure()
Out[36]: <matplotlib.figure.Figure at 0x2b4e7e783a90>

In [37]: plot(times, result.expect[0])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[37]: [<matplotlib.lines.
→Line2D at 0x2b4e7e79c9e8>]

In [38]: plot(times, result.expect[1])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
→[<matplotlib.lines.Line2D at 0x2b4e7e79cc88>]

In [39]: xlabel('Time')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
→<matplotlib.text.Text at 0x2b4e7e858198>

In [40]: ylabel('Expectation values')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
→<matplotlib.text.Text at 0x2b4e7e7e1710>

In [41]: legend(("cavity photon number", "atom excitation probability"))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
→<matplotlib.legend.Legend at 0x2b4e7e7a9438>

In [42]: show()
```
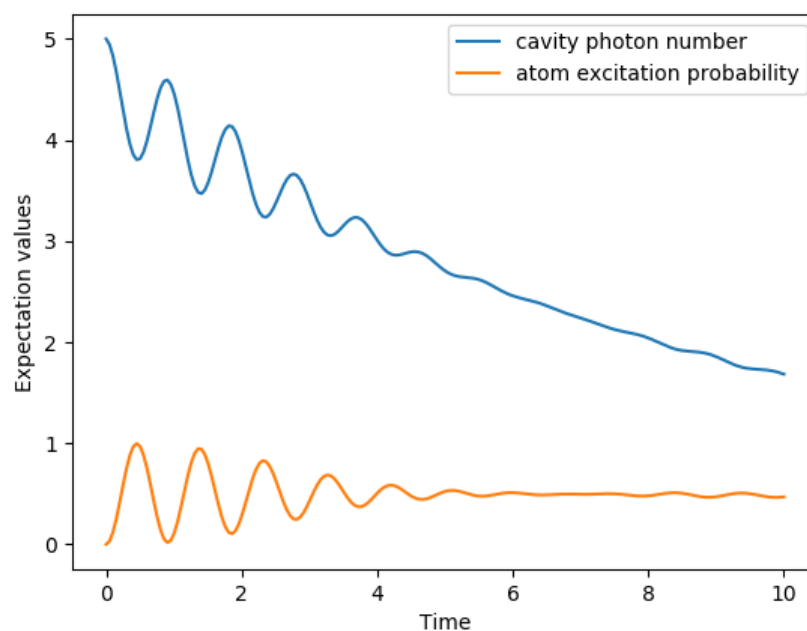
3.5.3 Monte Carlo Solver

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

$$H_{\text{eff}} = H_{\text{sys}} - \frac{i\hbar}{2} \sum_i C_n^+ C_n, \tag{3.4}$$

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. (3.4) 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

$$\delta p = \delta t \sum_n \langle \psi(t) | C_n^+ C_n | \psi(t) \rangle, \tag{3.5}$$

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. (3.5). 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

$$|\psi(t + \delta t)\rangle = C_n |\psi(t)\rangle / \langle \psi(t) | C_n^+ C_n | \psi(t) \rangle^{1/2}. \tag{3.6}$$

If more than a single collapse operator is present in Eq. (3.4), the probability of collapse due to the ith-operator C_i is given by

$$P_i(t) = \langle \psi(t) | C_i^+ C_i | \psi(t) \rangle / \delta p. \tag{3.7}$$

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 (3.4) 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. (3.6). The corresponding collapse operator C_n is chosen such that n is the smallest integer satisfying:

$$\sum_{i=1}^n P_n(\tau) \geq r \tag{3.8}$$

where the individual P_n are given by Eq. (3.7). Note that the left hand side of Eq. (3.8) 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`

```
In [1]: times = np.linspace(0.0, 10.0, 200)

In [2]: psi0 = tensor(fock(2, 0), fock(10, 5))

In [3]: a  = tensor(qeye(2), destroy(10))

In [4]: sm = tensor(destroy(2), qeye(10))

In [5]: H = 2 * np.pi * a.dag() * a + 2 * np.pi * sm.dag() * sm + 2 * np.pi * 0.25
   ↪* (sm * a.dag() + sm.dag() * a)

In [6]: data = mcsolve(H, psi0, times, [np.sqrt(0.1) * a], [a.dag() * a, sm.dag()
   ↪* sm])
10.0%. Run time:    0.58s. Est. time left: 00:00:00:05
20.0%. Run time:    1.12s. Est. time left: 00:00:00:04
30.0%. Run time:    1.65s. Est. time left: 00:00:00:03
40.0%. Run time:    2.18s. Est. time left: 00:00:00:03
50.0%. Run time:    2.70s. Est. time left: 00:00:00:02
60.0%. Run time:    3.26s. Est. time left: 00:00:00:02
70.0%. Run time:    3.79s. Est. time left: 00:00:00:01
80.0%. Run time:    4.31s. Est. time left: 00:00:00:01
90.0%. Run time:    4.83s. Est. time left: 00:00:00:00
100.0%. Run time:   5.41s. Est. time left: 00:00:00:00
Total run time:   5.43s

In [7]: figure()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
   ↪<matplotlib.figure.Figure at 0x2b4e7e75ce10>

In [8]: plot(times, data.expect[0], times, data.expect[1])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
   ↪
[<matplotlib.lines.Line2D at 0x2b4e7d335630>,
 <matplotlib.lines.Line2D at 0x2b4e7d710128>]

In [9]: title('Monte Carlo time evolution')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
   ↪<matplotlib.text.Text at 0x2b4e7d9bcf60>

In [10]: xlabel('Time')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
   ↪<matplotlib.text.Text at 0x2b4e7d47a8d0>

In [11]: ylabel('Expectation values')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
   ↪<matplotlib.text.Text at 0x2b4e7d6a1fd0>

In [12]: legend(("cavity photon number", "atom excitation probability"))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
   ↪<matplotlib.legend.Legend at 0x2b4e7eaf0ac8>

In [13]: show()
```
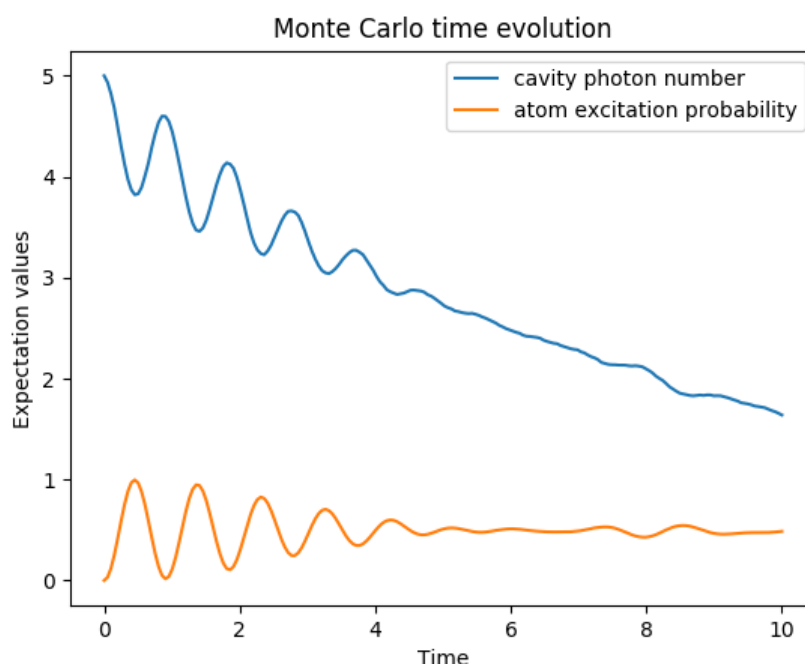
The advantage of the Monte Carlo method over the master equation approach is that only the state vector is required to be kept in the computers memory, as opposed to the entire density matrix. For large quantum system this becomes a significant advantage, and the Monte Carlo solver is therefore generally 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 *qutip.solver.Result* 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.solver.Result* 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 `times`. Furthermore, the output *qutip.solver.Result* 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.

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:

```
In [14]: data = mcsolve(H, psi0, times, [np.sqrt(0.1) * a], [a.dag() * a, sm.dag()
↪* sm], ntraj=1000)
10.0%. Run time:   1.45s. Est. time left: 00:00:00:13
20.0%. Run time:   2.89s. Est. time left: 00:00:00:11
30.0%. Run time:   4.34s. Est. time left: 00:00:00:10
40.0%. Run time:   5.78s. Est. time left: 00:00:00:08
50.0%. Run time:   7.21s. Est. time left: 00:00:00:07
60.0%. Run time:   8.64s. Est. time left: 00:00:00:05
```

```
70.0%. Run time:   10.17s. Est. time left: 00:00:00:04
80.0%. Run time:   11.59s. Est. time left: 00:00:00:02
90.0%. Run time:   13.06s. Est. time left: 00:00:00:01
100.0%. Run time:   14.45s. Est. time left: 00:00:00:00
Total run time:   14.55s
```

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:

```
In [15]: 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 *qutip.solver.Result* object returned by `mcsolve`. In the above example using:

```
In [16]: data = mcsolve(H, psi0, times, [np.sqrt(0.1) * a], [a.dag() * a, sm.dag()␣
→* sm], ntraj=[1, 10, 100, 1000])
10.0%. Run time:    1.51s. Est. time left: 00:00:00:13
20.0%. Run time:    2.97s. Est. time left: 00:00:00:11
30.0%. Run time:    4.43s. Est. time left: 00:00:00:10
40.0%. Run time:    5.93s. Est. time left: 00:00:00:08
50.0%. Run time:    7.39s. Est. time left: 00:00:00:07
60.0%. Run time:    8.85s. Est. time left: 00:00:00:05
70.0%. Run time:   10.33s. Est. time left: 00:00:00:04
80.0%. Run time:   11.77s. Est. time left: 00:00:00:02
90.0%. Run time:   13.22s. Est. time left: 00:00:00:01
100.0%. Run time:   14.66s. Est. time left: 00:00:00:00
Total run time:   14.75s
```

we can extract the relevant expectation values using:

```
In [17]: expt10 = data.expect[1]     # <- expectation values avg. over 10␣
→trajectories

In [18]: expt100 = data.expect[2]     # <- expectation  values avg. over 100␣
→trajectories

In [19]: expt1000 = data.expect[3]     # <- expectation values avg. over 1000␣
→trajectories
```

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

```
In [20]: options = Options(rhs_reuse=True)
```

A full account of this feature is given in *Setting Options for the Dynamics 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

```
In [21]: psi0 = tensor(fock(2, 0), fock(10, 5))

In [22]: a  = tensor(qeye(2), destroy(10))

In [23]: sm = tensor(destroy(2), qeye(10))

In [24]: H = 2 * np.pi * a.dag() * a + 2 * np.pi * sm.dag() * sm + \
    ....: 2 * np.pi * 0.25 * (sm * a.dag() + sm.dag() * a)
    ....:

In [25]: data1 = mcsolve(H, psi0, times, [np.sqrt(0.1) * a], [a.dag() * a, sm.
 →dag() * sm])
10.0%. Run time:    0.60s. Est. time left: 00:00:00:05
20.0%. Run time:    1.13s. Est. time left: 00:00:00:04
30.0%. Run time:    1.69s. Est. time left: 00:00:00:03
40.0%. Run time:    2.22s. Est. time left: 00:00:00:03
50.0%. Run time:    2.78s. Est. time left: 00:00:00:02
60.0%. Run time:    3.31s. Est. time left: 00:00:00:02
70.0%. Run time:    3.86s. Est. time left: 00:00:00:01
80.0%. Run time:    4.42s. Est. time left: 00:00:00:01
90.0%. Run time:    4.97s. Est. time left: 00:00:00:00
100.0%. Run time:    5.48s. Est. time left: 00:00:00:00
Total run time:    5.54s

In [26]: psi1 = tensor(fock(2, 0), coherent(10, 2 - 1j))

In [27]: opts = Options(rhs_reuse=True) # Run a second time, reusing RHS

In [28]: data2 = mcsolve(H, psi1, times, [np.sqrt(0.1) * a], [a.dag() * a, sm.
 →dag() * sm], options=opts)
10.0%. Run time:    1.42s. Est. time left: 00:00:00:12
20.0%. Run time:    2.74s. Est. time left: 00:00:00:10
30.0%. Run time:    4.08s. Est. time left: 00:00:00:09
40.0%. Run time:    5.36s. Est. time left: 00:00:00:08
50.0%. Run time:    6.74s. Est. time left: 00:00:00:06
60.0%. Run time:    8.04s. Est. time left: 00:00:00:05
70.0%. Run time:    9.33s. Est. time left: 00:00:00:03
80.0%. Run time:   10.66s. Est. time left: 00:00:00:02
90.0%. Run time:   11.99s. Est. time left: 00:00:00:01
100.0%. Run time:   13.35s. Est. time left: 00:00:00:00
Total run time:   13.45s

In [29]: figure()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
 →<matplotlib.figure.Figure at 0x2b4e7d242c18>

In [30]: plot(times, data1.expect[0], times, data1.expect[1], lw=2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
 →
[<matplotlib.lines.Line2D at 0x2b4e7cfb6898>,
 <matplotlib.lines.Line2D at 0x2b4e7cfb6ef0>]

In [31]: plot(times, data2.expect[0], '--', times, data2.expect[1], '--', lw=2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
 →
[<matplotlib.lines.Line2D at 0x2b4e7e9c64a8>,
```

```
 <matplotlib.lines.Line2D at 0x2b4e7d302b38>]

In [32]: title('Monte Carlo time evolution')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪<matplotlib.text.Text at 0x2b4e7e96c860>

In [33]: xlabel('Time', fontsize=14)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪<matplotlib.text.Text at 0x2b4e7d274d30>

In [34]: ylabel('Expectation values', fontsize=14)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪<matplotlib.text.Text at 0x2b4e7ee4ccc0>

In [35]: legend(("cavity photon number", "atom excitation probability"))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪<matplotlib.legend.Legend at 0x2b4e7d242e80>

In [36]: show()
```
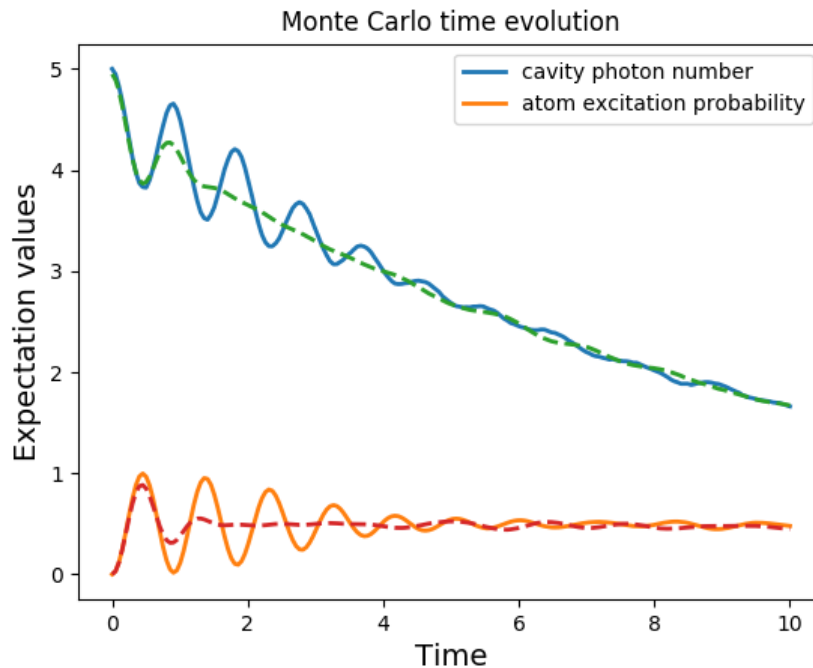

In addition to the initial state, one may reuse the Hamiltonian data when changing the number of trajectories `ntraj` or simulation times `times`. The reusing of Hamiltonian data is also supported for time-dependent Hamiltonians. See *Solving Problems with Time-dependent Hamiltonians* for further details.

3.5.4 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 time-evolutions solvers *qutip.mesolve*, *qutip.mcsolve*, *qutip.sesolve*, and `qutip.brmesolve` are all capable of handling time-dependent Hamiltonians and collapse terms. There are, in general, three different ways to implement time-dependent problems in QuTiP:

1. **Function based**: Hamiltonian / collapse operators expressed using [qobj, func] pairs, where the time-dependent coefficients of the Hamiltonian (or collapse operators) are expressed using 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 [This is also the only format that is supported in the `qutip.brmesolve` solver]. 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. Time-dependent coefficients using any of the following functions, or combinations thereof (including constants) can be compiled directly into C++-code:

```
'abs', 'acos', 'acosh', 'arg', 'asin', 'asinh', 'atan', 'atanh', 'conj',
'cos', 'cosh','exp', 'erf', 'imag', 'log', 'log10', 'norm', 'proj', 'real', 'sin',
→'sinh', 'sqrt',
'tan', 'tanh'
```

In addition, QuTiP supports cubic spline based interpolation functions [*Modeling Non-Analytic and/or Experimental Time-Dependent Parameters using Interpolating Functions*].

If you require mathematical functions other than those listed above, than it is possible to call any of the functions in the NumPy library using the prefix `np.` before the function name in the string, i.e `'np.sin(t)'`. This includes a wide range of functionality, but comes with a small overhead created by going from C++->Python->C++.

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 on the tutorials web page.

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_ops = [[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 an example 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

```python
In [1]: ustate = basis(3, 0)

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

In [3]: ground = basis(3, 2)

In [4]: N = 2 # Set where to truncate Fock state for cavity

In [5]: sigma_ge = tensor(qeye(N), ground * excited.dag())   # |g><e|

In [6]: sigma_ue = tensor(qeye(N), ustate * excited.dag())   # |u><e|

In [7]: a = tensor(destroy(N), qeye(3))

In [8]: ada = tensor(num(N), qeye(3))

In [9]: c_ops = []   # Build collapse operators

In [10]: kappa = 1.5 # Cavity decay rate

In [11]: c_ops.append(np.sqrt(kappa) * a)

In [12]: gamma = 6  # Atomic decay rate

In [13]: c_ops.append(np.sqrt(5*gamma/9) * sigma_ue) # Use Rb branching ratio of 5/
↪9 e->u

In [14]: c_ops.append(np.sqrt(4*gamma/9) * sigma_ge) # 4/9 e->g

In [15]: t = np.linspace(-15, 15, 100) # Define time vector

In [16]: psi0 = tensor(basis(N, 0), ustate) # Define initial state

In [17]: state_GG = tensor(basis(N, 1), ground) # Define states onto which to␣
↪project

In [18]: sigma_GG = state_GG * state_GG.dag()

In [19]: state_UU = tensor(basis(N, 0), ustate)

In [20]: sigma_UU = state_UU * state_UU.dag()

In [21]: g = 5   # coupling strength

In [22]: H0 = -g * (sigma_ge.dag() * a + a.dag() * sigma_ge)   # time-independent␣
↪term

In [23]: 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

---

3.5. Time Evolution and Quantum System Dynamics

```
In [24]: def H1_coeff(t, args):
    ....:         return 9 * np.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`)

```
In [25]: H = [H0,[H1,H1_coeff]]

In [26]: output = mesolve(H, psi0, t, c_ops, [ada, sigma_UU, sigma_GG])
```

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

```
In [27]: output = mcsolve(H, psi0, t, c_ops, [ada, sigma_UU, sigma_GG])
10.0%. Run time:   0.28s. Est. time left: 00:00:00:02
20.0%. Run time:   0.53s. Est. time left: 00:00:00:02
30.0%. Run time:   0.78s. Est. time left: 00:00:00:01
40.0%. Run time:   1.03s. Est. time left: 00:00:00:01
50.0%. Run time:   1.28s. Est. time left: 00:00:00:01
60.0%. Run time:   1.53s. Est. time left: 00:00:00:01
70.0%. Run time:   1.79s. Est. time left: 00:00:00:00
80.0%. Run time:   2.03s. Est. time left: 00:00:00:00
90.0%. Run time:   2.28s. Est. time left: 00:00:00:00
100.0%. Run time:   2.57s. Est. time left: 00:00:00:00
Total run time:   2.63s
```

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

```
In [28]: kappa = 0.5

In [29]: def col_coeff(t, args):  # coefficient function
    ....:         return np.sqrt(kappa * np.exp(-t))
    ....:

In [30]: N = 10  # number of basis states

In [31]: a = destroy(N)

In [32]: H = a.dag() * a  # simple HO

In [33]: psi0 = basis(N, 9)  # initial state

In [34]: c_ops = [[a, col_coeff]]  # time-dependent collapse term

In [35]: times = np.linspace(0, 10, 100)

In [36]: output = mesolve(H, psi0, times, c_ops, [a.dag() * a])
```

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

```
In [37]: def H1_coeff(t, args):
   ....:        return args['A'] * np.exp(-(t/args['sigma'])**2)
   ....:
```

or equivalently,

```
In [38]: def H1_coeff(t, args):
   ....:        A = args['A']
   ....:        sig = args['sigma']
   ....:        return A * np.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:

```
In [39]: output = mesolve(H, psi0, times, c_ops, [a.dag() * a], args={'A': 9,
→'sigma': 5})
```

or to keep things looking pretty

```
In [40]: args = {'A': 9, 'sigma': 5}

In [41]: output = mesolve(H, psi0, times, c_ops, [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.

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:

```
In [42]: 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:

```
In [43]: output = mesolve(H, psi0, t, c_ops, [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)'`

```
In [44]: H = [H0, [H1, 'A * exp(-(t / sig) ** 2)']]

In [45]: args = {'A': 9, 'sig': 5}
```

3.5. Time Evolution and Quantum System Dynamics   **59**

```
In [46]: output = mesolve(H, psi0, times, c_ops, [a.dag()*a], args=args)
```

Important: Naming your `args` variables `e`, `j` or `pi` will cause errors when using the string-based format.

Collapse operators are handled in the exact same way.

Modeling Non-Analytic and/or Experimental Time-Dependent Parameters using Interpolating Functions

Note: New in QuTiP 4.1

Sometimes it is necessary to model a system where the time-dependent parameters are non-analytic functions, or are derived from experimental data (i.e. a collection of data points). In these situations, one can use interpolating functions as an approximate functional form for input into a time-dependent solver. QuTiP includes it own custom cubic spline interpolation class *qutip.interpolate.Cubic_Spline* to provide this functionality. To see how this works, lets first generate some noisy data:

```
In [47]: t = np.linspace(-15, 15, 100)

In [48]: func = lambda t: 9*np.exp(-(t / 5)** 2)

In [49]: noisy_func = lambda t: func(t)+(0.05*func(t))*np.random.randn(t.shape[0])

In [50]: noisy_data = noisy_func(t)

In [51]: plt.figure()
Out[51]: <matplotlib.figure.Figure at 0x2b4e8082ebe0>

In [52]: plt.plot(t, func(t))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[52]: [<matplotlib.lines.
↪Line2D at 0x2b4e7f3962e8>]

In [53]: plt.plot(t, noisy_data, 'o')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[<matplotlib.lines.Line2D at 0x2b4e80838f98>]

In [54]: plt.show()
```
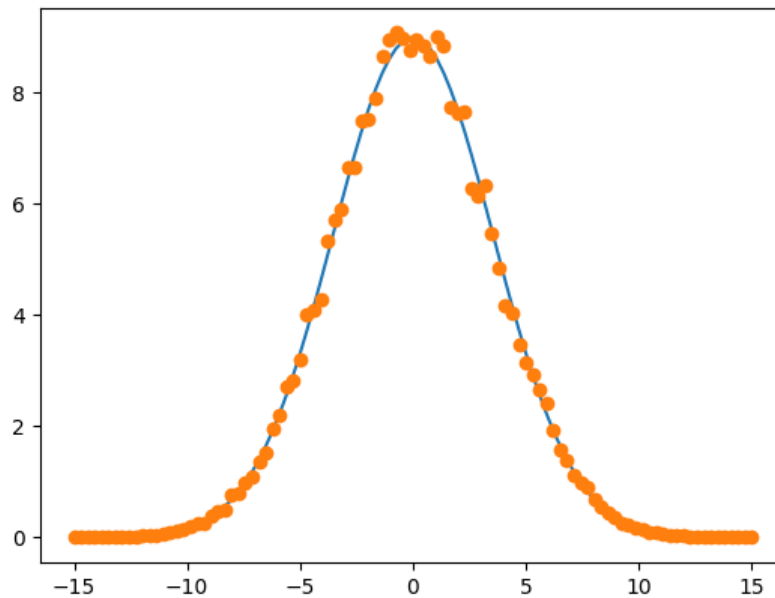
To turn these data points into a function we call the QuTiP *qutip.interpolate.Cubic_Spline* class using the first and last domain time points, `t[0]` and `t[-1]`, respectively, as well as the entire array of data points:

```
In [55]: S = Cubic_Spline(t[0], t[-1], noisy_data)

In [56]: plt.figure()
Out[56]: <matplotlib.figure.Figure at 0x2b4e8029b7f0>

In [57]: plt.plot(t, func(t))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[57]: [<matplotlib.lines.
→Line2D at 0x2b4e807ee208>]

In [58]: plt.plot(t, noisy_data, 'o')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
→[<matplotlib.lines.Line2D at 0x2b4e807ee400>]

In [59]: plt.plot(t, S(t), lw=2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
→[<matplotlib.lines.Line2D at 0x2b4e807cd588>]

In [60]: plt.show()
```
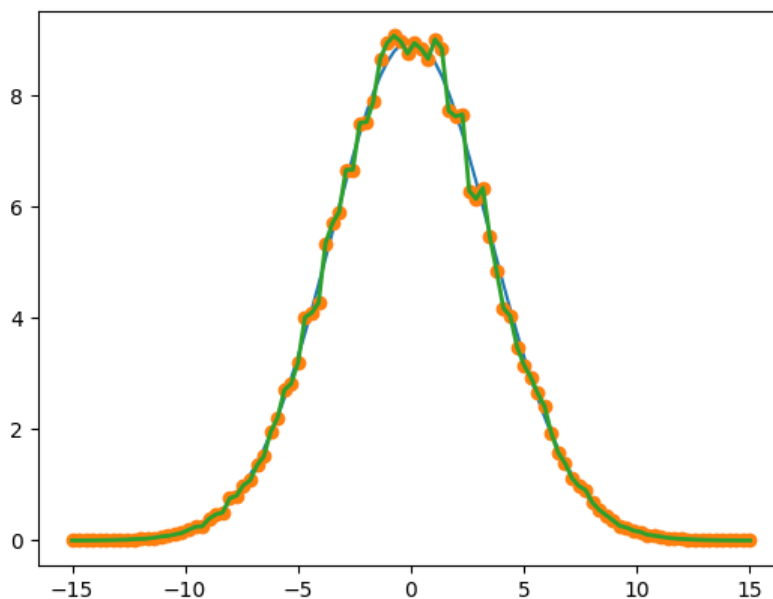
Note that, at present, only equally spaced real or complex data sets can be accommodated. This cubic spline class `S` can now be pasted to any of the `mesolve`, `mcsolve`, or `sesolve` functions where one would normally input a time-dependent function or string-representation. Taking the problem from the previous section as an example. We would make the replacement:

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

to

```
H = [H0, [H1, S]]
```

When combining interpolating functions with other Python functions or strings, the interpolating class will automatically pick the appropriate method for calling the class. That is to say that, if for example, you have other time-dependent terms that are given in the string-format, then the cubic spline representation will also be passed in a string-compatible format. In the string-format, the interpolation function is compiled into c-code, and thus is quite fast. This is the default method if no other time-dependent terms are present.

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.Options` object with the `rhs_reuse` flag turned on. Instructions on setting flags are found in *Setting Options for the Dynamics Solvers*. For example, we can do

```
In [61]: H = [H0, [H1, 'A * exp(-(t / sig) ** 2)']]

In [62]: args = {'A': 9, 'sig': 5}

In [63]: output = mcsolve(H, psi0, times, c_ops, [a.dag()*a], args=args)
10.0%. Run time:   0.12s. Est. time left: 00:00:00:01
20.0%. Run time:   0.20s. Est. time left: 00:00:00:00
```

```
30.0%. Run time:    0.28s. Est. time left: 00:00:00:00
40.0%. Run time:    0.36s. Est. time left: 00:00:00:00
50.0%. Run time:    0.44s. Est. time left: 00:00:00:00
60.0%. Run time:    0.52s. Est. time left: 00:00:00:00
70.0%. Run time:    0.60s. Est. time left: 00:00:00:00
80.0%. Run time:    0.68s. Est. time left: 00:00:00:00
90.0%. Run time:    0.76s. Est. time left: 00:00:00:00
100.0%. Run time:    0.84s. Est. time left: 00:00:00:00
Total run time:    0.93s

In [64]: opts = Options(rhs_reuse=True)

In [65]: args = {'A': 10, 'sig': 3}

In [66]: output = mcsolve(H, psi0, times, c_ops, [a.dag()*a], args=args,␣
→options=opts)
10.0%. Run time:    0.37s. Est. time left: 00:00:00:03
20.0%. Run time:    0.47s. Est. time left: 00:00:00:01
30.0%. Run time:    0.57s. Est. time left: 00:00:00:01
40.0%. Run time:    0.67s. Est. time left: 00:00:00:01
50.0%. Run time:    0.77s. Est. time left: 00:00:00:00
60.0%. Run time:    0.88s. Est. time left: 00:00:00:00
70.0%. Run time:    0.98s. Est. time left: 00:00:00:00
80.0%. Run time:    1.07s. Est. time left: 00:00:00:00
90.0%. Run time:    1.18s. Est. time left: 00:00:00:00
100.0%. Run time:    1.28s. Est. time left: 00:00:00:00
Total run time:    1.38s
```

The second call to *qutip.mcsolve* does not reorganize the data, and in the case of the string format, does not recompile the Cython code. For the small system here, the savings in computation time is quite small, however, 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.Options` 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 notebook "Time-dependent master equation: Landau-Zener-Stuckelberg inteferometry" in the tutorials section of the QuTiP web site.

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

```
In [67]: delta = 0.1  * 2 * np.pi  # qubit sigma_x coefficient

In [68]: w = 2.0  * 2 * np.pi      # driving frequency

In [69]: T = 2 * np.pi / w         # driving period

In [70]: gamma1 = 0.00001          # relaxation rate
```

```
In [71]: gamma2 = 0.005            # dephasing  rate

In [72]: eps_list = np.linspace(-10.0, 10.0, 51) * 2 * np.pi   # epsilon

In [73]: A_list = np.linspace(0.0, 20.0, 51) * 2 * np.pi        # Amplitude

In [74]: sx = sigmax(); sz = sigmaz(); sm = destroy(2); sn = num(2)

In [75]: c_ops = [np.sqrt(gamma1) * sm, np.sqrt(gamma2) * sz]   # relaxation and
→dephasing

In [76]: H0 = -delta / 2.0 * sx

In [77]: H1 = [sz, '-eps / 2.0 + A / 2.0 * sin(w * t)']

In [78]: H_td = [H0, H1]

In [79]: 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:

```
In [80]: opts = Options(rhs_reuse=True)

In [81]: rhs_generate(H_td, c_ops, 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 *qutip.parallel.parfor* with the m-index parallelized in loop over the elements of `p_mat[m,n]`:

```
In [82]: def task(args):
   ....:     m, eps = args
   ....:     p_mat_m = np.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_ops, Hargs, opts) #<- IMPORTANT LINE
   ....:         rho_ss = propagator_steadystate(U)
   ....:         p_mat_m[n] = expect(sn, rho_ss)
   ....:     return [m, p_mat_m]
   ....:
```

Notice the Options `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.

3.5.5 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. *[Coh92]* or *[Bre02]*. 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): $H = H_\text{S} + H_\text{B} + H_\text{I}$, where H is the total system+bath Hamiltonian, H_S and H_B are the system and bath Hamiltonians, respectively, and H_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

$$\frac{d}{dt}\rho_S(t) = -\hbar^{-2} \int_0^t d\tau \, \text{Tr}_B[H_I(t), [H_I(\tau), \rho_S(\tau) \otimes \rho_B]], \tag{3.9}$$

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 (3.9) 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. (3.9). The result is the Redfield equation

$$\frac{d}{dt}\rho_S(t) = -\hbar^{-2} \int_0^t d\tau \, \text{Tr}_B[H_I(t), [H_I(\tau), \rho_S(t) \otimes \rho_B]], \tag{3.10}$$

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:

$$\frac{d}{dt}\rho_S(t) = -\hbar^{-2} \int_0^\infty d\tau \, \text{Tr}_B[H_I(t), [H_I(t - \tau), \rho_S(t) \otimes \rho_B]]. \tag{3.11}$$

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 (3.11) 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:

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

$$g_{\alpha\beta}(-\tau) \left[\rho_S(t)A_\alpha(t-\tau)A_\beta(t) - A_\alpha(t)\rho_S(t)A_\beta(t-\tau)\right]\},$$

where $g_{\alpha\beta}(\tau) = \mathrm{Tr}_B\left[B_\alpha(t)B_\beta(t-\tau)\rho_B\right] = \langle B_\alpha(\tau)B_\beta(0)\rangle$, 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

$$\frac{d}{dt}\rho_{ab}(t) = -i\omega_{ab}\rho_{ab}(t) - \hbar^{-2}\sum_{\alpha,\beta}\sum_{c,d}^{\mathrm{sec}}\int_0^\infty d\tau \left\{ g_{\alpha\beta}(\tau)\left[\delta_{bd}\sum_n A_{an}^\alpha A_{nc}^\beta e^{i\omega_{cn}\tau} - A_{ac}^\alpha A_{db}^\beta e^{i\omega_{ca}\tau}\right] \right.$$

$$\left. + g_{\alpha\beta}(-\tau)\left[\delta_{ac}\sum_n A_{dn}^\alpha A_{nb}^\beta e^{i\omega_{nd}\tau} - A_{ac}^\alpha A_{db}^\beta e^{i\omega_{bd}\tau}\right]\right\}\rho_{cd}(t),$$

where the "sec" above the summation symbol indicate summation of the secular terms which satisfy $|\omega_{ab}-\omega_{cd}| \ll \tau_{\mathrm{decay}}$. 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 $S(\omega) = \int_{-\infty}^\infty d\tau e^{i\omega\tau}g(\tau)$:

$$\int_0^\infty d\tau \; g_{\alpha\beta}(\tau)e^{i\omega\tau} = \frac{1}{2}S_{\alpha\beta}(\omega) + i\lambda_{\alpha\beta}(\omega), \tag{3.12}$$

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

$$\frac{d}{dt}\rho_{ab}(t) = -i\omega_{ab}\rho_{ab}(t) + \sum_{c,d}^{\mathrm{sec}} R_{abcd}\rho_{cd}(t), \tag{3.13}$$

where

$$R_{abcd} = -\frac{\hbar^{-2}}{2}\sum_{\alpha,\beta}\left\{\delta_{bd}\sum_n A_{an}^\alpha A_{nc}^\beta S_{\alpha\beta}(\omega_{cn}) - A_{ac}^\alpha A_{db}^\beta S_{\alpha\beta}(\omega_{ca})\right.$$

$$\left. + \delta_{ac}\sum_n A_{dn}^\alpha A_{nb}^\beta S_{\alpha\beta}(\omega_{dn}) - A_{ac}^\alpha A_{db}^\beta S_{\alpha\beta}(\omega_{db})\right\},$$

is the Bloch-Redfield tensor.

The Bloch-Redfield master equation in the form Eq. (3.13) 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

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

$$\left. + \delta_{ac}\sum_n A_{dn}^\alpha A_{nb}^\alpha S_\alpha(\omega_{dn}) - A_{ac}^\alpha A_{db}^\alpha S_\alpha(\omega_{db})\right\}.$$

Bloch-Redfield master equation in QuTiP

In QuTiP, the Bloch-Redfield tensor Eq. (3.14) can be calculated using the function `qutip.bloch_redfield.bloch_redfield_tensor`. It takes two mandatory arguments: The system Hamiltonian H, a nested list of operator A_α, spectral density functions $S_\alpha(\omega)$ pairs that characterize the coupling between

system and bath. The spectral density functions are Python 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

$$H = -\frac{1}{2}\Delta\sigma_x - \frac{1}{2}\epsilon_0\sigma_z \qquad (3.14)$$

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

```
In [1]: delta = 0.2 * 2*np.pi; eps0 = 1.0 * 2*np.pi; gamma1 = 0.5

In [2]: H = - delta/2.0 * sigmax() - eps0/2.0 * sigmaz()

In [3]: def ohmic_spectrum(w):
   ...:     if w == 0.0: # dephasing inducing noise
   ...:         return gamma1
   ...:     else: # relaxation inducing noise
   ...:         return gamma1 / 2 * (w / (2 * np.pi)) * (w > 0.0)
   ...:

In [4]: R, ekets = bloch_redfield_tensor(H, [[sigmax(), ohmic_spectrum]])

In [5]: R
Out[5]:
Quantum object: dims = [[[2], [2]], [[2], [2]]], shape = (4, 4), type = super,␣
↪isherm = False
Qobj data =
[[ 0.00000000+0.j         0.00000000+0.j         0.00000000+0.j
   0.24514517+0.j       ]
 [ 0.00000000+0.j        -0.16103412-6.4076169j  0.00000000+0.j
   0.00000000+0.j       ]
 [ 0.00000000+0.j         0.00000000+0.j        -0.16103412+6.4076169j
   0.00000000+0.j       ]
 [ 0.00000000+0.j         0.00000000+0.j         0.00000000+0.j
  -0.24514517+0.j       ]]
```

Note that it is also possible to add Lindblad dissipation superoperators in the Bloch-Refield tensor by passing the operators via the c_ops keyword argument like you would in the *qutip.mesolve* or *qutip.mcsolve* functions. 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 (3.13), 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:

```
In [6]: import matplotlib.pyplot as plt

In [7]: tlist = np.linspace(0, 15.0, 1000)

In [8]: psi0 = rand_ket(2)

In [9]: e_ops = [sigmax(), sigmay(), sigmaz()]

In [10]: expt_list = bloch_redfield_solve(R, ekets, psi0, tlist, e_ops)

In [11]: sphere = Bloch()
```

```
In [12]: sphere.add_points([expt_list[0], expt_list[1], expt_list[2]])

In [13]: sphere.vector_color = ['r']

In [14]: sphere.add_vectors(np.array([delta, 0, eps0]) / np.sqrt(delta ** 2 + eps0
↪** 2))

In [15]: sphere.make_sphere()

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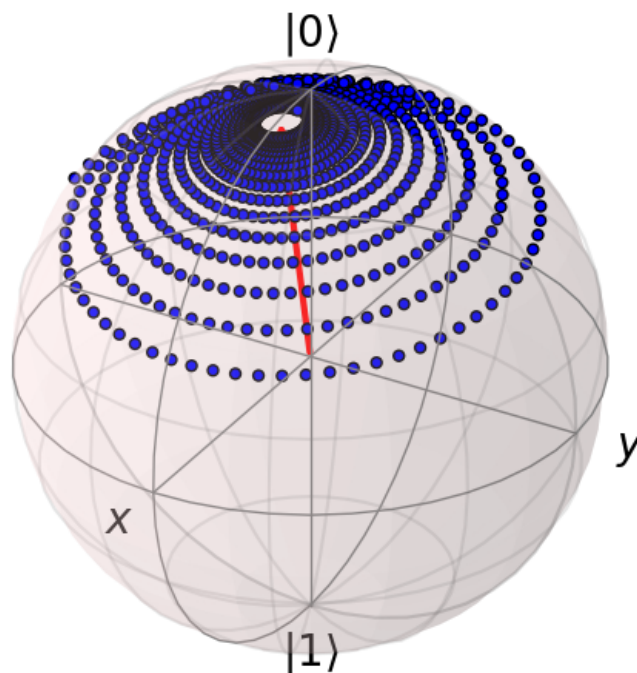

The two steps of calculating the Bloch-Redfield tensor and evolving according to the corresponding master equation can be combined into one by using the function *qutip.bloch_redfield.brmesolve*, which takes same arguments as *qutip.mesolve* and *qutip.mcsolve*, save for the additional nested list of operator-spectrum pairs that is called a_ops.

```
In [17]: output = brmesolve(H, psi0, tlist, a_ops=[[sigmax(),ohmic_spectrum]], e_
↪ops=e_ops)
```

where the resulting *output* is an instance of the class *qutip.solver.Result*.

Time-dependent Bloch-Redfield Dynamics

Note: New in QuTiP 4.2.

Warning: It takes ~3-5 seconds (~30 if using Visual Studio) to compile a time-dependent Bloch-Redfield problem. Therefore, if you are doing repeated simulations by varying parameters, then it is best to pass `options = Options(rhs_reuse=True)` to the solver.

If you have not done so already, please read the section: *Solving Problems with Time-dependent Hamiltonians*.

As we have already discussed, the Bloch-Redfield master equation requires transforming into the eigenbasis of the system Hamiltonian. For time-independent systems, this transformation need only be done once. However, for time-dependent systems, one must move to the instantaneous eigenbasis at each time-step in the evolution, thus greatly increasing the computational complexity of the dynamics. In addition, the requirement for computing all the eigenvalues severely limits the scalability of the method. Fortunately, this eigen decomposition occurs at the Hamiltonian level, as opposed to the super-operator level, and thus, with efficient programming, one can tackle many systems that are commonly encountered.

The time-dependent Bloch-Redfield solver in QuTiP relies on the efficient numerical computations afforded by the string-based time-dependent format, and Cython compilation. As such, all the time-dependent terms, and noise power spectra must be expressed in the string format. To begin, lets consider the previous example, but formatted to call the time-dependent solver:

```
In [18]: ohmic = "{gamma1} / 2.0 * (w / (2 * pi)) * (w > 0.0)".
→format(gamma1=gamma1)

In [19]: output = brmesolve(H, psi0, tlist, a_ops=[[sigmax(),ohmic]], e_ops=e_ops)
```

Although the problem itself is time-independent, the use of a string as the noise power spectrum tells the solver to go into time-dependent mode. The string is nearly identical to the Python function format, except that we replaced `np.pi` with `pi` to avoid calling Python in our Cython code, and we have hard coded the `gamma1` argument into the string as limitations prevent passing arguments into the time-dependent Bloch-Redfield solver.

For actual time-dependent Hamiltonians, the Hamiltonian itself can be passed into the solver like any other string-based Hamiltonian, as thus we will not discuss this topic further. Instead, here the focus is on time-dependent bath coupling terms. To this end, suppose that we have a dissipative harmonic oscillator, where the white-noise dissipation rate decreases exponentially with time $\kappa(t) = \kappa(0) \exp(-t)$. In the Lindblad or monte-carlo solvers, this could be implemented as a time-dependent collapse operator list `c_ops = [[a, 'sqrt(kappa*exp(-t))']]`. In the Bloch-Redfield solver, the bath coupling terms must be Hermitian. As such, in this example, our coupling operator is the position operator `a+a.dag()`. In addition, we do not need the `sqrt` operation that occurs in the `c_ops` definition. The complete example, and comparison to the analytic expression is:

```
In [20]: N = 10   # number of basis states to consider

In [21]: a = destroy(N)

In [22]: H = a.dag() * a

In [23]: psi0 = basis(N, 9)   # initial state

In [24]: kappa = 0.2   # coupling to oscillator

In [25]: a_ops = [[a+a.dag(), '{kappa}*exp(-t)*(w>=0)'.format(kappa=kappa)]]

In [26]: tlist = np.linspace(0, 10, 100)

In [27]: out = brmesolve(H, psi0, tlist, a_ops, e_ops=[a.dag() * a])

In [28]: actual_answer = 9.0 * np.exp(-kappa * (1.0 - np.exp(-tlist)))

In [29]: plt.figure()
Out[29]: <matplotlib.figure.Figure at 0x2b4e7dd74b38>

In [30]: plt.plot(tlist, out.expect[0])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[30]: [<matplotlib.lines.
→Line2D at 0x2b4e7eb36ac8>]

In [31]: plt.plot(tlist, actual_answer)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
→[<matplotlib.lines.Line2D at 0x2b4e7eb85470>]
```

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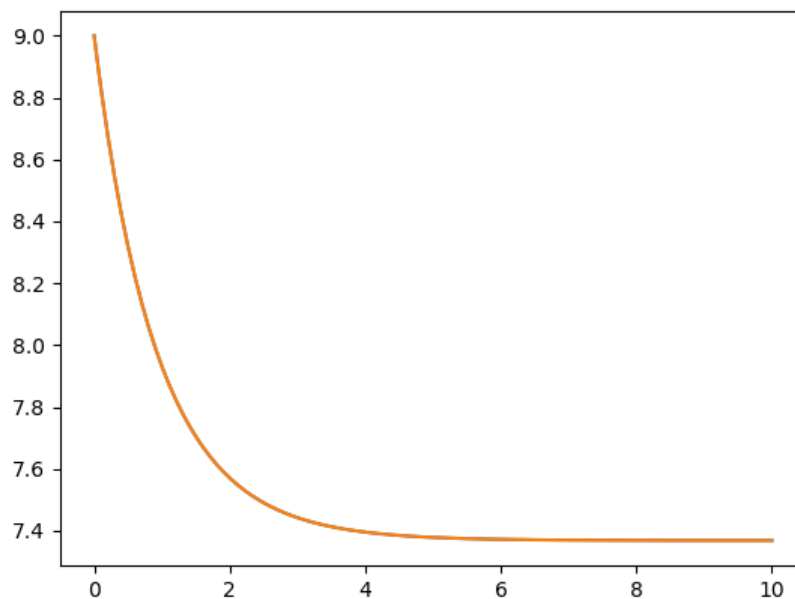

In many cases, the bath-coupling operators can take the form $A = f(t)a + f(t)^* a^+$. In this case, the above format for inputting the `a_ops` is not sufficient. Instead, one must construct a nested-list of tuples to specify this time-dependence. For example consider a white-noise bath that is coupled to an operator of the form `exp(1j*t)*a + exp(-1j*t)* a.dag()`. In this example, the `a_ops` list would be:

```
In [33]: a_ops = [ [ (a, a.dag()), ('{0} * (w >= 0)'.format(kappa), 'exp(1j*t)',
↪'exp(-1j*t)') ] ]
```

where the first tuple element `(a, a.dag())` tells the solver which operators make up the full Hermitian coupling operator. The second tuple `('{0} * (w >= 0)'.format(kappa), 'exp(1j*t)', 'exp(-1j*t)')`, gives the noise power spectrum, and time-dependence of each operator. Note that the noise spectrum must always come first in this second tuple. A full example is:

```
In [34]: N = 10

In [35]: w0 = 1.0 * 2 * np.pi

In [36]: g = 0.05 * w0

In [37]: kappa = 0.15

In [38]: times = np.linspace(0, 25, 1000)

In [39]: a = destroy(N)

In [40]: H = w0 * a.dag() * a + g * (a + a.dag())

In [41]: psi0 = ket2dm((basis(N, 4) + basis(N, 2) + basis(N, 0)).unit())

In [42]: a_ops = [[ (a, a.dag()), ('{0} * (w >= 0)'.format(kappa), 'exp(1j*t)',
↪'exp(-1j*t)') ]]

In [43]: e_ops = [a.dag() * a, a + a.dag()]

In [44]: res_brme = brmesolve(H, psi0, times, a_ops, e_ops)
```

```
In [45]: plt.figure()
Out[45]: <matplotlib.figure.Figure at 0x2b4e7ea755f8>

In [46]: plt.plot(times,res_brme.expect[0], label=r'$a^{+}a$')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[46]: [<matplotlib.lines.
↪Line2D at 0x2b4e7d58c5f8>]

In [47]: plt.plot(times,res_brme.expect[1], label=r'$a+a^{+}$')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪[<matplotlib.lines.Line2D at 0x2b4e7d054cc0>]

In [48]: plt.legend()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪<matplotlib.legend.Legend at 0x2b4e7e9b6cc0>

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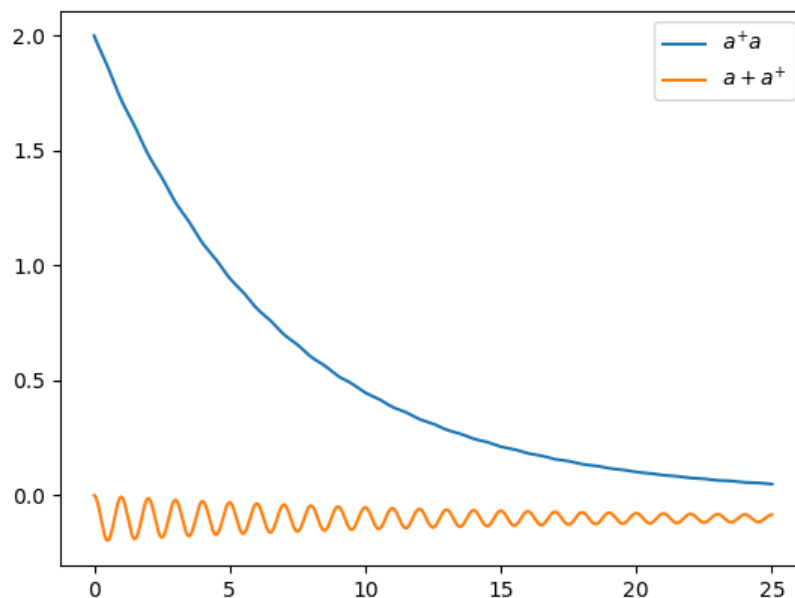

Further examples on time-dependent Bloch-Redfield simulations can be found in the online tutorials.

3.5.6 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., *[Gri98]*).

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

$$H(t)\Psi(t) = i\hbar \frac{\partial}{\partial t}\Psi(t), \tag{3.15}$$

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 (3.15) on the form

$$\Psi_\alpha(t) = \exp(-i\epsilon_\alpha t/\hbar)\Phi_\alpha(t), \tag{3.16}$$

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

$$\Psi(t) = \sum_\alpha c_\alpha \Psi_\alpha(t) = \sum_\alpha c_\alpha \exp(-i\epsilon_\alpha t/\hbar)\Phi_\alpha(t), \tag{3.17}$$

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 (3.15). By substituting (3.16) into the Schrödinger equation (3.15) we obtain an eigenvalue equation for the Floquet modes and quasienergies

$$\mathcal{H}(t)\Phi_\alpha(t) = \epsilon_\alpha \Phi_\alpha(t), \tag{3.18}$$

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 (3.15), which by definition satisfies

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

Inserting the Floquet states from (3.16) 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

$$H(t) = -\frac{1}{2}\Delta\sigma_x - \frac{1}{2}\epsilon_0\sigma_z + \frac{1}{2}A\sin(\omega t)\sigma_z. \tag{3.19}$$

In QuTiP we can define this Hamiltonian as follows:

```
In [1]: delta = 0.2 * 2*np.pi; eps0 = 1.0 * 2*np.pi; A = 2.5 * 2*np.pi; omega = 1.
↪0 * 2*np.pi

In [2]: H0 = - delta/2.0 * sigmax() - eps0/2.0 * sigmaz()

In [3]: H1 = A/2.0 * sigmaz()

In [4]: args = {'w': omega}

In [5]: H = [H0, [H1, 'sin(w * t)']]
```

The $t = 0$ Floquet modes corresponding to the Hamiltonian (3.19) can then be calculated using the `qutip.floquet.floquet_modes` function, which returns lists containing the Floquet modes and the quasienergies

```
In [6]: T = 2*pi / omega

In [7]: f_modes_0, f_energies = floquet_modes(H, T, args)

In [8]: f_energies
Out[8]: array([-2.83131211,  2.83131211])

In [9]: f_modes_0
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[9]:
[Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
 Qobj data =
 [[ 0.72964231+0.j        ]
  [-0.39993748+0.55468199j]],
 Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
 Qobj data =
 [[ 0.39993748+0.55468199j]
  [ 0.72964231+0.j        ]]]
```

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).

```
In [10]: delta = 0.2 * 2*np.pi; eps0  = 0.0 * 2*np.pi

In [11]: omega = 1.0 * 2*np.pi; A_vec = np.linspace(0, 10, 100) * omega;

In [12]: T = (2*pi)/omega

In [13]: tlist  = np.linspace(0.0, 10 * T, 101)

In [14]: psi0   = basis(2,0)

In [15]: q_energies = np.zeros((len(A_vec), 2))
```

```
In [16]: H0 = delta/2.0 * sigmaz() - eps0/2.0 * sigmax()

In [17]: args = omega

In [18]: 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
   ....:

In [19]: figure()
Out[19]: <matplotlib.figure.Figure at 0x2b4e7d799898>

In [20]: plot(A_vec/omega, q_energies[:,0] / delta, 'b', A_vec/omega, q_energies[:,
→1] / delta, 'r')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[20]:
[<matplotlib.lines.Line2D at 0x2b4e7ea87a58>,
 <matplotlib.lines.Line2D at 0x2b4e7ea87748>]

In [21]: xlabel(r'$A/\omega$')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
→<matplotlib.text.Text at 0x2b4e7d996390>

In [22]: ylabel(r'Quasienergy / $\Delta$')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
→<matplotlib.text.Text at 0x2b4e7d7f9320>

In [23]: title(r'Floquet quasienergies')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
→<matplotlib.text.Text at 0x2b4e7e8204e0>

In [24]: show()
```
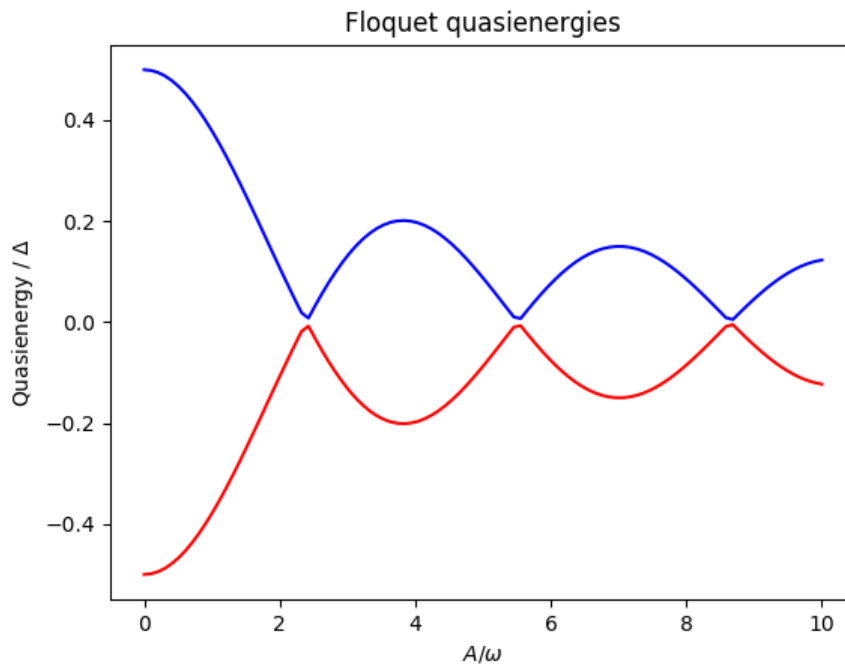

Given the Floquet modes at $t = 0$, we obtain the Floquet mode at some later time t using the function `qutip.floquet.floquet_mode_t`:

```
In [25]: f_modes_t = floquet_modes_t(f_modes_0, f_energies, 2.5, H, T, args)

In [26]: f_modes_t
Out[26]:
[Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
 Qobj data =
 [[-0.89630511-0.23191947j]
  [ 0.37793108-0.00431335j]],
 Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
 Qobj data =
 [[-0.37793108-0.00431335j]
  [-0.89630511+0.23191947j]]]
```

The purpose of calculating the Floquet modes is to find the wavefunction solution to the original problem (3.19) 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*

```
In [27]: psi0 = rand_ket(2)

In [28]:  f_coeff = floquet_state_decomposition(f_modes_0, f_energies, psi0)

In [29]: f_coeff
Out[29]:
[(-0.74351257922214109-0.34000983728423162j),
 (-0.48538254159679123-0.30981630589985082j)]
```

and given this decomposition of the initial state in the Floquet states we can easily evaluate the wavefunction that is the solution to (3.19) at an arbitrary time t using the function *qutip.floquet.floquet_wavefunction_t*

```
In [30]: t = 10 * np.random.rand()

In [31]: psi_t = floquet_wavefunction_t(f_modes_0, f_energies, f_coeff, t, H, T,␣
→args)

In [32]: psi_t
Out[32]:
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[-0.51967672+0.2539739j ]
 [ 0.25078818-0.77623363j]]
```

The following example illustrates how to use the functions introduced above to calculate and plot the time-evolution of (3.19).

```python
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)
```

3.5. Time Evolution and Quantum System Dynamics   **75**

```python
# 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()
```
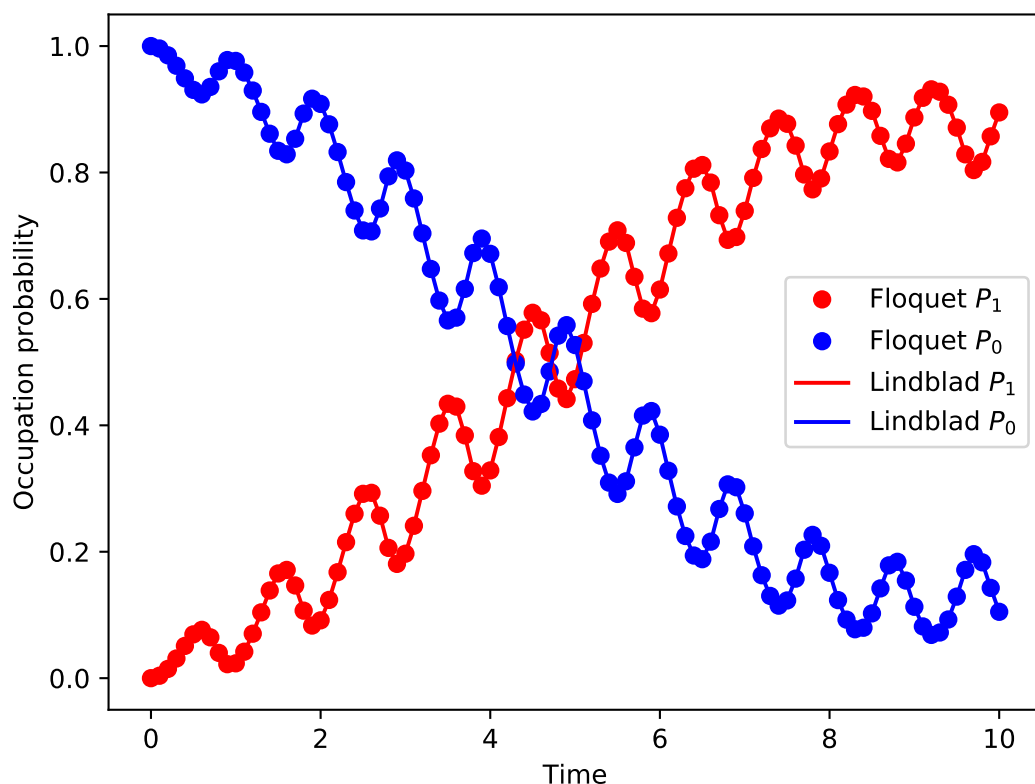

Pre-computing the Floquet modes for one period

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.

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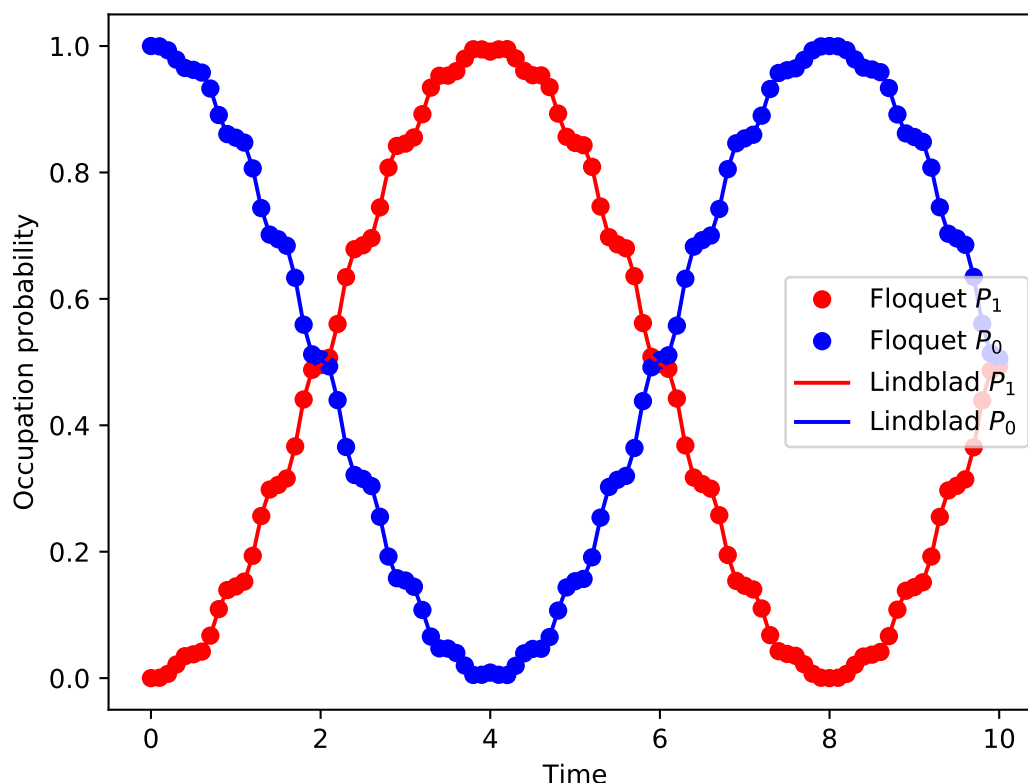
Note that the parameters and the Hamiltonian used in this example is not the same as in the previous section, and hence the different appearance of the resulting figure.

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, times, [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:

```python
>>> 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.solver.Result*. The following example extends the example studied above, and uses *qutip.floquet.fmmesolve* to introduce dissipation into the calculation

```python
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, True))

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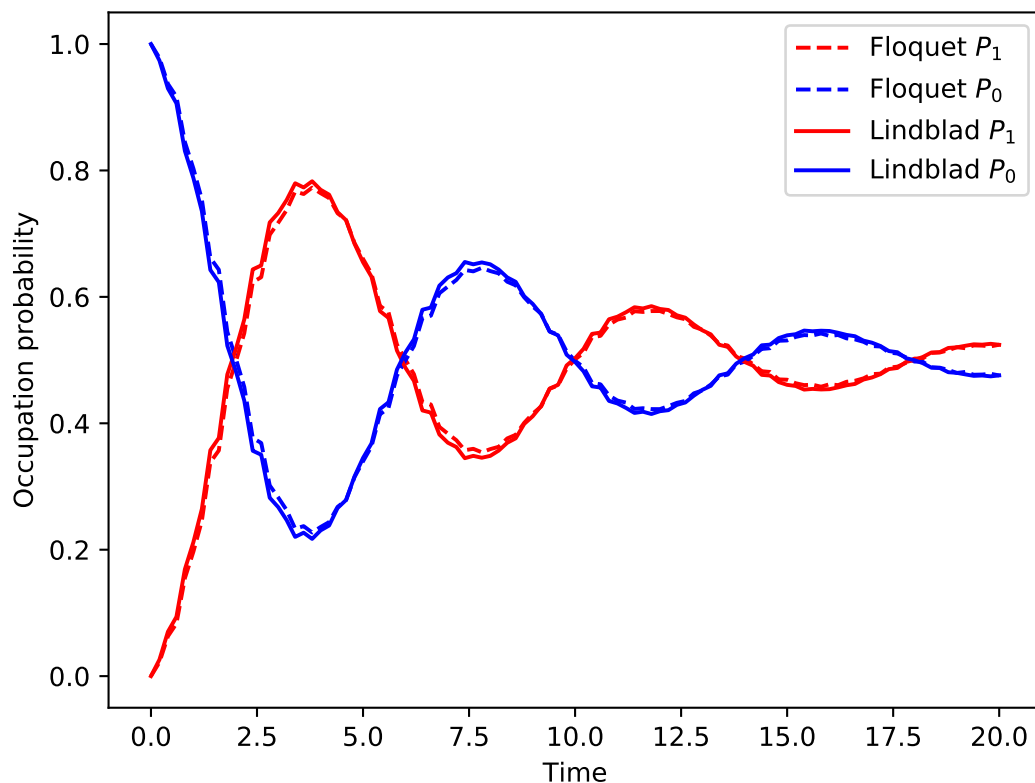
Alternatively, we can let the *qutip.floquet.fmmesolve* function transform the density matrix at each time step back to the computational basis, and calculating the expectation values for us, but using:

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

3.5.7 Setting Options for the Dynamics Solvers

Occasionally it is necessary to change the built in parameters of the dynamics solvers used by for example the *qutip.mesolve* and *qutip.mcsolve* functions. The options for all dynamics solvers may be changed by using the Options class *qutip.solver.Options*.

```
In [1]: options = Options()
```

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

```
In [2]: print(options)
Options:
-----------
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:       0
norm_tol:       0.001
```

```
norm_steps:       5
rhs_filename:     None
rhs_reuse:        False
seeds:            0
rhs_with_state:   False
average_expect:   True
average_states:   False
ntraj:            500
store_states:     False
store_final_state: False
```

These properties are detailed in the following table. Assuming `options = Options()`:

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

or one can use an inline method,

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

```
In [3]: print(options)
Options:
-----------
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:         0
norm_tol:         0.001
norm_steps:       5
rhs_filename:     None
rhs_reuse:        False
seeds:            0
```

```
rhs_with_state:    False
average_expect:    True
average_states:    False
ntraj:             500
store_states:      False
store_final_state: False
```

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

or:

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

3.6 Solving for Steady-State Solutions

Important: Updated in QuTiP 3.2.

3.6.1 Introduction

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

$$\frac{d\hat{\rho}_{ss}}{dt} = \mathcal{L}\hat{\rho}_{ss} = 0.$$

Although the requirement for time-independence seems quite resitrictive, one can often employ a transformation to the interaction picture that yields a time-independent Hamiltonian. For many these systems, solving for the asymptotic density matrix $\hat{\rho}_{ss}$ can be achieved using direct or iterative solution methods faster than using master equation or Monte Carlo simulations. Although the steady state equation has a simple mathematical form, the properties of the Liouvillian operator are such that the solutions to this equation are anything but straightforward to find.

3.6.2 Steady State solvers in QuTiP

In QuTiP, the steady-state solution for a system Hamiltonian or Liouvillian is given by `qutip.steadystate.steadystate`. This function implements a number of different methods for finding the steady state, each with their own pros and cons, where the method used can be chosen using the `method` keyword argument.

Method	Keyword	Description
Direct (default)	'direct'	Direct solution solving $Ax = b$ via sparse LU decomposition.
Eigenvalue	'eigen'	Iteratively find the zero eigenvalue of $\mathcal{L}$.
Inverse-Power	'power'	Solve using the inverse-power method.
GMRES	'iterative-gmres'	Solve using the GMRES method and optional preconditioner.
LGMRES	'iterative-lgmres'	Solve using the LGMRES method and optional preconditioner.
BICGSTAB	'iterative-bicgstab'	Solve using the BICGSTAB method and optional preconditioner.
SVD	'svd'	Steady-state solution via the **dense** SVD of the Liouvillian.

The function `qutip.steadystate.steadystate` can take either a Hamiltonian and a list of collapse operators as input, generating internally the corresponding Liouvillian super operator in Lindblad form, or alternatively, a Liouvillian passed by the user. When possible, we recommend passing the Hamiltonian and collapse operators to `qutip.steadystate.steadystate`, and letting the function automatically build the Liouvillian (in Lindblad form) for the system.

As of QuTiP 3.2, the `direct` and `power` methods can take advantage of the Intel Pardiso LU solver in the Intel Math Kernel library that comes with the Anacoda (2.5+) and Intel Python distributions. This gives a substantial increase in performance compared with the standard SuperLU method used by SciPy. To verify that QuTiP can find the necessary libraries, one can check for `INTEL MKL Ext:   True` in the QuTiP about box (`qutip.about`).

3.6.3 Using the Steadystate Solver

Solving for the steady state solution to the Lindblad master equation for a general system with `qutip.steadystate.steadystate` can be accomplished using:

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

where `H` is a quantum object representing the system Hamiltonian, and `c_ops` is a list of quantum objects for the system collapse operators. The output, labeled as `rho_ss`, is the steady-state solution for the systems. If no other keywords are passed to the solver, the default 'direct' method is used, generating a solution that is exact to machine precision at the expense of a large memory requirement. The large amount of memory need for the direct LU decomposition method stems from the large bandwidth of the system Liouvillian and the correspondingly large fill-in (extra nonzero elements) generated in the LU factors. This fill-in can be reduced by using bandwidth minimization algorithms such as those discussed in *Additional Solver Arguments*. However, in most cases, the default fill-in reducing algorithm is nearly optimal. Additional parameters may be used by calling the steady-state solver as:

```
rho_ss = steadystate(H, c_ops, method='power', use_rcm=True)
```

where `method='power'` indicates that we are using the inverse-power solution method, and `use_rcm=True` turns on a bandwidth minimization routine.

Although it is not obvious, the `'direct'`, `eigen`, and `'power'` methods all use an LU decomposition internally and thus suffer from a large memory overhead. In contrast, iterative methods such as the `'iterative-gmres'`, `'iterative-lgmres'`, and `'iterative-bicgstab'` methods do not factor the matrix and thus take less memory than these previous methods and allowing, in principle, for extremely large system sizes. The downside is that these methods can take much longer than the direct method as the condition number of the Liouvillian matrix is large, indicating that these iterative methods require a large number of iterations for convergence. To overcome this, one can use a preconditioner M that solves for an approximate inverse for the (modified) Liouvillian, thus better conditioning the problem, leading to faster convergence. The use of a preconditioner can actually make these iterative methods faster than the other solution methods. The problem with preconditioning is that it is only well defined for Hermitian matrices. Since the Liouvillian is non-Hermitian, the ability to find a good preconditioner is not guaranteed. And moreover, if a preconditioner is found, it is not guaranteed to have a good condition number. QuTiP can make use of an incomplete LU preconditioner when using the iterative `'gmres'`, `'lgmres'`, and `'bicgstab'` solvers by setting `use_precond=True`. The preconditioner optionally makes use of a combination of symmetric and anti-symmetric matrix permutations that attempt to improve the preconditioning process. These features are discussed in the *Additional Solver Arguments* section. Even with these state-of-the-art permutations, the generation of a successful preconditoner for non-symmetric matrices is currently a trial-and-error process due to the lack of mathematical work done in this area. It is always recommended to begin with the direct solver with no additional arguments before selecting a different method.

Finding the steady-state solution is not limited to the Lindblad form of the master equation. Any time-independent Liouvillian constructed from a Hamiltonian and collapse operators can be used as an input:

```
>>> rho_ss = steadystate(L)
```

where `L` is the Louvillian. All of the additional arguments can also be used in this case.

3.6.4 Additional Solver Arguments

The following additional solver arguments are available for the steady-state solver:

Key-word	Options (default listed first)	Description
method	'direct', 'eigen', 'power', 'iterative-gmres','iterative-lgmres', 'svd'	Method used for solving for the steady-state density matrix.
sparse	True, False	Use sparse version of direct solver.
weight	None	Allows the user to define the weighting factor used in the `'direct'`, `'GMRES'`, and `'LGMRES'` solvers.
permc_spec	'COLAMD', 'NATURAL'	Column ordering used in the sparse LU decomposition.
use_rcm	False, True	Use a Reverse Cuthill-Mckee reordering to minimize the bandwidth of the modified Liouvillian used in the LU decomposition. If `use_rcm=True` then the column ordering is set to `'Natural'` automatically unless explicitly set.
use_precond	False, True	Attempt to generate a preconditioner when using the `'iterative-gmres'` and `'iterative-lgmres'` methods.
M	None, sparse_matrix, LinearOperator	A user defined preconditioner, if any.
use_wbm	False, True	Use a Weighted Bipartite Matching algorithm to attempt to make the modified Liouvillian more diagonally dominate, and thus for favorable for preconditioning. Set to `True` automatically when using a iterative method, unless explicitly set.
tol	1e-9	Tolerance used in finding the solution for all methods expect `'direct'` and `'svd'`.
max-iter	10000	Maximum number of iterations to perform for all methods expect `'direct'` and `'svd'`.
fill_factor	10	Upper-bound on the allowed fill-in for the approximate inverse preconditioner. This value may need to be set much higher than this in some cases.
drop_tol	1e-3	Sets the threshold for the relative magnitude of preconditioner elements that should be dropped. A lower number yields a more accurate approximate inverse at the expense of fill-in and increased runtime.
diag_pivot_thresh	None	Sets the threshold between $[0, 1]$ for which diagonal elements are considered acceptable pivot points when using a preconditioner.
ILU_MILU	'smilu_2'	Selects the incomplete LU decomposition method algorithm used.

Further information can be found in the *qutip.steadystate.steadystate* docstrings.

3.6.5 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 the $|10\rangle$ 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.

```python
import numpy as np
import pylab as plt
from qutip import *
# Define paramters
N = 20  # number of basis states to consider
a = destroy(N)
```

```python
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(np.sqrt(rate) * a)   # decay operators
rate = kappa * n_th_a
if rate > 0.0:
    c_op_list.append(np.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 = np.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])

plt.plot(tlist, mcdata.expect[0], tlist, medata.expect[0], lw=2)
# plot steady-state expt. value as horizontal line (should be = 2)
plt.axhline(y=fexpt, color='r', lw=1.5)
plt.ylim([0, 10])
plt.xlabel('Time', fontsize=14)
plt.ylabel('Number of excitations', fontsize=14)
plt.legend(('Monte-Carlo', 'Master Equation', 'Steady State'))
plt.title('Decay of Fock state $\left|10\\rangle\\right.$' +
          ' in a thermal environment with $\langle n\\rangle=2$')
plt.show()
```

Decay of Fock state $|10\rangle$ in a thermal environment with $\langle n \rangle = 2$
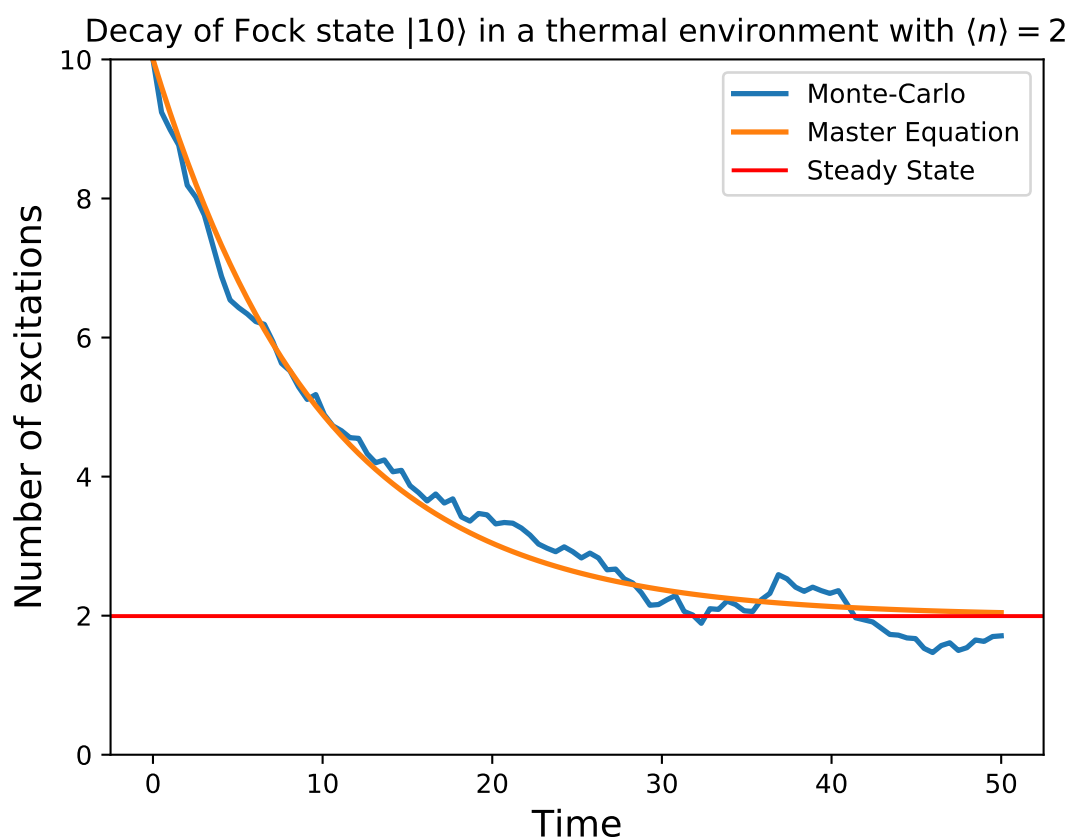
3.7 An Overview of the Eseries Class

3.7.1 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 [2]: omega=1.0

In [3]: 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 [4]: 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 [5]: es2
Out[5]:
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 [6]: esval(es2, 0.0)      # equivalent to es2.value(0.0)
Out[6]:
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 [7]: times = [0.0, 1.0 * pi, 2.0 * pi]

In [8]: esval(es2, times)    # equivalent to es2.value(times)
Out[8]:
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 [9]: es3 = (eseries([0.5*sigmaz(), 0.5*sigmaz()], [1j, -1j]) +
   ...:        eseries([-0.5j*sigmax(), 0.5j*sigmax()], [1j, -1j]))
   ...:
```

```
In [10]: rho = fock_dm(2, 1)

In [11]: es3_expect = expect(rho, es3)

In [12]: es3_expect
Out[12]:
ESERIES object: 2 terms
Hilbert space dimensions: [[1, 1]]
Exponent #0 = (-0-1j)
(-0.5+0j)
Exponent #1 = 1j
(-0.5+0j)

In [13]: es3_expect.value([0.0, pi/2])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
→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 *times* we use esval(es3_expect, times), or, equivalently, es3_expect.value(times).

3.7.2 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 [14]: psi0 = basis(2,1)

In [15]: H = sigmaz()

In [16]: L = liouvillian(H, [sqrt(1.0) * destroy(2)])

In [17]: 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 [18]: es
Out[18]:
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 [19]: es.value([0.0, 1.0])
Out[19]:
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 [20]: 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 [21]: es_expect.value([0.0, 1.0, 2.0, 3.0])
Out[21]: array([-1.       ,  0.26424112,  0.72932943,  0.90042586])
```

```
In [22]: times = linspace(0.0, 10.0, 100)

In [23]: sz_expect = es_expect.value(times)

In [24]: from pylab import *

In [25]: plot(times, sz_expect, lw=2);

In [26]: xlabel("Time", fontsize=16)
   ....: ylabel("Expectation value of sigma-z", fontsize=16);
   ....:

In [28]: title("The expectation value of the $\sigma_{z}$ operator", fontsize=16);
```

The expectation value of the σ_z operator
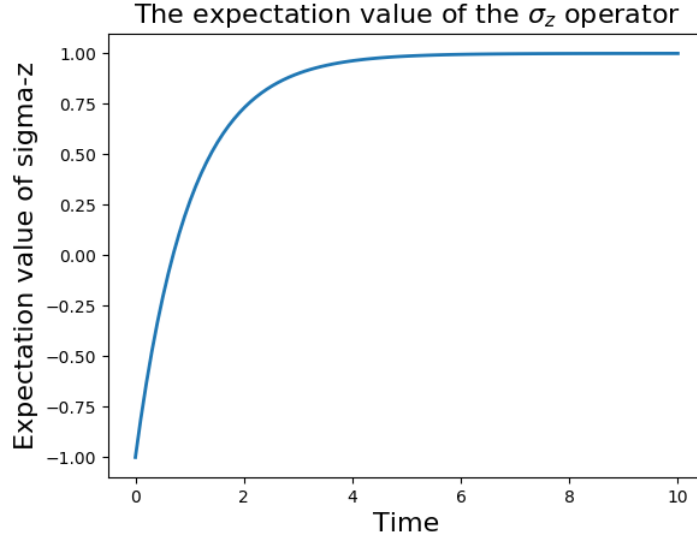
3.8 Two-time correlation functions

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, $\rho(t) = V(t, t_0)\{\rho(t_0)\}$, 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., *[Gar03]*) to write

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

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

Note that if the initial state is the steady state, then $\rho(t) = V(t, 0)\{\rho_\text{ss}\} = \rho_\text{ss}$ and

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

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 shown 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` or `qutip.correlation.correlation_2op_2t`	$\langle A(t+\tau)B(t)\rangle$ or $\langle A(t)B(t+\tau)\rangle$.
`qutip.correlation.correlation_ss` 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 solvers return a vector or matrix (in general complex) with the correlations as a function of the delays times.

3.8.1 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.

```
In [1]: times = np.linspace(0,10.0,200)

In [2]: a = destroy(10)

In [3]: x = a.dag() + a

In [4]: H = a.dag() * a

In [5]: corr1 = correlation_ss(H, times, [np.sqrt(0.5) * a], x, x)

In [6]: corr2 = correlation_ss(H, times, [np.sqrt(1.0) * a], x, x)

In [7]: corr3 = correlation_ss(H, times, [np.sqrt(2.0) * a], x, x)

In [8]: figure()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪<matplotlib.figure.Figure at 0x2b4e7d523710>

In [9]: plot(times, np.real(corr1), times, np.real(corr2), times, np.real(corr3))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
[<matplotlib.lines.Line2D at 0x2b4e7dbbf4e0>,
 <matplotlib.lines.Line2D at 0x2b4e7dbbf7b8>,
 <matplotlib.lines.Line2D at 0x2b4e7e717320>]

In [10]: legend(['0.5','1.0','2.0'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪<matplotlib.legend.Legend at 0x2b4e80cc99e8>

In [11]: xlabel(r'Time $t$')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪<matplotlib.text.Text at 0x2b4e7e6bcd30>

In [12]: ylabel(r'Correlation $\left<x(t)x(0)\right>$')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪<matplotlib.text.Text at 0x2b4e7dd219b0>

In [13]: show()
```
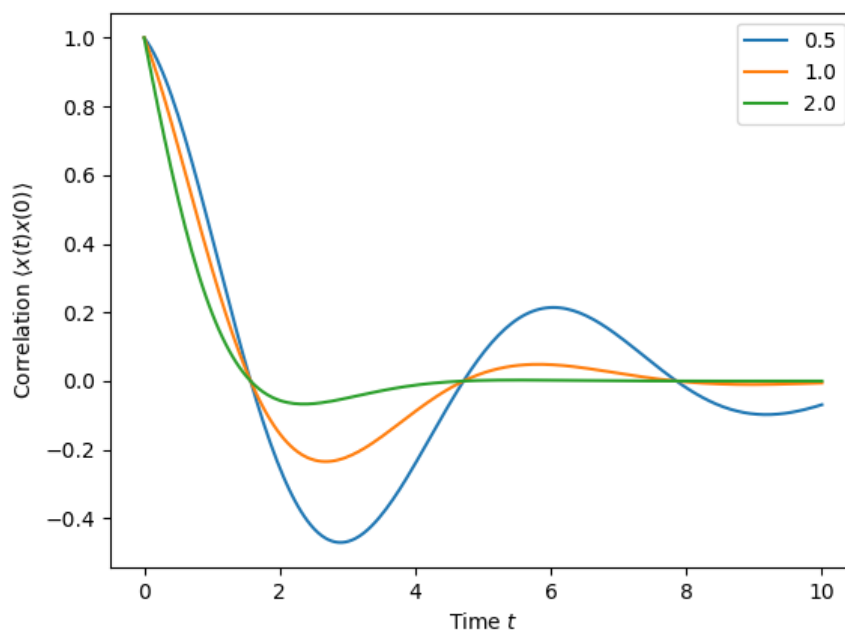
3.8.2 Emission spectrum

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.

```python
import numpy as np
import pylab as plt
from qutip import *

N = 4                          # number of cavity fock states
wc = wa = 1.0 * 2 * np.pi      # cavity and atom frequency
g  = 0.1 * 2 * np.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 = [np.sqrt(kappa * (1 + n_th)) * a, np.sqrt(kappa * n_th) * a.dag(), np.
→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 = np.linspace(0, 100, 5000)
corr = correlation_2op_1t(H, None, tlist, c_ops, a.dag(), a)
wlist1, spec1 = spectrum_correlation_fft(tlist, corr)


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

# plot the spectra
fig, ax = plt.subplots(1, 1)
ax.plot(wlist1 / (2 * np.pi), spec1, 'b', lw=2, label='eseries method')
ax.plot(wlist2 / (2 * np.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*np.pi), wlist2[-1]/(2*np.pi))
plt.show()
```
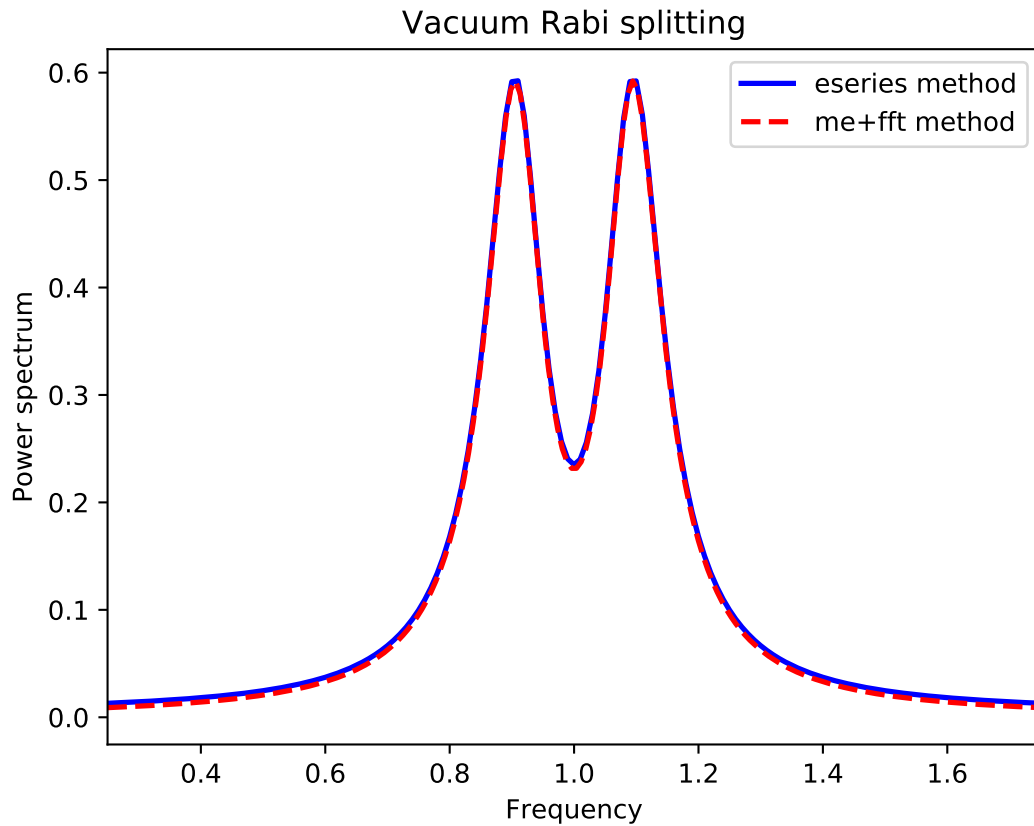

3.8.3 Non-steadystate correlation function

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_2op_2t*. 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).

```python
import numpy as np
import pylab as plt
from qutip import *

times = np.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_2op_2t(H, rho0, times, times, [np.sqrt(0.25) * a], x, x)

plt.pcolor(np.real(corr))
plt.xlabel(r'Time $t_2$')
plt.ylabel(r'Time $t_1$')
plt.title(r'Correlation $\left<x(t)x(0)\right>$')
plt.show()
```
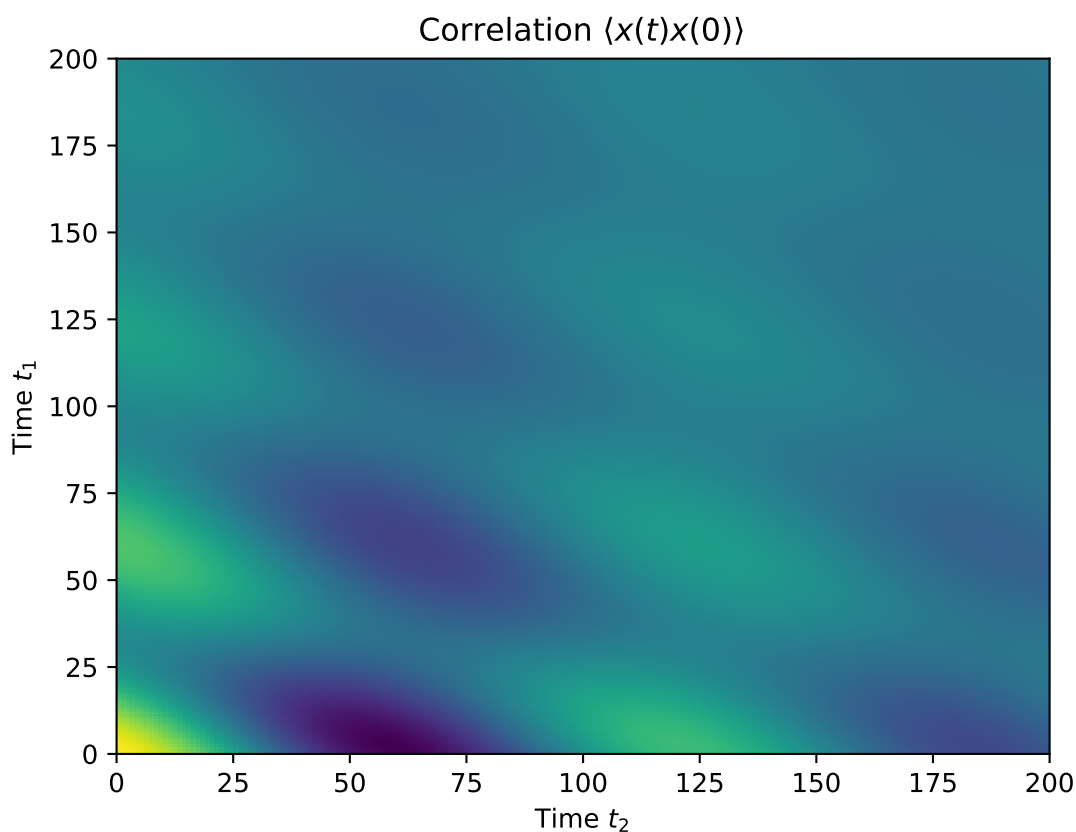

However, in some cases we might be interested in the correlation functions on the form $\langle A(t_1 + t_2)B(t_1) \rangle$, but only as a function of time coordinate t_2. In this case we can also use the *qutip.correlation.correlation_2op_2t* function, if we pass the density matrix at time t_1 as second argument, and *None* as third argument. The *qutip.correlation.correlation_2op_2t* 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.

```python
import numpy as np
import pylab as plt
from qutip import *

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

# collapse operator
G1 = 0.75
n_th = 2.00   # bath temperature in terms of excitation number
c_ops = [np.sqrt(G1 * (1 + n_th)) * a, np.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, taus, c_ops, [a.dag() * a]).expect[0]

# calculate the correlation function G1 and normalize with n to obtain g1
G1 = correlation_2op_2t(H, rho0, None, taus, c_ops, a.dag(), a)
g1 = G1 / np.sqrt(n[0] * n)

plt.plot(taus, np.real(g1), 'b', lw=2)
plt.plot(taus, n, 'r', lw=2)
plt.title('Decay of a coherent state to an incoherent (thermal) state')
plt.xlabel(r'$\tau$')
plt.legend((r'First-order coherence function $g^{(1)}(\tau)$',
        r'occupation number $n(\tau)$'))
plt.show()
```
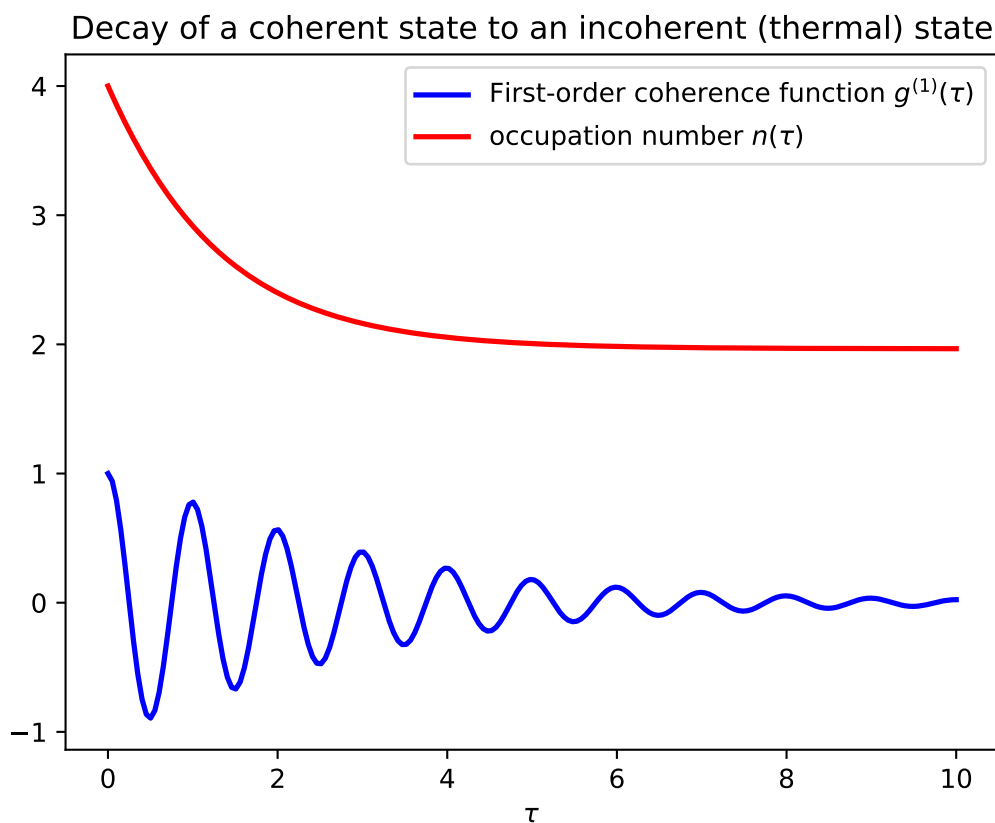
For convenience, the steps for calculating the first-order coherence function have been collected in the function `qutip.correlation.coherence_function_g1`.

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.

```python
import numpy as np
import pylab as plt
from qutip import *

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

kappa = 0.25
n_th = 2.0  # bath temperature in terms of excitation number
```

```
c_ops = [np.sqrt(kappa * (1 + n_th)) * a, np.sqrt(kappa * n_th) * a.dag()]

states = [{'state': coherent_dm(N, np.sqrt(2)), 'label': "coherent state"},
          {'state': thermal_dm(N, 2), '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, taus, c_ops, [a.dag() * a]).expect[0]

    # calculate the correlation function G2 and normalize with n(0)n(t) to
    # obtain g2
    G2 = correlation_3op_1t(H, rho0, taus, c_ops, a.dag(), a.dag()*a, a)
    g2 = G2 / (n[0] * n)

    ax.plot(taus, np.real(g2), label=state['label'], lw=2)

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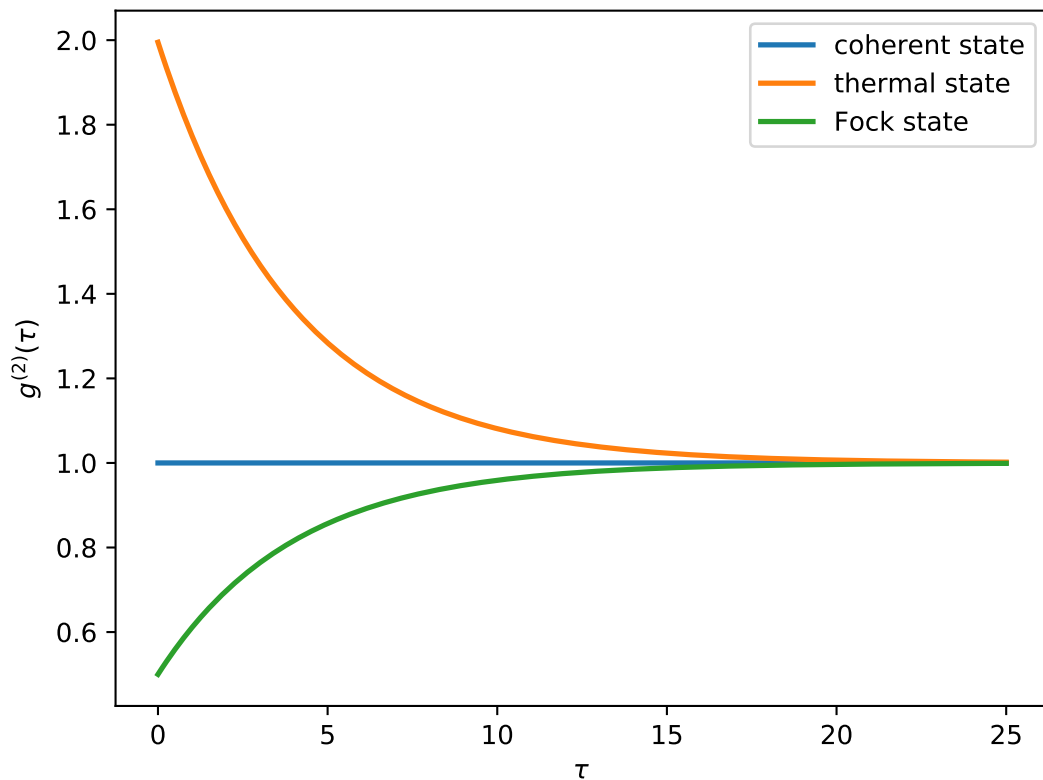

For convenience, the steps for calculating the second-order coherence function have been collected in the function *qutip.correlation.coherence_function_g2*.

3.9 Quantum Optimal Control

3.9.1 Introduction

In quantum control we look to prepare some specific state, effect some state-to-state transfer, or effect some transformation (or gate) on a quantum system. For a given quantum system there will always be factors that effect the dynamics that are outside of our control. As examples, the interactions between elements of the system or a magnetic field required to trap the system. However, there may be methods of affecting the dynamics in a controlled way, such as the time varying amplitude of the electric component of an interacting laser field. And so this leads to some questions; given a specific quantum system with known time-independent dynamics generator (referred to as the *drift* dynamics generators) and set of externally controllable fields for which the interaction can be described by *control* dynamics generators:

1. what states or transformations can we achieve (if any)?

2. what is the shape of the control pulse required to achieve this?

These questions are addressed as *controllability* and *quantum optimal control [dAless08]*. The answer to question of *controllability* is determined by the commutability of the dynamics generators and is formalised as the *Lie Algebra Rank Criterion* and is discussed in detail in *[dAless08]*. The solutions to the second question can be determined through optimal control algorithms, or control pulse optimisation.
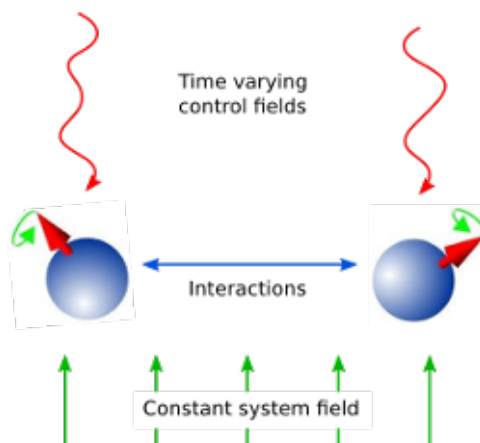
Fig. 3.3: Schematic showing the principle of quantum control.

Quantum Control has many applications including NMR, *quantum metrology*, *control of chemical reactions*, and *quantum information processing*.

To explain the physics behind these algorithms we will first consider only finite-dimensional, closed quantum systems.

3.9.2 Closed Quantum Systems

In closed quantum systems the states can be represented by kets, and the transformations on these states are unitary operators. The dynamics generators are Hamiltonians. The combined Hamiltonian for the system is given by

$$H(t) = H_0 + \sum_{j=1} u_j(t) H_j$$

where H_0 is the drift Hamiltonian and the H_j are the control Hamiltonians. The u_j are time varying amplitude functions for the specific control.

The dynamics of the system are governed by *Schrödingers equation*.

$$\frac{d}{dt} |\psi\rangle = -iH(t) |\psi\rangle$$

Note we use units where $\hbar = 1$ throughout. The solutions to Schrödinger's equation are of the form:

$$|\psi(t)\rangle = U(t)|\psi_0\rangle$$

where ψ_0 is the state of the system at $t = 0$ and $U(t)$ is a unitary operator on the Hilbert space containing the states. $U(t)$ is a solution to the *Schrödinger operator equation*

$$\tfrac{d}{dt}U = -iH(t)U, \quad U(0) = \mathbb{1}$$

We can use optimal control algorithms to determine a set of u_j that will drive our system from $|\psi_0\rangle$ to $|\psi_1\rangle$, this is state-to-state transfer, or drive the system from some arbitary state to a given state $|\psi_1\rangle$, which is state preparation, or effect some unitary transformation U_{target}, called gate synthesis. The latter of these is most important in quantum computation.

3.9.3 The GRAPE algorithm

The **GR**adient **A**scent **P**ulse **E**ngineering was first proposed in [2]. Solutions to Schrödinger's equation for a time-dependent Hamiltonian are not generally possible to obtain analytically. Therefore, a piecewise constant approximation to the pulse amplitudes is made. Time allowed for the system to evolve T is split into M timeslots (typically these are of equal duration), during which the control amplitude is assumed to remain constant. The combined Hamiltonian can then be approximated as:

$$H(t) \approx H(t_k) = H_0 + \sum_{j=1}^{N} u_{jk} H_j$$

where k is a timeslot index, j is the control index, and N is the number of controls. Hence t_k is the evolution time at the start of the timeslot, and u_{jk} is the amplitude of control j throughout timeslot k. The time evolution operator, or propagator, within the timeslot can then be calculated as:

$$X_k := e^{-iH(t_k)\Delta t_k}$$

where Δt_k is the duration of the timeslot. The evolution up to (and including) any timeslot k (including the full evolution $k = M$) can the be calculated as

$$X(t_k) := X_k X_{k-1} \cdots X_1 X_0$$

If the objective is state-to-state transfer then $X_0 = |\psi_0\rangle$ and the target $X_{targ} = |\psi_1\rangle$, for gate synthesis $X_0 = U(0) = \mathbb{1}$ and the target $X_{targ} = U_{targ}$.

A *figure of merit* or *fidelity* is some measure of how close the evolution is to the target, based on the control amplitudes in the timeslots. The typical figure of merit for unitary systems is the normalised overlap of the evolution and the target.

$$f_{PSU} = \tfrac{1}{d}\left| \operatorname{tr}\{X_{targ}^{\dagger} X(T)\} \right|$$

where d is the system dimension. In this figure of merit the absolute value is taken to ignore any differences in global phase, and $0 \leq f \leq 1$. Typically the fidelity error (or *infidelity*) is more useful, in this case defined as $\varepsilon = 1 - f_{PSU}$. There are many other possible objectives, and hence figures of merit.

As there are now $N \times M$ variables (the u_{jk}) and one parameter to minimise ε, then the problem becomes a finite multi-variable optimisation problem, for which there are many established methods, often referred to as 'hill-climbing' methods. The simplest of these to understand is that of steepest ascent (or descent). The gradient of the fidelity with respect to all the variables is calculated (or approximated) and a step is made in the variable space in the direction of steepest ascent (or descent). This method is a first order gradient method. In two dimensions this describes a method of climbing a hill by heading in the direction where the ground rises fastest. This analogy also clearly illustrates one of the main challenges in multi-variable optimisation, which is that all methods have a tendency to get stuck in local maxima. It is hard to determine whether one has found a global maximum or not - a local peak is likely not to be the highest mountain in the region. In quantum optimal control we can typically define an infidelity that has a lower bound of zero. We can then look to minimise the infidelity (from here on we will

only consider optimising for infidelity minima). This means that we can terminate any pulse optimisation when the infidelity reaches zero (to a sufficient precision). This is however only possible for fully controllable systems; otherwise it is hard (if not impossible) to know that the minimum possible infidelity has been achieved. In the hill walking analogy the step size is roughly fixed to a stride, however, in computations the step size must be chosen. Clearly there is a trade-off here between the number of steps (or iterations) required to reach the minima and the possibility that we might step over a minima. In practice it is difficult to determine an efficient and effective step size.

The second order differentials of the infidelity with respect to the variables can be used to approximate the local landscape to a parabola. This way a step (or jump) can be made to where the minima would be if it were parabolic. This typically vastly reduces the number of iterations, and removes the need to guess a step size. The method where all the second differentials are calculated explicitly is called the *Newton-Raphson* method. However, calculating the second-order differentials (the Hessian matrix) can be computationally expensive, and so there are a class of methods known as *quasi-Newton* that approximate the Hessian based on successive iterations. The most popular of these (in quantum optimal control) is the Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS). The default method in the QuTiP Qtrl GRAPE implementation is the L-BFGS-B method in Scipy, which is a wrapper to the implementation described in *[Byrd95]*. This limited memory and bounded method does not need to store the entire Hessian, which reduces the computer memory required, and allows bounds to be set for variable values, which considering these are field amplitudes is often physical.

The pulse optimisation is typically far more efficient if the gradients can be calculated exactly, rather than approximated. For simple fidelity measures such as f_{PSU} this is possible. Firstly the propagator gradient for each timeslot with respect to the control amplitudes is calculated. For closed systems, with unitary dynamics, a method using the eigendecomposition is used, which is efficient as it is also used in the propagator calculation (to exponentiate the combined Hamiltonian). More generally (for example open systems and symplectic dynamics) the Frechet derivative (or augmented matrix) method is used, which is described in *[Flo12]*. For other optimisation goals it may not be possible to calculate analytic gradients. In these cases it is necessary to approximate the gradients, but this can be very expensive, and can lead to other algorithms out-performing GRAPE.

3.9.4 The CRAB Algorithm

It has been shown *[Lloyd14]*, the dimension of a quantum optimal control problem is a polynomial function of the dimension of the manifold of the time-polynomial reachable states, when allowing for a finite control precision and evolution time. You can think of this as the information content of the pulse (as being the only effective input) being very limited e.g. the pulse is compressible to a few bytes without loosing the target.

This is where the **C**hopped **RA**ndom **B**asis (CRAB) algorithm *[Doria11]*, *[Caneva11]* comes into play: Since the pulse complexity is usually very low, it is sufficient to transform the optimal control problem to a few parameter search by introducing a physically motivated function basis that builds up the pulse. Compared to the number of time slices needed to accurately simulate quantum dynamics (often equals basis dimension for Gradient based algorithms), this number is lower by orders of magnitude, allowing CRAB to efficiently optimize smooth pulses with realistic experimental constraints. It is important to point out, that CRAB does not make any suggestion on the basis function to be used. The basis must be chosen carefully considered, taking into account a priori knowledge of the system (such as symmetries, magnitudes of scales,...) and solution (e.g. sign, smoothness, bang-bang behavior, singularities, maximum excursion or rate of change,....). By doing so, this algorithm allows for native integration of experimental constraints such as maximum frequencies allowed, maximum amplitude, smooth ramping up and down of the pulse and many more. Moreover initial guesses, if they are available, can (however not have to) be included to speed up convergence.

As mentioned in the GRAPE paragraph, for CRAB local minima arising from algorithmic design can occur, too. However, for CRAB a 'dressed' version has recently been introduced *[Rach15]* that allows to escape local minima.

For some control objectives and/or dynamical quantum descriptions, it is either not possible to derive the gradient for the cost functional with respect to each time slice or it is computationally expensive to do so. The same can apply for the necessary (reverse) propagation of the co-state. All this trouble does not occur within CRAB as those elements are not in use here. CRAB, instead, takes the time evolution as a black-box where the pulse goes as an input and the cost (e.g. infidelity) value will be returned as an output. This concept, on top, allows for direct integration in a closed loop experimental environment where both the preliminarily open loop optimization, as well as the final adoption, and integration to the lab (to account for modeling errors, experimental systematic noise, ...) can be done all in one, using this algorithm.

3.9.5 Optimal Quantum Control in QuTiP

There are two separate implementations of optimal control inside QuTiP. The first is an implementation of first order GRAPE, and is not further described here, but there are the example notebooks. The second is referred to as Qtrl (when a distinction needs to be made) as this was its name before it was integrated into QuTiP. Qtrl uses the Scipy optimize functions to perform the multi-variable optimisation, typically the L-BFGS-B method for GRAPE and Nelder-Mead for CRAB. The GRAPE implementation in Qtrl was initially based on the open-source package DYNAMO, which is a MATLAB implementation, and is described in *[DYNAMO]*. It has since been restructured and extended for flexibility and compatibility within QuTiP.

The rest of this section describes the Qtrl implementation and how to use it.

Object Model The Qtrl code is organised in a hierarchical object model in order to try and maximise configurability whilst maintaining some clarity. It is not necessary to understand the model in order to use the pulse optimisation functions, but it is the most flexible method of using Qtrl. If you just want to use a simple single function call interface, then jump to *Using the pulseoptim functions*
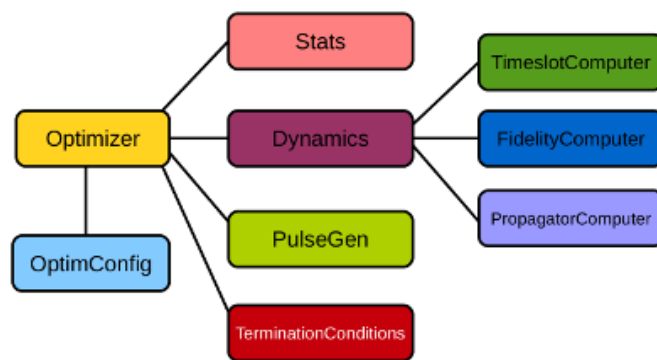
Fig. 3.4: Qtrl code object model.

The object's properties and methods are described in detail in the documentation, so that will not be repeated here.

OptimConfig The OptimConfig object is used simply to hold configuration parameters used by all the objects. Typically this is the subclass types for the other objects and parameters for the users specific requirements. The `loadparams` module can be used read parameter values from a configuration file.

Optimizer This acts as a wrapper to the `Scipy.optimize` functions that perform the work of the pulse optimisation algorithms. Using the main classes the user can specify which of the optimisation methods are to be used. There are subclasses specifically for the BFGS and L-BFGS-B methods. There is another subclass for using the CRAB algorithm.

Dynamics This is mainly a container for the lists that hold the dynamics generators, propagators, and time evolution operators in each timeslot. The combining of dynamics generators is also complete by this object. Different subclasses support a range of types of quantum systems, including closed systems with unitary dynamics, systems with quadratic Hamiltonians that have Gaussian states and symplectic transforms, and a general subclass that can be used for open system dynamics with Lindbladian operators.

PulseGen There are many subclasses of pulse generators that generate different types of pulses as the initial amplitudes for the optimisation. Often the goal cannot be achieved from all starting conditions, and then typically some kind of random pulse is used and repeated optimisations are performed until the desired infidelity is reached or the minimum infidelity found is reported. There is a specific subclass that is used by the CRAB algorithm to generate the pulses based on the basis coefficients that are being optimised.

TerminationConditions This is simply a convenient place to hold all the properties that will determine when the single optimisation run terminates. Limits can be set for number of iterations, time, and of course the target infidelity.

Stats Performance data are optionally collected during the optimisation. This object is shared to a single location to store, calculate and report run statistics.

FidelityComputer The subclass of the fidelity computer determines the type of fidelity measure. These are closely linked to the type of dynamics in use. These are also the most commonly user customised subclasses.

PropagatorComputer This object computes propagators from one timeslot to the next and also the propagator gradient. The options are using the spectral decomposition or Frechet derivative, as discussed above.

TimeslotComputer Here the time evolution is computed by calling the methods of the other computer objects.

OptimResult The result of a pulse optimisation run is returned as an object with properties for the outcome in terms of the infidelity, reason for termination, performance statistics, final evolution, and more.

3.9.6 Using the pulseoptim functions

The simplest method for optimising a control pulse is to call one of the functions in the `pulseoptim` module. This automates the creation and configuration of the necessary objects, generation of initial pulses, running the optimisation and returning the result. There are functions specifically for unitary dynamics, and also specifically for the CRAB algorithm (GRAPE is the default). The `optimise_pulse` function can in fact be used for unitary dynamics and / or the CRAB algorithm, the more specific functions simply have parameter names that are more familiar in that application.

A semi-automated method is to use the `create_optimizer_objects` function to generate and configure all the objects, then manually set the initial pulse and call the optimisation. This would be more efficient when repeating runs with different starting conditions.

3.10 Plotting on the Bloch Sphere

Important: Updated in QuTiP version 3.0.

3.10.1 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.

3.10.2 The Bloch and Bloch3d Classes

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

```
In [1]: b = Bloch()
```

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

```
>>> 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 [2]: b.show()
```
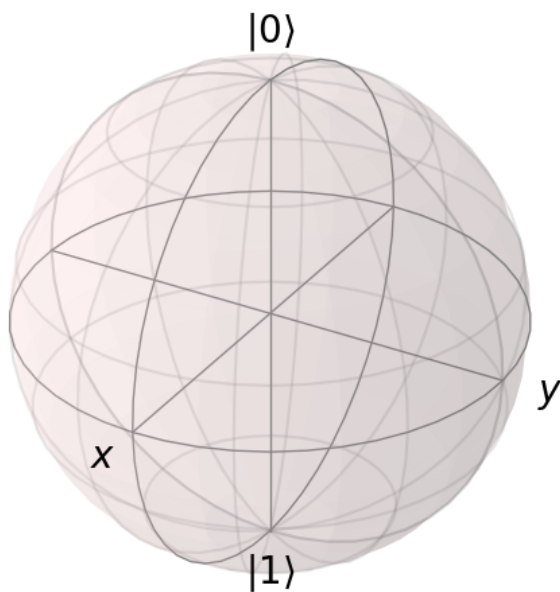
or
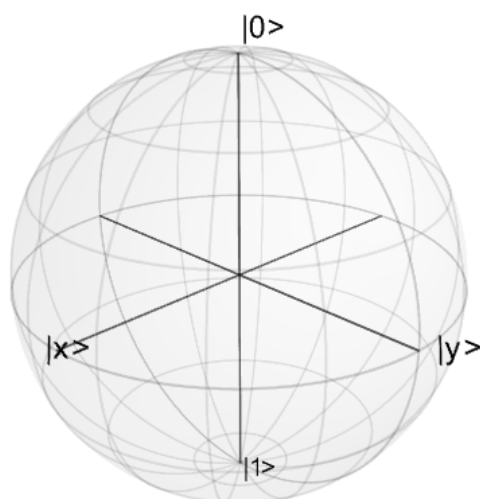
In addition to the show() command, the Bloch class has the following functions:

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 [3]: pnt = [1/np.sqrt(3),1/np.sqrt(3),1/np.sqrt(3)]

In [4]: b.add_points(pnt)

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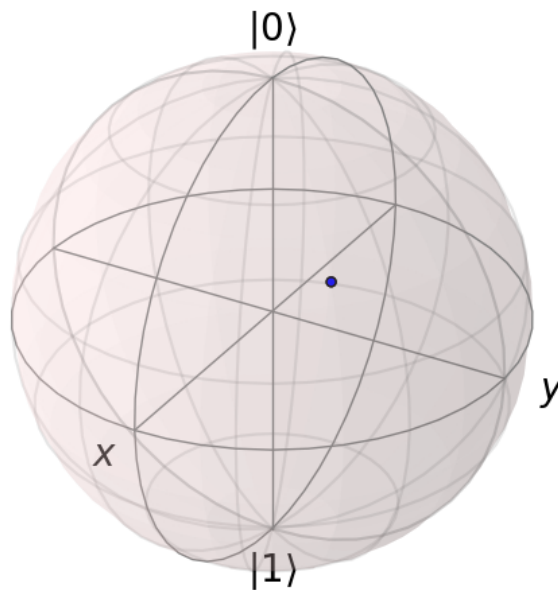

and then a single vector:

```
In [6]: vec = [0,1,0]

In [7]: b.add_vectors(vec)

In [8]: b.show()
```
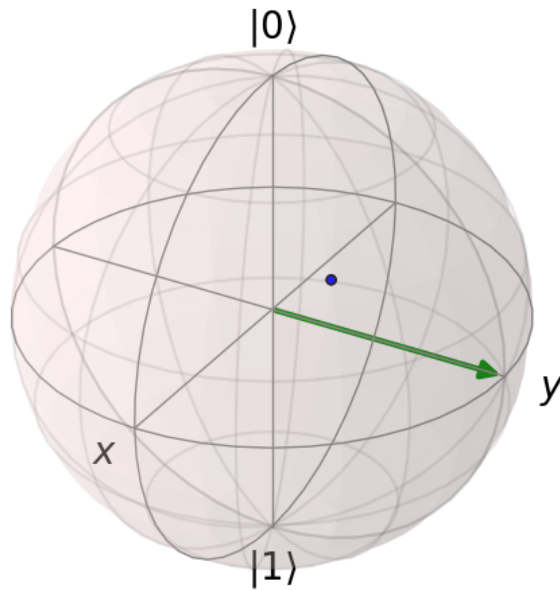
and then add another vector corresponding to the $|\text{up}\rangle$ state:

```
In [9]: up = basis(2,0)

In [10]: b.add_states(up)

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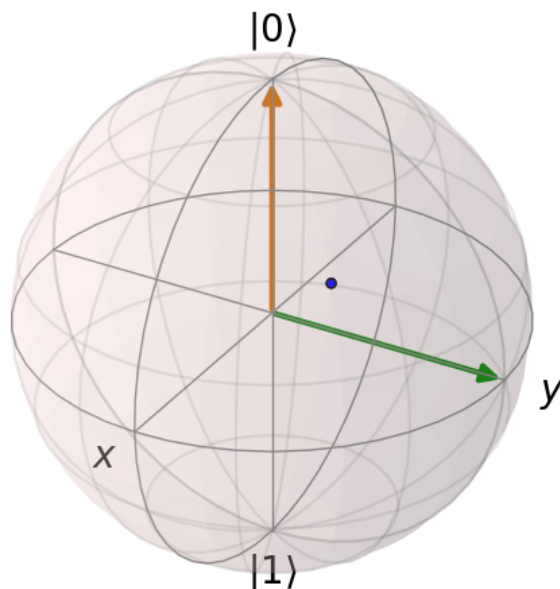

Notice that when we add more than a single vector (or data point), a different color will automatically be applied to the later data set (mod 4). In total, the code for constructing our Bloch sphere with one vector, one state, and a single data point is:

```
In [12]: pnt = [1/np.sqrt(3),1/np.sqrt(3),1/np.sqrt(3)]

In [13]: b.add_points(pnt)
```

3.10. Plotting on the Bloch Sphere   105

```
In [14]: b.add_vectors(vec)

In [15]: b.add_states(up)

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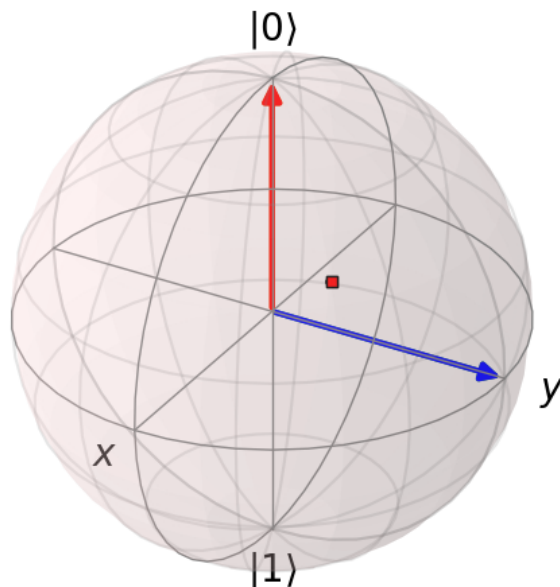

where we have removed the extra `show()` commands. Replacing `b=Bloch()` with `b=Bloch3d()` in the above code generates the following 3D Bloch sphere.
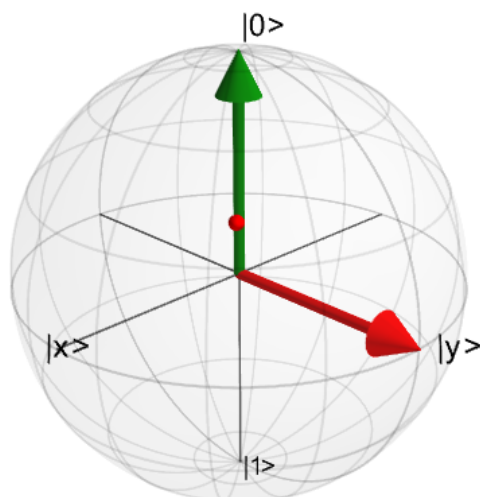
We can also plot multiple points, vectors, and states at the same time by passing list or arrays instead of individual elements. Before giving an example, we can use the *clear()* command to remove the current data from our Bloch sphere instead of creating a new instance:

```
In [17]: b.clear()

In [18]: b.show()
```
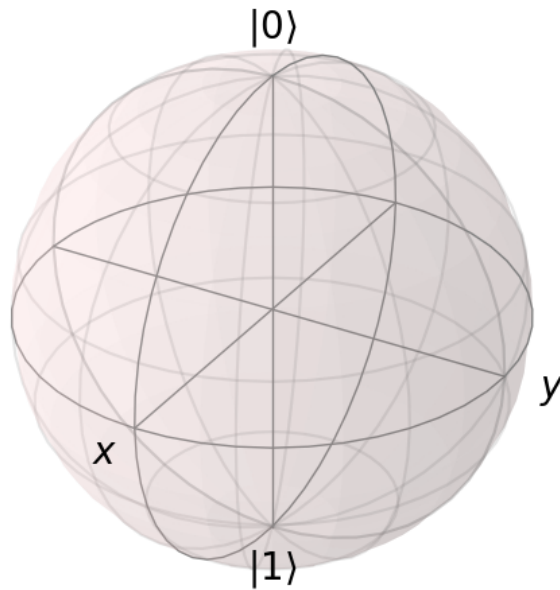
Now on the same Bloch sphere, we can plot the three states associated with the x, y, and z directions:

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

In [20]: y = (basis(2,0)+(0+1j)*basis(2,1)).unit()

In [21]: z = (basis(2,0)+(0+0j)*basis(2,1)).unit()

In [22]: b.add_states([x,y,z])

In [23]: b.show()
```
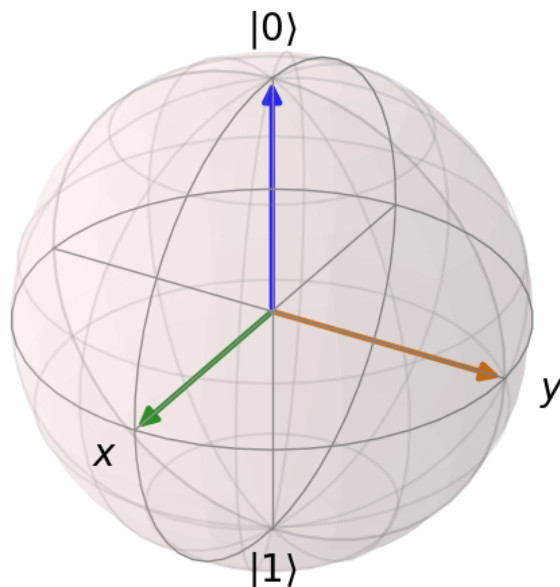

a similar method works for adding vectors:

```
In [24]: b.clear()
```

```
In [25]: vec = [[1,0,0],[0,1,0],[0,0,1]]

In [26]: b.add_vectors(vec)

In [27]: b.show()
```
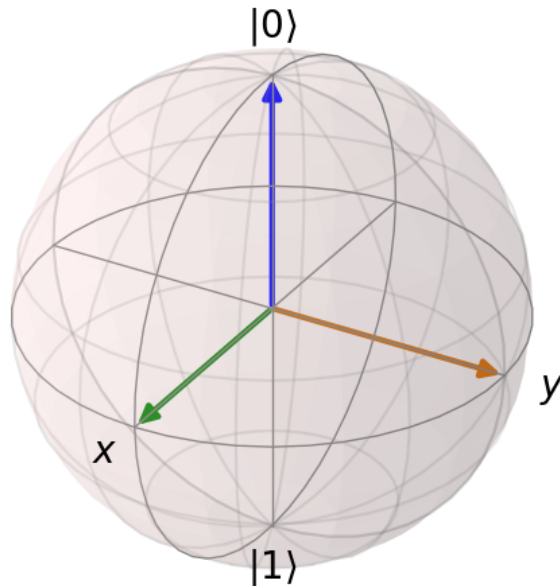

Adding multiple points to the Bloch sphere works slightly differently than adding multiple states or vectors. For example, lets add a set of 20 points around the equator (after calling *clear()*):

```
In [28]: xp = [np.cos(th) for th in np.linspace(0, 2*pi, 20)]

In [29]: yp = [np.sin(th) for th in np.linspace(0, 2*pi, 20)]

In [30]: zp = np.zeros(20)

In [31]: pnts = [xp, yp, zp]

In [32]: b.add_points(pnts)

In [33]: b.show()
```
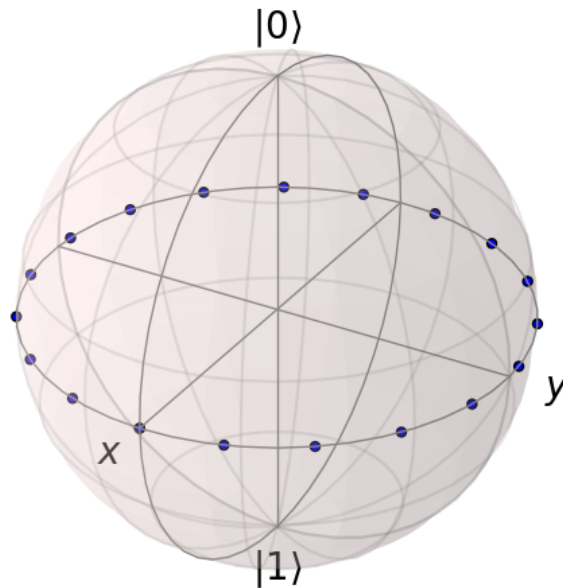
Notice that, in contrast to states or vectors, each point remains the same color as the initial point. This is because adding multiple data points using the add_points function is interpreted, by default, to correspond to a single data point (single qubit state) plotted at different times. This is very 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 [34]: xz = np.zeros(20)

In [35]: yz = [np.sin(th) for th in np.linspace(0, pi, 20)]

In [36]: zz = [np.cos(th) for th in np.linspace(0, pi, 20)]

In [37]: b.add_points([xz, yz, zz])

In [38]: b.show()
```
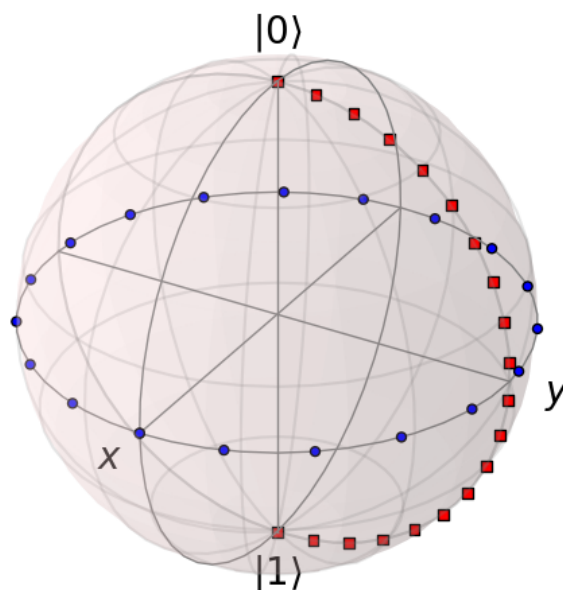
The color and shape of the data points is varied automatically by the Bloch class. Notice how the color and point markers change for each set of data. Again, we have had to call `add_points` twice because adding more than one set of multiple data points is *not* supported by the `add_points` function.

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 = [np.cos(th) for th in np.linspace(0, 2*pi, 20)]

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

In [41]: zp = np.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()
```
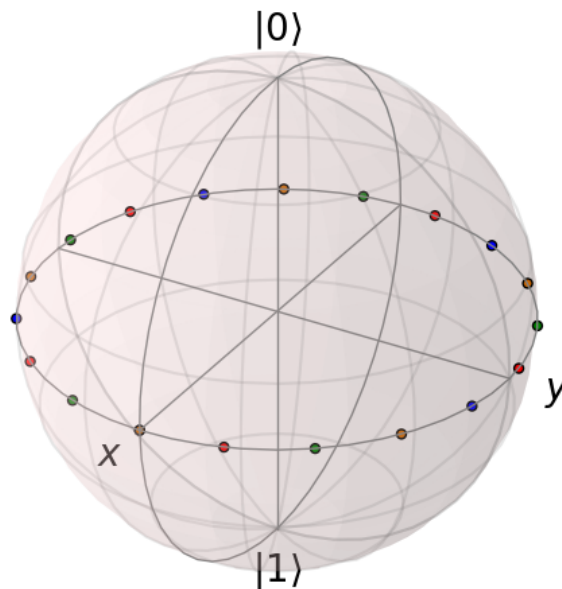

Now, the data points cycle through a variety of predefined colors. Now lets add another set of points, but this time we want the set to be a single color, representing say a qubit going from the $|up\rangle$ state to the $|down\rangle$ state in the y-z plane:

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

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

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

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

In [49]: b.show()
```
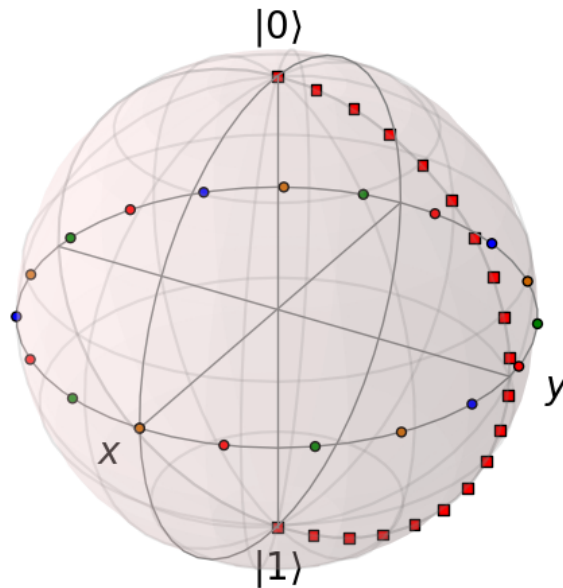
Again, the same plot can be generated using the `qutip.Bloch3d` class by replacing `Bloch` with `Bloch3d`:
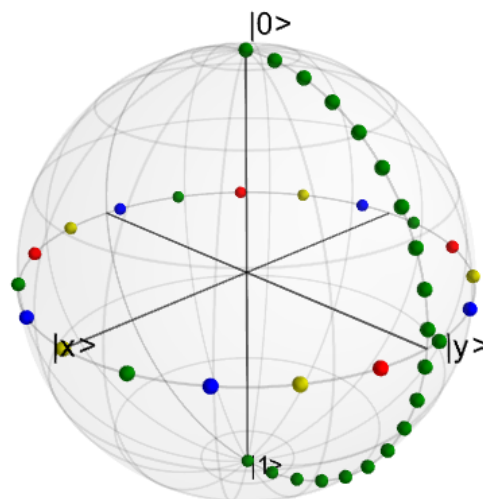
A more slick way of using this 'multi' color feature is also given in the example, where we set the color of the markers as a function of time.

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.

3.10.3 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
figsize:          [5, 5]
vector_color:     ['g', '#CC6600', 'b', 'r']
vector_width:     3
vector_style:     -|>
vector_mutation:  20
view:             [-60, 30]
xlabel:           ['$x$', '']
xlpos:            [1.2, -1.2]
ylabel:           ['$y$', '']
ylpos:            [1.2, -1.2]
zlabel:           ['$\\left|0\\right>$', '$\\left|1\\right>$']
zlpos:            [1.2, -1.2]
```

3.10. Plotting on the Bloch Sphere 113

3.10.4 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.

```python
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 range(len(sx)):
    b.clear()
    b.add_vectors([np.sin(theta),0,np.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([np.sin(theta),0,np.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

3.11 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 come 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.

3.11.1 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, $\text{Tr}[\rho a^\dagger a] = 2$. 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, np.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 [5]: fig, axes = plt.subplots(1, 3, figsize=(12,3))

In [6]: bar0 = axes[0].bar(np.arange(0, N)-.5, rho_coherent.diag())

In [7]: lbl0 = axes[0].set_title("Coherent state")

In [8]: lim0 = axes[0].set_xlim([-.5, N])

In [9]: bar1 = axes[1].bar(np.arange(0, N)-.5, rho_thermal.diag())

In [10]: lbl1 = axes[1].set_title("Thermal state")

In [11]: lim1 = axes[1].set_xlim([-.5, N])

In [12]: bar2 = axes[2].bar(np.arange(0, N)-.5, rho_fock.diag())

In [13]: lbl2 = axes[2].set_title("Fock state")

In [14]: lim2 = axes[2].set_xlim([-.5, N])

In [15]: plt.show()
```
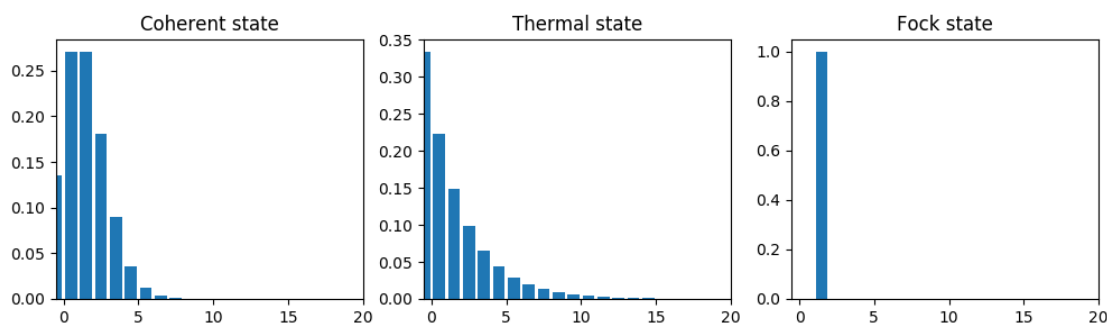

All these states correspond to an average of two photons, but by visualizing the photon distribution in Fock basis the differences between these states are easily appreciated.

One frequently need to visualize the Fock-distribution in the way described above, so QuTiP provides a conve-

nience function for doing this, see `qutip.visualization.plot_fock_distribution`, and the following example:

```
In [16]: fig, axes = plt.subplots(1, 3, figsize=(12,3))

In [17]: plot_fock_distribution(rho_coherent, fig=fig, ax=axes[0], title="Coherent
→state");

In [18]: plot_fock_distribution(rho_thermal, fig=fig, ax=axes[1], title="Thermal
→state");

In [19]: plot_fock_distribution(rho_fock, fig=fig, ax=axes[2], title="Fock state");

In [20]: fig.tight_layout()

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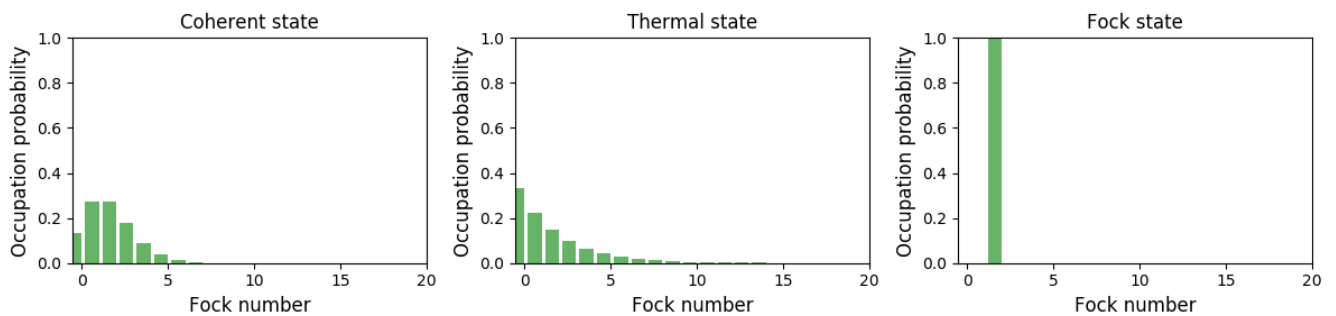

3.11.2 Quasi-probability distributions

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 [22]: xvec = np.linspace(-5,5,200)

In [23]: W_coherent = wigner(rho_coherent, xvec, xvec)

In [24]: W_thermal = wigner(rho_thermal, xvec, xvec)

In [25]: W_fock = wigner(rho_fock, xvec, xvec)

In [26]: # plot the results

In [27]: fig, axes = plt.subplots(1, 3, figsize=(12,3))
```

```
In [28]: cont0 = axes[0].contourf(xvec, xvec, W_coherent, 100)

In [29]: lbl0 = axes[0].set_title("Coherent state")

In [30]: cont1 = axes[1].contourf(xvec, xvec, W_thermal, 100)

In [31]: lbl1 = axes[1].set_title("Thermal state")

In [32]: cont0 = axes[2].contourf(xvec, xvec, W_fock, 100)

In [33]: lbl2 = axes[2].set_title("Fock state")

In [34]: plt.show()
```
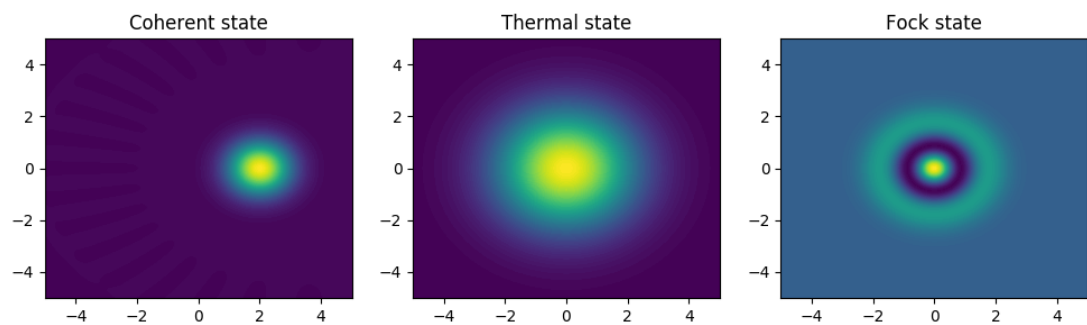

Custom Color Maps

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 [35]: import matplotlib as mpl

In [36]: from matplotlib import cm

In [37]: psi = (basis(10, 0) + basis(10, 3) + basis(10, 9)).unit()

In [38]: xvec = np.linspace(-5, 5, 500)

In [39]: W = wigner(psi, xvec, xvec)

In [40]: wmap = wigner_cmap(W)   # Generate Wigner colormap

In [41]: nrm = mpl.colors.Normalize(-W.max(), W.max())

In [42]: fig, axes = plt.subplots(1, 2, figsize=(10, 4))

In [43]: plt1 = axes[0].contourf(xvec, xvec, W, 100, cmap=cm.RdBu, norm=nrm)

In [44]: axes[0].set_title("Standard Colormap");

In [45]: cb1 = fig.colorbar(plt1, ax=axes[0])

In [46]: plt2 = axes[1].contourf(xvec, xvec, W, 100, cmap=wmap)   # Apply Wigner␣
↪colormap
```

```
In [47]: axes[1].set_title("Wigner Colormap");

In [48]: cb2 = fig.colorbar(plt2, ax=axes[1])

In [49]: fig.tight_layout()

In [50]: plt.show()
```
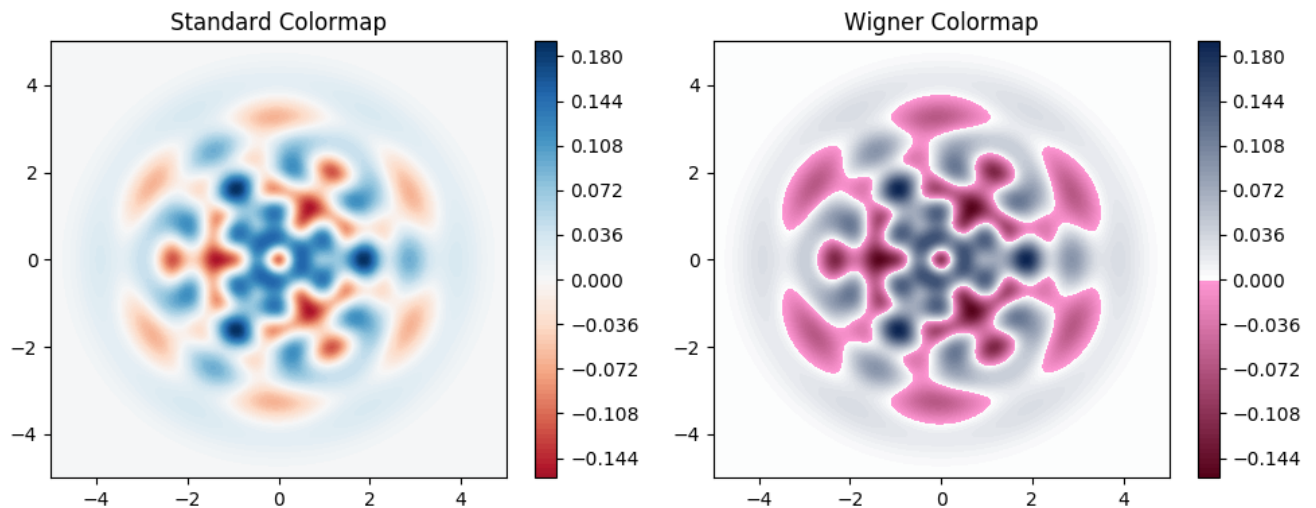

Husimi Q-function

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 [51]: Q_coherent = qfunc(rho_coherent, xvec, xvec)

In [52]: Q_thermal = qfunc(rho_thermal, xvec, xvec)

In [53]: Q_fock = qfunc(rho_fock, xvec, xvec)

In [54]: fig, axes = plt.subplots(1, 3, figsize=(12,3))

In [55]: cont0 = axes[0].contourf(xvec, xvec, Q_coherent, 100)

In [56]: lbl0 = axes[0].set_title("Coherent state")

In [57]: cont1 = axes[1].contourf(xvec, xvec, Q_thermal, 100)

In [58]: lbl1 = axes[1].set_title("Thermal state")

In [59]: cont0 = axes[2].contourf(xvec, xvec, Q_fock, 100)

In [60]: lbl2 = axes[2].set_title("Fock state")

In [61]: plt.show()
```
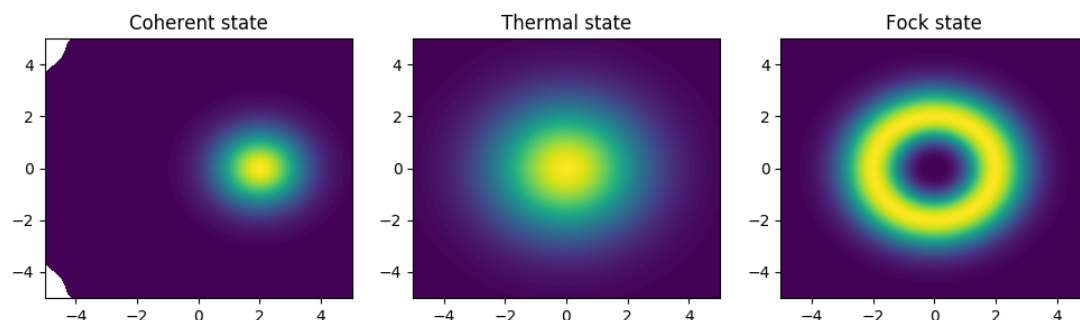
3.11.3 Visualizing operators

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 [62]: N = 5

In [63]: a = tensor(destroy(N), qeye(2))

In [64]: b = tensor(qeye(N), destroy(2))

In [65]: sx = tensor(qeye(N), sigmax())

In [66]: H = a.dag() * a + sx - 0.5 * (a * b.dag() + a.dag() * b)

In [67]: # visualize H

In [68]: lbls_list = [[str(d) for d in range(N)], ["u", "d"]]

In [69]: xlabels = []

In [70]: 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 [71]: fig, ax = matrix_histogram(H, xlabels, xlabels, limits=[-4,4])

In [72]: ax.view_init(azim=-55, elev=45)

In [73]: plt.show()
```
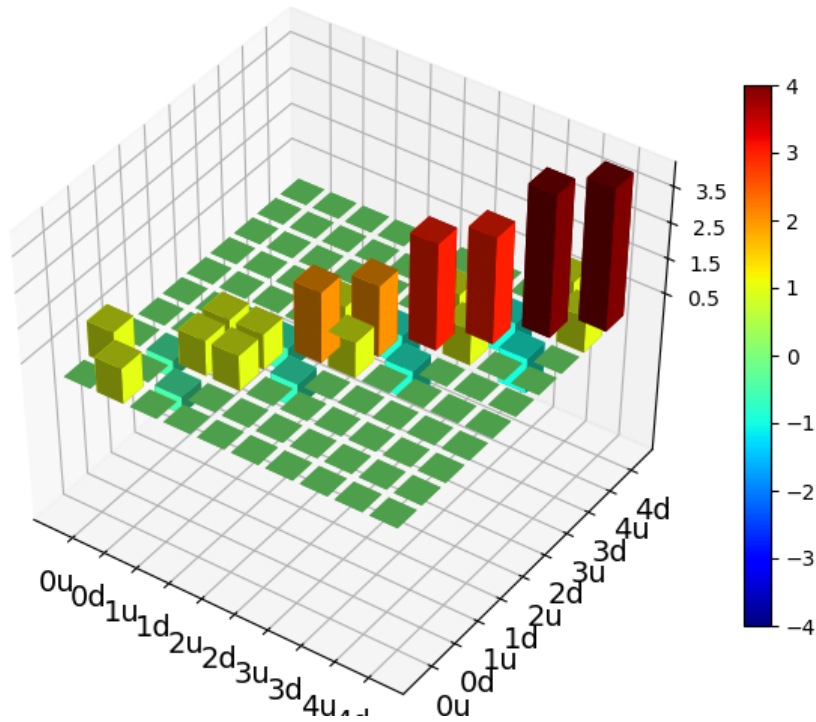
Similarly, we can use the function *qutip.visualization.hinton*, which is used below to visualize the corresponding steadystate density matrix:

```
In [74]: rho_ss = steadystate(H, [np.sqrt(0.1) * a, np.sqrt(0.4) * b.dag()])

In [75]: fig, ax = hinton(rho_ss) # xlabels=xlabels, ylabels=xlabels)

In [76]: plt.show()
```
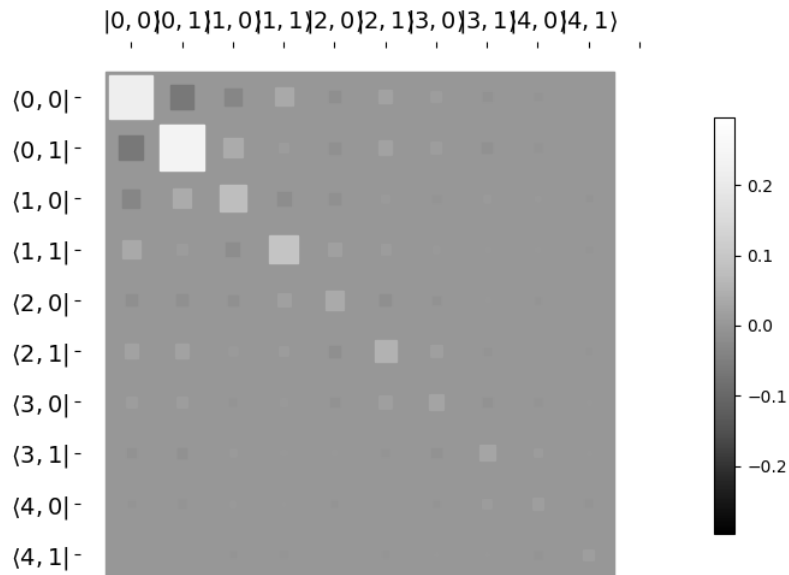
3.11.4 Quantum process tomography

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. *[Moh08]* for more details), consider a process that is described by quantum map $\epsilon(\rho_{\text{in}}) = \rho_{\text{out}}$, which can be written

$$\epsilon(\rho_{\text{in}}) = \rho_{\text{out}} = \sum_{i}^{N^2} A_i \rho_{\text{in}} A_i^\dagger, \tag{3.20}$$

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 $\text{Tr}[B_i^\dagger B_j] = N\delta_{ij}$, we can write the map as

$$\epsilon(\rho_{\text{in}}) = \rho_{\text{out}} = \sum_{mn} \chi_{mn} B_m \rho_{\text{in}} B_n^\dagger. \tag{3.21}$$

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 $B_m \rho_{\text{in}} B_n^\dagger$ contributes to ρ_{out}.

In a numerical simulation of a quantum process we usually do not have access to the quantum map in the form Eq. (3.20). 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

$$\epsilon(\tilde{\rho}_{\text{in}}) = U\tilde{\rho}_{\text{in}} = \tilde{\rho}_{\text{out}}$$

where $\tilde{\rho}$ is the vector representation of the density matrix ρ. If we write Eq. (3.21) in superoperator form as well we obtain

$$\tilde{\rho}_{\text{out}} = \sum_{mn} \chi_{mn} \tilde{B}_m \tilde{B}_n^\dagger \tilde{\rho}_{\text{in}} = U\tilde{\rho}_{\text{in}}.$$

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 [77]: 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 [78]: 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 [79]: op_basis = [[qeye(2), sigmax(), sigmay(), sigmaz()]] * 2

In [80]: 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 [81]: chi = qpt(U_rho, op_basis)

In [82]: fig = qpt_plot_combined(chi, op_label, r'$i$SWAP')

In [83]: plt.show()
```
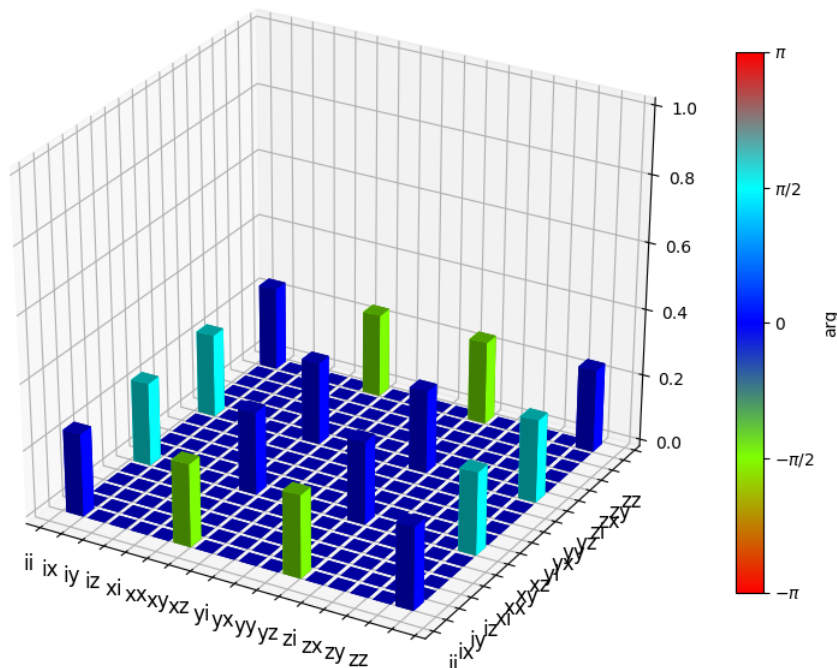

For a slightly more advanced example, where the density matrix propagator is calculated from the dynamics of a system defined by its Hamiltonian and collapse operators using the function `qutip.propagator.propagator`, see notebook "Time-dependent master equation: Landau-Zener transitions" on the tutorials section on the QuTiP web site.

3.12 Parallel computation

3.12.1 Parallel map and parallel for-loop

Often one is interested in the output of a given function as a single-parameter is varied. For instance, we can calculate the steady-state response of our system 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.parallel.parallel_map` function or the `qutip.parallel.parfor` (parallel-for-loop) function.

To use the these functions we need to define a function of one or more variables, and the range over which one of these variables are to be evaluated. 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]
```

or

```
In [6]: result = parallel_map(func1, range(10))

In [7]: result_array = np.array(result)

In [8]: print(result_array[:, 0])  # == a
[0 1 2 3 4 5 6 7 8 9]

In [9]: print(result_array[:, 1])  # == b
\\\\\\\\\\\\\\\\\\\\\\\[ 0  1  4  9 16 25 36 49 64 81]

In [10]: print(result_array[:, 2])  # == c
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\[  0   1   8  27  64 125 216
→343 512 729]
```

Note that the return values are arranged differently for the `qutip.parallel.parallel_map` and the `qutip.parallel.parfor` functions, as illustrated below. In particular, the return value of `qutip.parallel.parallel_map` is not enforced to be NumPy arrays, which can avoid unnecessary copying if all that is needed is to iterate over the resulting list:

```
In [11]: result = parfor(func1, range(5))

In [12]: print(result)
[array([0, 1, 2, 3, 4]), array([ 0,  1,  4,  9, 16]), array([ 0,  1,  8, 27, 64])]

In [13]: result = parallel_map(func1, range(5))

In [14]: print(result)
[(0, 0, 0), (1, 1, 1), (2, 4, 8), (3, 9, 27), (4, 16, 64)]
```

The `qutip.parallel.parallel_map` and `qutip.parallel.parfor` functions are not limited to just numbers, but also works for a variety of outputs:

```
In [15]: def func2(x): return x, Qobj(x), 'a' * x

In [16]: a, b, c = parfor(func2, range(5))

In [17]: print(a)
[0 1 2 3 4]

In [18]: print(b)
\\\\\\\\\\\\[ Quantum object: dims = [[1], [1]], shape = (1, 1), type = bra
Qobj data =
[[ 0.]]
 Quantum object: dims = [[1], [1]], shape = (1, 1), type = bra
Qobj data =
[[ 1.]]
 Quantum object: dims = [[1], [1]], shape = (1, 1), type = bra
Qobj data =
[[ 2.]]
 Quantum object: dims = [[1], [1]], shape = (1, 1), type = bra
Qobj data =
[[ 3.]]
 Quantum object: dims = [[1], [1]], shape = (1, 1), type = bra
Qobj data =
[[ 4.]]]

In [19]: print(c)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
→'' 'a' 'aa' 'aaa' 'aaaa']
```

Note: New in QuTiP 3.

One can also define functions with **multiple** input arguments and even keyword arguments. Here the `qutip.parallel.parallel_map` and `qutip.parallel.parfor` functions behaves differently: While `qutip.parallel.parallel_map` only iterate over the values *arguments*, the `qutip.parallel.parfor` function simultaneously iterates over all arguments:

```
In [20]: def sum_diff(x, y, z=0): return x + y, x - y, z

In [21]: parfor(sum_diff, [1, 2, 3], [4, 5, 6], z=5.0)
Out[21]: [array([5, 7, 9]), array([-3, -3, -3]), array([ 5.,  5.,  5.])]

In [22]: parallel_map(sum_diff, [1, 2, 3], task_args=(np.array([4, 5, 6]),), task_
→kwargs=dict(z=5.0))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[22]:
[(array([5, 6, 7]), array([-3, -4, -5]), 5.0),
 (array([6, 7, 8]), array([-2, -3, -4]), 5.0),
 (array([7, 8, 9]), array([-1, -2, -3]), 5.0)]
```

Note that the keyword arguments can be anything you like, but the keyword values are **not** iterated over. The keyword argument *num_cpus* is reserved as it sets the number of CPU's used by parfor. By default, this value is set to the total number of physical processors on your system. You can change this number to a lower value, however setting it higher than the number of CPU's will cause a drop in performance. In `qutip.parallel.parallel_map`, keyword arguments to the task function are specified using *task_kwargs* argument, so there is no special reserved keyword arguments.

The `qutip.parallel.parallel_map` function also supports progressbar, using the keyword argument *progress_bar* which can be set to *True* or to an instance of `qutip.ui.progressbar.BaseProgressBar`. There is a function called `qutip.parallel.serial_map` that works as a non-parallel drop-in replacement for `qutip.parallel.parallel_map`, which allows easy switching between serial and parallel computation.

3.12. Parallel computation 125

```
In [23]: import time

In [24]: def func(x): time.sleep(1)

In [25]: result = parallel_map(func, range(50), progress_bar=True)
10.0%. Run time:    2.02s. Est. time left: 00:00:00:18
20.0%. Run time:    3.03s. Est. time left: 00:00:00:12
30.0%. Run time:    4.03s. Est. time left: 00:00:00:09
40.0%. Run time:    5.03s. Est. time left: 00:00:00:07
50.0%. Run time:    7.03s. Est. time left: 00:00:00:07
60.0%. Run time:    8.03s. Est. time left: 00:00:00:05
70.0%. Run time:    9.03s. Est. time left: 00:00:00:03
80.0%. Run time:   10.04s. Est. time left: 00:00:00:02
90.0%. Run time:   12.04s. Est. time left: 00:00:00:01
100.0%. Run time:  13.04s. Est. time left: 00:00:00:00
Total run time:   13.06s
```

Parallel processing is useful for repeated tasks such as generating plots corresponding to the dynamical evolution of your system, or simultaneously simulating different parameter configurations.

3.12.2 IPython-based parallel_map

Note: New in QuTiP 3.

When QuTiP is used with IPython interpreter, there is an alternative parallel for-loop implementation in the QuTiP module *qutip.ipynbtools*, see *qutip.ipynbtools.parallel_map*. The advantage of this parallel_map 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.

3.13 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).

3.13.1 Storing and loading QuTiP objects

To store and load arbitrary QuTiP related objects (*qutip.Qobj*, *qutip.solver.Result*, 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 = [np.sqrt(0.5) * a, np.sqrt(0.
→25) * a.dag()]

In [2]: rho_ss = steadystate(H, c_ops)
```

The steadystate density matrix *rho_ss* is an instance of `qutip.Qobj`. It can be stored to a file *steadystate.qu* using

```
In [3]: qsave(rho_ss, 'steadystate')

In [4]: ls *.qu
density_matrix_vs_time.qu  steadystate.qu
```

and it can later be loaded again, and used in further calculations:

```
In [5]: rho_ss_loaded = qload('steadystate')
Loaded Qobj object:
Quantum object: dims = [[10], [10]], shape = (10, 10), type = oper, isHerm = True


In [6]: a = destroy(10)

In [7]: expect(a.dag() * a, rho_ss_loaded)
Out[7]: 0.9902248289345064
```

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 [8]: a = destroy(10); H = a.dag() * a ; c_ops = [np.sqrt(0.5) * a, np.sqrt(0.
→25) * a.dag()]

In [9]: psi0 = rand_ket(10)

In [10]: times = np.linspace(0, 10, 10)

In [11]: dm_list = mesolve(H, psi0, times, c_ops, [])

In [12]: qsave(dm_list, 'density_matrix_vs_time')
```

And it can then be loaded and used again, for example in an other program:

```
In [13]: dm_list_loaded = qload('density_matrix_vs_time')
Loaded Result object:
Result object with mesolve data.
--------------------------------
states = True
num_collapse = 0

In [14]: a = destroy(10)

In [15]: expect(a.dag() * a, dm_list_loaded.states)
Out[15]:
array([ 3.40657352,  2.74980446,  2.29441406,  1.96186716,  1.71601185,
        1.5332011 ,  1.39683111,  1.29490432,  1.21862519,  1.16149242])
```

3.13.2 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.fileio.file_data_store` and `qutip.fileio.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.fileio.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.fileio.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 [16]: a = destroy(10); H = a.dag() * a ; c_ops = [np.sqrt(0.5) * a, np.sqrt(0.
→25) * a.dag()]

In [17]: psi0 = rand_ket(10)

In [18]: times = np.linspace(0, 100, 100)

In [19]: medata = mesolve(H, psi0, times, c_ops, [a.dag() * a, a + a.dag(), -1j *
→(a - a.dag())])

In [20]: shape(medata.expect)
Out[20]: (3, 100)

In [21]: shape(times)
\\\\\\\\\\\\\\\\\\\\\Out[21]: (100,)

In [22]: output_data = np.vstack((times, medata.expect))   # join time and expt
→data

In [23]: file_data_store('expect.dat', output_data.T) # Note the .T for transpose!

In [24]: ls *.dat
expect.dat

In [25]: !head expect.dat
\\\\\\\\\\\\\# Generated by QuTiP: 100x4 complex matrix in decimal format [','
→separated values].
0.0000000000+0.0000000000j,4.6283109309+0.0000000000j,-0.7139985949+0.0000000000j,-
→1.9302925449+0.0000000000j
1.0101010101+0.0000000000j,3.7394716905+0.0000000000j,-1.5795745709+0.0000000000j,-
→0.2887283809+0.0000000000j
2.0202020202+0.0000000000j,3.0806355195+0.0000000000j,-0.8933912070+0.0000000000j,
→1.0022231476+0.0000000000j
3.0303030303+0.0000000000j,2.5873637218+0.0000000000j,0.3268461698+0.0000000000j,1.
→0991113962+0.0000000000j
4.0404040404+0.0000000000j,2.2137873173+0.0000000000j,0.9536851397+0.0000000000j,0.
→2615765022+0.0000000000j
5.0505050505+0.0000000000j,1.9291690866+0.0000000000j,0.6303772657+0.0000000000j,-
→0.5811924392+0.0000000000j
6.0606060606+0.0000000000j,1.7115553526+0.0000000000j,-0.1377033059+0.0000000000j,-
→0.7330049151+0.0000000000j
7.0707070707+0.0000000000j,1.5447938440+0.0000000000j,-0.6052011708+0.0000000000j,-
→0.2374842702+0.0000000000j
8.0808080808+0.0000000000j,1.4168074420+0.0000000000j,-0.4562520460+0.0000000000j,
→0.3374981829+0.0000000000j
```

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 [26]: file_data_store('expect.dat', output_data.T, numtype="real")

In [27]: !head -n5 expect.dat
# Generated by QuTiP: 100x4 real matrix in decimal format [',' separated values].
0.0000000000,4.6283109309,-0.7139985949,-1.9302925449
1.0101010101,3.7394716905,-1.5795745709,-0.2887283809
2.0202020202,3.0806355195,-0.8933912070,1.0022231476
3.0303030303,2.5873637218,0.3268461698,1.0991113962
```

and if we prefer scientific notation we can request that using the *numformat="exp"* option

```
In [28]: file_data_store('expect.dat', output_data.T, numtype="real", numformat=
→"exp")

In [29]: !head -n 5 expect.dat
# Generated by QuTiP: 100x4 real matrix in exp format [',' separated values].
0.0000000000e+00,4.6283109309e+00,-7.1399859487e-01,-1.9302925449e+00
1.0101010101e+00,3.7394716905e+00,-1.5795745709e+00,-2.8872838088e-01
2.0202020202e+00,3.0806355195e+00,-8.9339120701e-01,1.0022231476e+00
3.0303030303e+00,2.5873637218e+00,3.2684616978e-01,1.0991113962e+00
```

Loading data previously stored using `qutip.fileio.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.fileio.file_data_read`, which only takes the filename as mandatory argument.

```
In [30]: input_data = file_data_read('expect.dat')

In [31]: shape(input_data)
Out[31]: (100, 4)

In [32]: plot(input_data[:,0], input_data[:,1]);  # plot the data
```
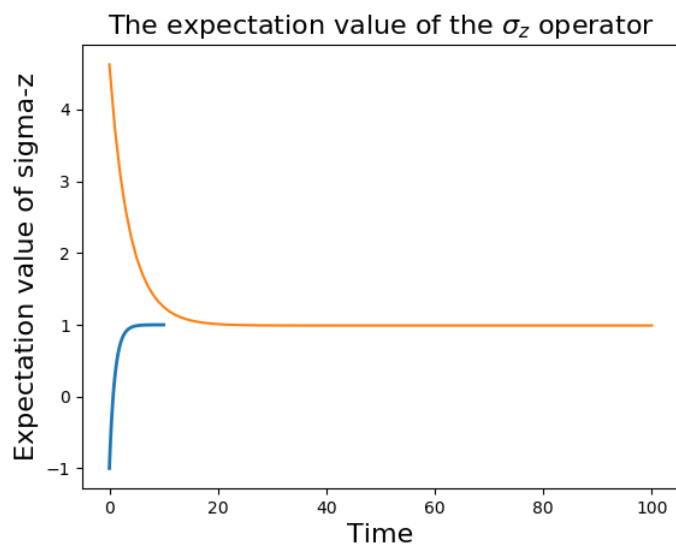

(If a particularly obscure choice of deliminator was used it might be necessary to use the optional second argument, for example *sep="_"* if _ is the deliminator).

3.14 Generating Random Quantum States & Operators

QuTiP includes a collection of random state, unitary and channel generators for simulations, Monte Carlo evaluation, theorem evaluation, and code testing. Each of these objects can be sampled from one of several different

distributions including the default distributions used by QuTiP versions prior to 3.2.0.

For example, a random Hermitian operator can be sampled by calling *rand_herm* function:

```
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        -0.61443139+0.5565611j   0.00000000+0.j
  -0.07695716+0.33797916j  0.00000000+0.j        ]
 [-0.61443139-0.5565611j   0.00000000+0.j        -0.58658376+0.47851684j
   0.00000000+0.j          0.29087032-0.04403322j]
 [ 0.00000000+0.j        -0.58658376-0.47851684j -1.48253672+0.j
   0.14821531-0.00900551j  0.21133530-0.4525358j ]
 [-0.07695716-0.33797916j  0.00000000+0.j         0.14821531+0.00900551j
   0.00000000+0.j          0.08601466-0.31579196j]
 [ 0.00000000+0.j         0.29087032+0.04403322j  0.21133530+0.4525358j
   0.08601466+0.31579196j  0.00000000+0.j        ]]
```

Random Variable Type	Sampling Functions	Dimensions
State vector (`ket`)	*rand_ket*, *rand_ket_haar*	$N \times 1$
Hermitian operator (`oper`)	*rand_herm*	$N \times 1$
Density operator (`oper`)	*rand_dm*, *rand_dm_hs*, *rand_dm_ginibre*	$N \times N$
Unitary operator (`oper`)	*rand_unitary*, *rand_unitary_haar*	$N \times N$
CPTP channel (`super`)	*rand_super*, *rand_super_bcsz*	$(N \times N) \times (N \times N)$

In all cases, these functions can be called with a single parameter N that dimension of the relevant Hilbert space. The optional `dims` keyword argument allows for the dimensions of a random state, unitary or channel to be broken down into subsystems.

```
In [2]: print rand_super_bcsz(7).dims
  File "<ipython-input-2-aef77a269b21>", line 1
    print rand_super_bcsz(7).dims
                             ^
SyntaxError: invalid syntax


In [3]: print rand_super_bcsz(6, dims=[[[2, 3], [2, 3]], [[2, 3], [2, 3]]]).dims
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪ File "<ipython-input-3-fbc42bdd7e86>", line 1
    print rand_super_bcsz(6, dims=[[[2, 3], [2, 3]], [[2, 3], [2, 3]]]).dims
                            ^
SyntaxError: invalid syntax
```

Several of the distributions supported by QuTiP support additional parameters as well, namely *density* and *rank*. In particular, the *rand_herm* and *rand_dm* functions return quantum objects such that a fraction of the elements are identically equal to zero. The ratio of nonzero elements is passed as the `density` keyword argument. By contrast, the *rand_dm_ginibre* and *rand_super_bcsz* take as an argument the rank of the generated object, such that passing `rank=1` returns a random pure state or unitary channel, respectively. Passing `rank=None` specifies that the generated object should be full-rank for the given dimension.

For example,

```
In [4]: rand_dm(5, density=0.5)
Out[4]:
Quantum object: dims = [[5], [5]], shape = (5, 5), type = oper, isherm = True
Qobj data =
[[ 0.45350481+0.j          0.00000000+0.j         -0.01385138-0.08093636j
  -0.02190011+0.03272436j  0.00000000+0.j        ]
 [ 0.00000000+0.j          0.29181490+0.j          0.00000000+0.j
   0.21426922-0.0356683j   0.00000000+0.j        ]
 [-0.01385138+0.08093636j  0.00000000+0.j          0.04170053+0.j
  -0.01450458-0.01376581j  0.00000000+0.j        ]
 [-0.02190011-0.03272436j  0.21426922+0.0356683j  -0.01450458+0.01376581j
   0.21297977+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        ]]

In [5]: rand_dm_ginibre(5, rank=2)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
Quantum object: dims = [[5], [5]], shape = (5, 5), type = oper, isherm = True
Qobj data =
[[ 0.14066942 -1.01991394e-18j -0.08005758 +2.58808924e-02j
   0.11820086 +1.53135504e-01j  0.00792885 +1.49036180e-02j
   0.02169210 +2.07400940e-01j]
 [-0.08005758 -2.58808924e-02j  0.08398545 -1.22838721e-18j
  -0.00748508 -7.37014680e-02j  0.05206634 -8.96589039e-03j
   0.06426153 -1.29411293e-01j]
 [ 0.11820086 -1.53135504e-01j -0.00748508 +7.37014680e-02j
   0.33251664 -3.12769705e-18j  0.07446297 -5.14868086e-02j
   0.27239256 +1.03521226e-01j]
 [ 0.00792885 -1.49036180e-02j  0.05206634 +8.96589039e-03j
   0.07446297 +5.14868086e-02j  0.08815831 +2.19682196e-18j
   0.08447533 -3.53205371e-03j]
 [ 0.02169210 -2.07400940e-01j  0.06426153 +1.29411293e-01j
   0.27239256 -1.03521226e-01j  0.08447533 +3.53205371e-03j
   0.35467018 +3.17917623e-18j]]
```

See the API documentation: *Random Operators and States* for details.

> **Warning:** When using the `density` keyword argument, setting the density too low may result in not enough diagonal elements to satisfy trace constraints.

3.14.1 Random objects with a given eigen spectrum

Note: New in QuTiP 3.2

It is also possible to generate random Hamiltonian (`rand_herm`) and densitiy matrices (`rand_dm`) with a given eigen spectrum. This is done by passing an array of eigenvalues as the first argument to either function. For example,

```
In [6]: eigs = np.arange(5)

In [7]: H = rand_herm(eigs, density=0.5)

In [8]: H
Out[8]:
Quantum object: dims = [[5], [5]], shape = (5, 5), type = oper, isherm = True
Qobj data =
```

```
[[ 1.25000000 +2.77555756e-17j -0.25000000 -2.77555756e-17j
  -1.02918489 +2.56473139e-01j  0.00000000 +0.00000000e+00j
   0.14115753 +4.79660870e-01j]
 [-0.25000000 -2.77555756e-17j  1.25000000 +2.77555756e-17j
   1.02918489 -2.56473139e-01j  0.00000000 +0.00000000e+00j
  -0.14115753 -4.79660870e-01j]
 [-1.02918489 -2.56473139e-01j  1.02918489 +2.56473139e-01j
   1.50000000 +0.00000000e+00j  0.00000000 +0.00000000e+00j
  -0.02967610 -7.06483779e-01j]
 [ 0.00000000 +0.00000000e+00j  0.00000000 +0.00000000e+00j
   0.00000000 +0.00000000e+00j  3.00000000 +0.00000000e+00j
   0.00000000 +0.00000000e+00j]
 [ 0.14115753 -4.79660870e-01j -0.14115753 +4.79660870e-01j
  -0.02967610 +7.06483779e-01j  0.00000000 +0.00000000e+00j
   3.00000000 +0.00000000e+00j]]

In [9]: H.eigenenergies()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
↪
array([  2.36825980e-16,   1.00000000e+00,   2.00000000e+00,
         3.00000000e+00,   4.00000000e+00])
```

In order to generate a random object with a given spectrum QuTiP applies a series of random complex Jacobi rotations. This technique requires many steps to build the desired quantum object, and is thus suitable only for objects with Hilbert dimensionality $\lesssim 1000$.

3.14.2 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 [10]: rand_dm(4, 0.5, dims=[[2,2], [2,2]])
Out[10]:
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper, isherm = True
Qobj data =
[[ 0.00000000+0.j         0.00000000+0.j         0.00000000+0.j
   0.00000000+0.j        ]
 [ 0.00000000+0.j         0.67661788+0.j         0.15116885-0.27120685j
  -0.07200880+0.09761737j]
 [ 0.00000000+0.j         0.15116885+0.27120685j  0.26754373+0.j
  -0.05378799+0.05057372j]
 [ 0.00000000+0.j        -0.07200880-0.09761737j -0.05378799-0.05057372j
   0.05583839+0.j        ]]
```

3.15 Modifying Internal QuTiP Settings

3.15.1 User Accessible Parameters

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
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
atol	General tolerance	any *float* value > 0
num_cpus	Number of CPU's used for multi-processing.	*int* between 1 and # cpu's
debug	Show debug printouts.	True / False
openmp_thresh	NNZ matrix must have for OPENMP.	Int

3.15.2 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
```

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()
```

3.15.3 Persistent Settings

When QuTiP is imported, it looks for a file named `qutiprc` in a folder called `.qutip` 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]
auto_tidyup=True
auto_herm=True
auto_tidyup_atol=1e-12
num_cpus=4
debug=False
```

Note that the `openmp_thresh` value is automatically generatd by QuTiP. It is also possible to set a specific compiler for QuTiP to use when generating runtime Cython code for time-dependent problems. For example, the following section in the `qutiprc` file will set the compiler to be `clang-3.9`:

```
[compiler]
cc = clang-3.9
cxx = clang-3.9
```

Chapter 4

API documentation

This chapter contains automatically generated API documentation, including a complete list of QuTiP's public classes and functions.

4.1 Classes

4.1.1 Qobj

class Qobj(*inpt=None, dims=[[], []], shape=[], type=None, isherm=None, copy=True, fast=False, superrep=None*)
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).
>
> **copy** : bool
>
> > Flag specifying whether Qobj should get a copy of the input data, or use the original.
>
> **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.
type	(str) Type of quantum object: 'bra', 'ket', 'oper', 'operator-ket', 'operator-bra', or 'super'.
super-rep	(str) Representation used if *type* is 'super'. One of 'super' (Liouville form) or 'choi' (Choi matrix with tr = dimension).
isherm	(bool) Indicates if quantum object represents Hermitian operator.
iscp	(bool) Indicates if the quantum object represents a map, and if that map is completely positive (CP).
ishp	(bool) Indicates if the quantum object represents a map, and if that map is hermicity preserving (HP).
istp	(bool) Indicates if the quantum object represents a map, and if that map is trace preserving (TP).
iscptp	(bool) Indicates if the quantum object represents a map that is completely positive and trace preserving (CPTP).
isket	(bool) Indicates if the quantum object represents a ket.
isbra	(bool) Indicates if the quantum object represents a bra.
isoper	(bool) Indicates if the quantum object represents an operator.
issuper	(bool) Indicates if the quantum object represents a superoperator.
isoper-ket	(bool) Indicates if the quantum object represents an operator in column vector form.
isoper-bra	(bool) Indicates if the quantum object represents an operator in row vector form.

Methods

copy()	Create copy of Qobj
conj()	Conjugate of quantum object.
cosm()	Cosine of quantum object.
dag()	Adjoint (dagger) of quantum object.
dnorm()	Diamond norm of quantum operator.
dual_chan()	Dual channel of quantum object representing a CP map.
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(order='C')	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(norm='tr', sparse=False, tol=0, maxiter=100000)	Returns norm of a ket or an operator.
permute(order)	Returns composite qobj with indices reordered.
ptrace(sel)	Returns quantum object for selected dimensions after performing partial trace.
sinm()	Sine of quantum object.
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.
trunc_neg(method='clip')	Removes negative eigenvalues and returns a new Qobj that is a valid density operator.
unit(norm='tr', sparse=False, tol=0, maxiter=100000)	Returns normalized quantum object.

check_herm()
> Check if the quantum object is hermitian.

> > **Returns isherm** : bool

> > > Returns the new value of isherm property.

conj()
> Conjugate operator of quantum object.

copy()
> Create identical copy

cosm()
> Cosine of a quantum operator.

> Operator must be square.

> > **Returns oper** : qobj

> > > Matrix cosine of operator.

> > **Raises TypeError**

> > > Quantum object is not square.

Notes

Uses the Q.expm() method.

dag()
Adjoint operator of quantum object.

diag()
Diagonal elements of quantum object.

> **Returns diags** : array
>
> > Returns array of `real` values if operators is Hermitian, otherwise `complex` values are returned.

dnorm(*B=None*)
Calculates the diamond norm, or the diamond distance to another operator.

> **Parameters B** : Qobj or None
>
> > If B is not None, the diamond distance d(A, B) = dnorm(A - B) between this operator and B is returned instead of the diamond norm.
>
> **Returns d** : float
>
> > Either the diamond norm of this operator, or the diamond distance from this operator to B.

dual_chan()
Dual channel of quantum object representing a completely positive map.

eigenenergies(*sparse=False*, *sort='low'*, *eigvals=0*, *tol=0*, *maxiter=100000*)
Eigenenergies of a quantum object.

Eigenenergies (eigenvalues) 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*)
Eigenstates and eigenenergies.

Eigenstates and eigenenergies 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.

eliminate_states(*states_inds*, *normalize=False*)

Creates a new quantum object with states in state_inds eliminated.

Parameters states_inds : list of integer

The states that should be removed.

normalize : True / False

Weather or not the new Qobj instance should be normalized (default is False). For Qobjs that represents density matrices or state vectors normalized should probably be set to True, but for Qobjs that represents operators in for example an Hamiltonian, normalize should be False.

Returns q : Qobj

A new instance of *qutip.Qobj* that contains only the states corresponding to indices that are **not** in *state_inds*.

Notes

Experimental.

static evaluate(*qobj_list*, *t*, *args*)

Evaluate a time-dependent quantum object in list format. For example,

qobj_list = [H0, [H1, func_t]]

is evaluated to

Qobj(t) = H0 + H1 * func_t(t, args)

and

qobj_list = [H0, [H1, 'sin(w * t)']]

is evaluated to

Qobj(t) = H0 + H1 * sin(args['w'] * t)

> **Parameters** **qobj_list** : list
>
> > A nested list of Qobj instances and corresponding time-dependent coefficients.
>
> **t** : float
>
> > The time for which to evaluate the time-dependent Qobj instance.
>
> **args** : dictionary
>
> > A dictionary with parameter values required to evaluate the time-dependent Qobj intance.
>
> **Returns** **output** : Qobj
>
> > A Qobj instance that represents the value of qobj_list at time t.

expm(*method='dense'*)

Matrix exponential of quantum operator.

Input operator must be square.

> **Parameters** **method** : str {'dense', 'sparse'}
>
> > Use set method to use to calculate the matrix exponentiation. The available choices includes 'dense' and 'sparse'. Since the exponential of a matrix is nearly always dense, method='dense' is set as default.s
>
> **Returns** **oper** : qobj
>
> > Exponentiated quantum operator.
>
> **Raises** **TypeError**
>
> > Quantum operator is not square.

extract_states(*states_inds*, *normalize=False*)

Qobj with states in state_inds only.

> **Parameters** **states_inds** : list of integer
>
> > The states that should be kept.
>
> **normalize** : True / False
>
> > Weather or not the new Qobj instance should be normalized (default is False). For Qobjs that represents density matrices or state vectors normalized should probably be set to True, but for Qobjs that represents operators in for example an Hamiltonian, normalize should be False.
>
> **Returns** **q** : Qobj
>
> > A new instance of *qutip.Qobj* that contains only the states corresponding to the indices in *state_inds*.

Notes

Experimental.

full(*order='C'*, *squeeze=False*)

Dense array from quantum object.

> **Parameters** **order** : str {'C', 'F'}

Return array in C (default) or Fortran ordering.

squeeze : bool {False, True}

Squeeze output array.

Returns data : array

Array of complex data from quantum objects *data* attribute.

groundstate(*sparse=False*, *tol=0*, *maxiter=100000*, *safe=True*)

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).

safe : bool (default=True)

Check for degenerate ground state

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 the matrix element for the quantum object 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(*norm=None*, *sparse=False*, *tol=0*, *maxiter=100000*)

Norm of a quantum object.

Default norm is L2-norm for kets and trace-norm for operators. Other ket and operator norms may be specified using the *norm* and argument.

Parameters norm : str

Which norm to use for ket/bra vectors: L2 'l2', max norm 'max', or for operators: trace 'tr', Frobius 'fro', one 'one', or max 'max'.

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.

overlap(*state*)

Overlap between two state vectors.

Gives the overlap (scalar product) for the quantum object and *state* state vector.

Parameters state : qobj

Quantum object for a state vector of type 'ket' or 'bra'.

Returns overlap : complex

Complex valued overlap.

Raises TypeError

Can only calculate overlap between a bra and ket quantum objects.

permute(*order*)

Permutes a composite quantum object.

Parameters order : list/array

List specifying new tensor order.

Returns P : qobj

Permuted quantum object.

ptrace(*sel*)

Partial trace of the quantum object.

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.

sinm()
> Sine of a quantum operator.
>
> Operator must be square.
>
> > **Returns oper** : qobj
> >
> > > Matrix sine of operator.
> >
> > **Raises TypeError**
> >
> > > Quantum object is not square.

Notes

Uses the Q.expm() method.

sqrtm(*sparse=False*, *tol=0*, *maxiter=100000*)
> 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=None*)
> Removes small elements from the 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()
> 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*, *sparse=True*)
> Basis transform defined by 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.

>> **sparse** : bool

>>> Use sparse matrices when possible. Can be slower.

>> **Returns** **oper** : qobj

>>> Operator in new basis.

Notes

> This function is still in development.

trunc_neg(*method='clip'*)
> Truncates negative eigenvalues and renormalizes.

> Returns a new Qobj by removing the negative eigenvalues of this instance, then renormalizing to obtain a valid density operator.

>> **Parameters** **method** : str

>>> Algorithm to use to remove negative eigenvalues. "clip" simply discards negative eigenvalues, then renormalizes. "sgs" uses the SGS algorithm (doi:10/bb76) to find the positive operator that is nearest in the Shatten 2-norm.

>> **Returns** **oper** : qobj

>>> A valid density operator.

unit(*norm=None*, *sparse=False*, *tol=0*, *maxiter=100000*)
> Operator or state normalized to unity.

> Uses norm from Qobj.norm().

>> **Parameters** **norm** : str

>>> Requested norm for states / operators.

>> **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.

4.1.2 eseries

class eseries(*q=array([], dtype=object)*, *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`.

4.1.3 Bloch sphere

class Bloch(*fig=None*, *axes=None*, *view=None*, *figsize=None*, *background=False*)
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.
figsize	(list {[7,7]}) Figure 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 {[r'$\left|0\right>$',r'$\left|1\right>$']}) 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_annotation(*state_or_vector*, *text*, ***kwargs*)

Add a text or LaTeX annotation to Bloch sphere, parametrized by a qubit state or a vector.

> **Parameters state_or_vector** : Qobj/array/list/tuple
>
> > Position for the annotaion. Qobj of a qubit or a vector of 3 elements.
>
> **text** : str/unicode
>
> > Annotation text. You can use LaTeX, but remember to use raw string e.g. r"$\langle x \rangle$" or escape backslashes e.g. "$\langle x \rangle$".
>
> ****kwargs** :
>
> > Options as for mplot3d.axes3d.text, including: fontsize, color, horizontalalignment, verticalalignment.

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', 'l'}
>
> > Type of points to plot, use 'm' for multicolored, 'l' for points connected with a line.

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_like

>>> Array with vectors of unit length or smaller.

clear()
> Resets Bloch sphere data sets to empty.

make_sphere()
> Plots Bloch sphere and data sets.

render(*fig=None*, *axes=None*)
> Render the Bloch sphere and its data sets in on given figure and axes.

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.

set_label_convention(*convention*)
> Set x, y and z labels according to one of conventions.

>> **Parameters** **convention** : string

>>> One of the following:

>>> - "original"
>>> - "xyz"
>>> - "sx sy sz"
>>> - "01"
>>> - "polarization jones"
>>> - "polarization jones letters" see also: http://en.wikipedia.org/wiki/Jones_calculus
>>> - "polarization stokes" see also: http://en.wikipedia.org/wiki/Stokes_parameters

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

4.1.4 Cubic Spline

class **Cubic_Spline**(*a*, *b*, *y*, *alpha=0*, *beta=0*)
> Calculates coefficients for a cubic spline interpolation of a given data set.

> This function assumes that the data is sampled uniformly over a given interval.

>> **Parameters a** : float

>>> Lower bound of the interval.

>> **b** : float

>>> Upper bound of the interval.

>> **y** : ndarray

>>> Function values at interval points.

>> **alpha** : float

>>> Second-order derivative at a. Default is 0.

>> **beta** : float

>>> Second-order derivative at b. Default is 0.

Notes

This object can be called like a normal function with a single or array of input points at which to evaluate the interplating function.

Habermann & Kindermann, "Multidimensional Spline Interpolation: Theory and Applications", Comput Econ 30, 153 (2007).

Attributes

a	(float) Lower bound of the interval.
b	(float) Upper bound of the interval.
coeffs	(ndarray) Array of coeffcients defining cubic spline.

4.1.5 Non-Markovian Solvers

class **HEOMSolver**
> This is superclass for all solvers that use the HEOM method for calculating the dynamics evolution. There are many references for this. A good introduction, and perhaps closest to the notation used here is: DOI:10.1103/PhysRevLett.104.250401 A more canonical reference, with full derivation is: DOI: 10.1103/PhysRevA.41.6676 The method can compute open system dynamics without using any Markovian or rotating wave approximation (RWA) for systems where the bath correlations can be approximated to a sum of complex eponentials. The method builds a matrix of linked differential equations, which are then solved used the same ODE solvers as other qutip solvers (e.g. mesolve)

> This class should be treated as abstract. Currently the only subclass implemented is that for the Drude-Lorentz spectral density. This covers the majority of the work that has been done using this model, and there are some performance advantages to assuming this model where it is appropriate.

> There are opportunities to develop a more general spectral density code.

Attributes

H_sys	(Qobj) System Hamiltonian
coup_op	(Qobj) Operator describing the coupling between system and bath.
coup_strength	(float) Coupling strength.
temperature	(float) Bath temperature, in units corresponding to planck
N_cut	(int) Cutoff parameter for the bath
N_exp	(int) Number of exponential terms used to approximate the bath correlation functions
planck	(float) reduced Planck constant
boltzmann	(float) Boltzmann's constant
options	(`qutip.solver.Options`) Generic solver options. If set to None the default options will be used
progress_bar: BaseProgress-Bar	Optional instance of BaseProgressBar, or a subclass thereof, for showing the progress of the simulation.
stats	(`qutip.solver.Stats`) optional container for holding performance statitics If None is set, then statistics are not collected There may be an overhead in collecting statistics
exp_coeff	(list of complex) Coefficients for the exponential series terms
exp_freq	(list of complex) Frequencies for the exponential series terms

configure (*H_sys*, *coup_op*, *coup_strength*, *temperature*, *N_cut*, *N_exp*, *planck=None*, *boltzmann=None*, *renorm=None*, *bnd_cut_approx=None*, *options=None*, *progress_bar=None*, *stats=None*)

Configure the solver using the passed parameters The parameters are described in the class attributes, unless there is some specific behaviour

> **Parameters options** : `qutip.solver.Options`
>
>> Generic solver options. If set to None the default options will be used
>
> **progress_bar: BaseProgressBar**
>
>> Optional instance of BaseProgressBar, or a subclass thereof, for showing the progress of the simulation. If set to None, then the default progress bar will be used Set to False for no progress bar
>
> **stats: :class:'qutip.solver.Stats'**
>
>> Optional instance of solver.Stats, or a subclass thereof, for storing performance statistics for the solver If set to True, then the default Stats for this class will be used Set to False for no stats

create_new_stats ()

Creates a new stats object suitable for use with this solver Note: this solver expects the stats object to have sections

> config integrate

reset ()

Reset any attributes to default values

class HSolverDL (*H_sys*, *coup_op*, *coup_strength*, *temperature*, *N_cut*, *N_exp*, *cut_freq*, *planck=1.0*, *boltzmann=1.0*, *renorm=True*, *bnd_cut_approx=True*, *options=None*, *progress_bar=None*, *stats=None*)

HEOM solver based on the Drude-Lorentz model for spectral density. Drude-Lorentz bath the correlation functions can be exactly analytically expressed as an infinite sum of exponentials which depend on the temperature, these are called the Matsubara terms or Matsubara frequencies

For practical computation purposes an approximation must be used based on a small number of Matsubara terms (typically < 4).

Attributes

cut_freq	(float) Bath spectral density cutoff frequency.
renorm	(bool) Apply renormalisation to coupling terms Can be useful if using SI units for planck and boltzmann
bnd_cut_approx	(bool) Use boundary cut off approximation Can be

configure(*H_sys*, *coup_op*, *coup_strength*, *temperature*, *N_cut*, *N_exp*, *cut_freq*, *planck=None*, *boltzmann=None*, *renorm=None*, *bnd_cut_approx=None*, *options=None*, *progress_bar=None*, *stats=None*)

Calls configure from `HEOMSolver` and sets any attributes that are specific to this subclass

reset()

Reset any attributes to default values

run(*rho0*, *tlist*)

Function to solve for an open quantum system using the HEOM model.

> **Parameters rho0** : Qobj
>
> > Initial state (density matrix) of the system.
> >
> > **tlist** : list
> >
> > Time over which system evolves.
>
> **Returns results** : [`qutip.solver.Result`]
>
> > Object storing all results from the simulation.

class **MemoryCascade**(*H_S*, *L1*, *L2*, *S_matrix=None*, *c_ops_markov=None*, *integrator='propagator'*, *parallel=False*, *options=None*)

Class for running memory cascade simulations of open quantum systems with time-delayed coherent feedback.

Attributes

H_S	([`qutip.Qobj`]) System Hamiltonian (can also be a Liouvillian)
L1	([`qutip.Qobj`] / list of [`qutip.Qobj`]) System operators coupling into the feedback loop. Can be a single operator or a list of operators.
L2	([`qutip.Qobj`] / list of [`qutip.Qobj`]) System operators coupling out of the feedback loop. Can be a single operator or a list of operators. L2 must have the same length as L1.
S_matrix: *array*	S matrix describing which operators in L1 are coupled to which operators in L2 by the feedback channel. Defaults to an n by n identity matrix where n is the number of elements in L1/L2.
c_ops_markov	([`qutip.Qobj`] / list of [`qutip.Qobj`]) Decay operators describing conventional Markovian decay channels. Can be a single operator or a list of operators.
integra-tor	(str {'propagator', 'mesolve'}) Integrator method to use. Defaults to 'propagator' which tends to be faster for long times (i.e., large Hilbert space).
parallel	(bool) Run integrator in parallel if True. Only implemented for 'propagator' as the integrator method.
options	([`qutip.solver.Options`]) Generic solver options.

outfieldcorr(*rho0*, *blist*, *tlist*, *tau*, *c1=None*, *c2=None*)

Compute output field expectation value <O_n(tn)...O_2(t2)O_1(t1)> for times t1,t2,... and O_i = I, b_out, b_out^dagger, b_loop, b_loop^dagger

> **Parameters rho0** : [`qutip.Qobj`]
>
> > initial density matrix or state vector (ket).
> >
> > **blist** : array_like

List of integers specifying the field operators: 0: I (nothing) 1: b_out 2: b_out^dagger 3: b_loop 4: b_loop^dagger

tlist : array_like

list of corresponding times t1,..,tn at which to evaluate the field operators

tau : float

time-delay

c1 : *qutip.Qobj*

system collapse operator that couples to the in-loop field in question (only needs to be specified if self.L1 has more than one element)

c2 : *qutip.Qobj*

system collapse operator that couples to the output field in question (only needs to be specified if self.L2 has more than one element)

Returns : complex

expectation value of field correlation function

outfieldpropagator(*blist*, *tlist*, *tau*, *c1=None*, *c2=None*, *notrace=False*)
Compute propagator for computing output field expectation values <O_n(tn)...O_2(t2)O_1(t1)> for times t1,t2,... and O_i = I, b_out, b_out^dagger, b_loop, b_loop^dagger

Parameters blist : array_like

List of integers specifying the field operators: 0: I (nothing) 1: b_out 2: b_out^dagger 3: b_loop 4: b_loop^dagger

tlist : array_like

list of corresponding times t1,..,tn at which to evaluate the field operators

tau : float

time-delay

c1 : *qutip.Qobj*

system collapse operator that couples to the in-loop field in question (only needs to be specified if self.L1 has more than one element)

c2 : *qutip.Qobj*

system collapse operator that couples to the output field in question (only needs to be specified if self.L2 has more than one element)

notrace : bool {False}

If this optional is set to True, a propagator is returned for a cascade of k systems, where $(k-1)tau < t < ktau$. If set to False (default), a generalized partial trace is performed and a propagator for a single system is returned.

Returns : *qutip.Qobj*

time-propagator for computing field correlation function

propagator(*t*, *tau*, *notrace=False*)
Compute propagator for time t and time-delay tau

Parameters t : *float*

current time

tau : *float*

time-delay

notrace : *bool* {False}

If this optional is set to True, a propagator is returned for a cascade of k systems, where $(k-1)tau < t < ktau$. If set to False (default), a generalized partial trace is performed and a propagator for a single system is returned.

Returns

——-

: :class:'qutip.Qobj'

time-propagator for reduced system dynamics

rhot (*rho0*, *t*, *tau*)
Compute the reduced system density matrix $\rho(t)$

Parameters rho0 : *qutip.Qobj*

initial density matrix or state vector (ket)

t : float

current time

tau : float

time-delay

Returns : *qutip.Qobj*

density matrix at time t

class **TTMSolverOptions** (*dynmaps=None*, *times=[]*, *learningtimes=[]*, *thres=0.0*, *options=None*)
Class of options for the Transfer Tensor Method solver.

Attributes

dyn-maps	(list of *qutip.Qobj*) List of precomputed dynamical maps (superoperators), or a callback function that returns the superoperator at a given time.
times	(array_like) List of times t_n at which to calculate $\rho(t_n)$
learn-ing-times	(array_like) List of times t_k to use as learning times if argument *dynmaps* is a callback function.
thres	(float) Threshold for halting. Halts if $||T_n - T_{n-1}||$ is below treshold.
options	(*qutip.solver.Options*) Generic solver options.

4.1.6 Solver Options and Results

class **Options** (*atol=1e-08*, *rtol=1e-06*, *method='adams'*, *order=12*, *nsteps=1000*, *first_step=0*, *max_step=0*, *min_step=0*, *average_expect=True*, *average_states=False*, *tidy=True*, *num_cpus=0*, *norm_tol=0.001*, *norm_steps=5*, *rhs_reuse=False*, *rhs_filename=None*, *ntraj=500*, *gui=False*, *rhs_with_state=False*, *store_final_state=False*, *store_states=False*, *seeds=None*, *steady_state_average=False*, *normalize_output=True*, *use_openmp=None*, *openmp_threads=None*)
Class of options for evolution solvers such as *qutip.mesolve* and *qutip.mcsolve*. Options can be specified either as arguments to the constructor:

```
opts = Options(order=10, ...)
```

or by changing the class attributes after creation:

```
opts = Options()
opts.order = 10
```

Returns options class to be used as options in evolution solvers.

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.
aver- age_states	(bool {False}) Average states values over trajectories in stochastic solvers.
aver- age_expect	(bool {True}) Average expectation values over trajectories for stochastic solvers.
mc_corr_eps	(float {1e-10}) Arbitrarily small value for eliminating any divide-by-zero errors in correlation calculations when using mcsolve.
ntraj	(int {500}) Number of trajectories in stochastic solvers.
openmp_threads	(int) Number of OPENMP threads to use. Default is number of cpu cores.
rhs_reuse	(bool {False,True}) Reuse Hamiltonian data.
rhs_with_state	(bool {False,True}) Whether or not to include the state in the Hamiltonian function callback signature.
rhs_filename	(str) Name for compiled Cython file.
seeds	(ndarray) Array containing random number seeds for mcsolver.
store_final_state	(bool {False, True}) Whether or not to store the final state of the evolution in the result class.
store_states	(bool {False, True}) Whether or not to store the state vectors or density matrices in the result class, even if expectation values operators are given. If no expectation are provided, then states are stored by default and this option has no effect.
use_openmp	(bool {True, False}) Use OPENMP for sparse matrix vector multiplication. Default None means auto check.

class **Result**

Class for storing simulation results from any of the dynamics solvers.

Attributes

solver	(str) Which solver was used [e.g., 'mesolve', 'mcsolve', 'brmesolve', ...]
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 trajectories (for stochastic solvers). A 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. Only for Monte Carlo solver.

class **Stats**(*section_names=None*)

Statistical information on the solver performance Statistics can be grouped into sections. If no section names are given in the the contructor, then all statistics will be added to one section 'main'

    Parameters section_names : list

list of keys that will be used as keys for the sections These keys will also be used as names for the sections The text in the output can be overidden by setting the header property of the section If no names are given then one section called 'main' is created

Attributes

sec-tions	(OrderedDict of _StatsSection) These are the sections that are created automatically on instantiation or added using add_section
header	(string) Some text that will be used as the heading in the report By default there is None
to-tal_time	(float) Time in seconds for the solver to complete processing Can be None, meaning that total timing percentages will be reported

Methods

add_section
add_count
add_timing
add_message

report:	Output the statistics report to console or file.

add_count (*key*, *value*, *section=None*)

Add value to count. If key does not already exist in section then it is created with this value. If key already exists it is increased by the give value value is expected to be an integer

Parameters key : string

key for the section.counts dictionary reusing a key will result in numerical addition of value

value : int

Initial value of the count, or added to an existing count

section: string or 'class' : _StatsSection

Section which to add the count to. If None given, the default (first) section will be used

add_message (*key*, *value*, *section=None*, *sep=';'*)

Add value to message. If key does not already exist in section then it is created with this value. If key already exists the value is added to the message The value will be converted to a string

Parameters key : string

key for the section.messages dictionary reusing a key will result in concatenation of value

value : int

Initial value of the message, or added to an existing message

sep : string

Message will be prefixed with this string when concatenating

section: string or 'class' : _StatsSection

Section which to add the message to. If None given, the default (first) section will be used

add_section(*name*)

Add another section with the given name

> **Parameters name** : string
>
> > will be used as key for sections dict will also be the header for the section
>
> **Returns section** : *class*
>
> > The new section

add_timing(*key*, *value*, *section=None*)

Add value to timing. If key does not already exist in section then it is created with this value. If key already exists it is increased by the give value value is expected to be a float, and given in seconds.

> **Parameters key** : string
>
> > key for the section.timings dictionary reusing a key will result in numerical addition of value
>
> **value** : int
>
> > Initial value of the timing, or added to an existing timing
>
> **section: string or 'class'** : _StatsSection
>
> > Section which to add the timing to. If None given, the default (first) section will be used

clear()

Clear counts, timings and messages from all sections

report(*output=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Report the counts, timings and messages from the sections. Sections are reported in the order that the names were supplied in the constructor. The counts, timings and messages are reported in the order that they are added to the sections The output can be written to anything that supports a write method, e.g. a file or the console (default) The output is intended to in markdown format

> **Parameters output** : stream
>
> > file or console stream - anything that support write - where the output will be written

set_total_time(*value*, *section=None*)

Sets the total time for the complete solve or for a specific section value is expected to be a float, and given in seconds

> **Parameters value** : float
>
> > Time in seconds to complete the solver section
>
> **section** : string or *class*
>
> > Section which to set the total_time for If None given, the total_time for complete solve is set

class StochasticSolverOptions(*H=None*, *state0=None*, *times=None*, *c_ops=[]*, *sc_ops=[]*, *e_ops=[]*, *m_ops=None*, *args=None*, *ntraj=1*, *nsubsteps=1*, *d1=None*, *d2=None*, *d2_len=1*, *dW_factors=None*, *rhs=None*, *generate_A_ops=None*, *generate_noise=None*, *homogeneous=True*, *solver=None*, *method=None*, *distribution='normal'*, *store_measurement=False*, *noise=None*, *normalize=True*, *options=None*, *progress_bar=None*, *map_func=None*, *map_kwargs=None*)

Class of options for stochastic solvers such as *qutip.stochastic.ssesolve*, *qutip.stochastic.smesolve*, etc. Options can be specified either as arguments to the constructor:

```
sso = StochasticSolverOptions(nsubsteps=100, ...)
```

or by changing the class attributes after creation:

```
sso = StochasticSolverOptions()
sso.nsubsteps = 1000
```

The stochastic solvers *qutip.stochastic.ssesolve*, *qutip.stochastic.smesolve*, *qutip.stochastic.ssepdpsolve* and *qutip.stochastic.smepdpsolve* all take the same keyword arguments as the constructor of these class, and internally they use these arguments to construct an instance of this class, so it is rarely needed to explicitly create an instance of this class.

Attributes

H	(`qutip.Qobj`) System Hamiltonian.
state0	(`qutip.Qobj`) Initial state vector (ket) or density matrix.
times	(*list / array*) List of times for t. Must be uniformly spaced.
c_ops	(list of `qutip.Qobj`) List of deterministic collapse operators.
sc_ops	(list of `qutip.Qobj`) List of stochastic collapse operators. Each stochastic collapse operator will give a deterministic and stochastic contribution to the equation of motion according to how the d1 and d2 functions are defined.
e_ops	(list of `qutip.Qobj`) Single operator or list of operators for which to evaluate expectation values.
m_ops	(list of `qutip.Qobj`) List of operators representing the measurement operators. The expected format is a nested list with one measurement operator for each stochastic increament, for each stochastic collapse operator.
args	(dict / list) List of dictionary of additional problem-specific parameters. Implicit methods can adjust tolerance via args = {'tol':value}
ntraj	(int) Number of trajectors.
nsubsteps	(int) Number of sub steps between each time-spep given in *times*.
d1	(function) Function for calculating the operator-valued coefficient to the deterministic increment dt.
d2	(function) Function for calculating the operator-valued coefficient to the stochastic increment(s) dW_n, where n is in [0, d2_len[.
d2_len	(int (default 1)) The number of stochastic increments in the process.
dW_factors	(array) Array of length d2_len, containing scaling factors for each measurement operator in m_ops.
rhs	(function) Function for calculating the deterministic and stochastic contributions to the right-hand side of the stochastic differential equation. This only needs to be specified when implementing a custom SDE solver.
generate_A_ops	(function) Function that generates a list of pre-computed operators or super- operators. These precomputed operators are used in some d1 and d2 functions.
generate_noise	(function) Function for generate an array of pre-computed noise signal.
homogeneous	(bool (True)) Wheter or not the stochastic process is homogenous. Inhomogenous processes are only supported for poisson distributions.
solver	(string) Name of the solver method to use for solving the stochastic equations. Valid values are: 1/2 order algorithms: 'euler-maruyama', 'fast-euler-maruyama', 'pc-euler' is a predictor-corrector method which is more stable than explicit methods, 1 order algorithms: 'milstein', 'fast-milstein', 'platen', 'milstein-imp' is semi-implicit Milstein method, 3/2 order algorithms: 'taylor15', 'taylor15-imp' is semi-implicit Taylor 1.5 method. Implicit methods can adjust tolerance via args = {'tol':value}, default is {'tol':1e-6}
method	(string ('homodyne', 'heterodyne', 'photocurrent')) The name of the type of measurement process that give rise to the stochastic equation to solve. Specifying a method with this keyword argument is a short-hand notation for using pre-defined d1 and d2 functions for the corresponding stochastic processes.
distribution	(string ('normal', 'poission')) The name of the distribution used for the stochastic increments.
store_measurement	(bool (default False)) Whether or not to store the measurement results in the `qutip.solver.SolverResult` instance returned by the solver.
noise	(array) Vector specifying the noise.
normalize	(bool (default True)) Whether or not to normalize the wave function during the evolution.
options	(`qutip.solver.Options`) Generic solver options.
map_func: function	A map function or managing the calls to single-trajactory solvers.
map_kwargs: dictionary	Optional keyword arguments to the map_func function function.
progress_bar	(`qutip.ui.BaseProgressBar`) Optional progress bar class instance.

4.1.7 Distribution functions

class Distribution (*data=None*, *xvecs=[]*, *xlabels=[]*)

A class for representation spatial distribution functions.

The Distribution class can be used to prepresent spatial distribution functions of arbitray dimension (although only 1D and 2D distributions are used so far).

It is indented as a base class for specific distribution function, and provide implementation of basic functions that are shared among all Distribution functions, such as visualization, calculating marginal distributions, etc.

Parameters data : array_like

Data for the distribution. The dimensions must match the lengths of the coordinate arrays in xvecs.

xvecs : list

List of arrays that spans the space for each coordinate.

xlabels : list

List of labels for each coordinate.

marginal (*dim=0*)

Calculate the marginal distribution function along the dimension *dim*. Return a new Distribution instance describing this reduced- dimensionality distribution.

Parameters dim : int

The dimension (coordinate index) along which to obtain the marginal distribution.

Returns d : Distributions

A new instances of Distribution that describes the marginal distribution.

project (*dim=0*)

Calculate the projection (max value) distribution function along the dimension *dim*. Return a new Distribution instance describing this reduced-dimensionality distribution.

Parameters dim : int

The dimension (coordinate index) along which to obtain the projected distribution.

Returns d : Distributions

A new instances of Distribution that describes the projection.

visualize (*fig=None*, *ax=None*, *figsize=(8*, *6)*, *colorbar=True*, *cmap=None*, *style='colormap'*, *show_xlabel=True*, *show_ylabel=True*)

Visualize the data of the distribution in 1D or 2D, depending on the dimensionality of the underlaying distribution.

Parameters:

fig [matplotlib Figure instance] If given, use this figure instance for the visualization,

ax [matplotlib Axes instance] If given, render the visualization using this axis instance.

figsize [tuple] Size of the new Figure instance, if one needs to be created.

colorbar: Bool Whether or not the colorbar (in 2D visualization) should be used.

cmap: matplotlib colormap instance If given, use this colormap for 2D visualizations.

style [string] Type of visualization: 'colormap' (default) or 'surface'.

Returns fig, ax : tuple

A tuple of matplotlib figure and axes instances.

class WignerDistribution(*rho=None, extent=[[-5, 5], [-5, 5]], steps=250*)

class QDistribution(*rho=None, extent=[[-5, 5], [-5, 5]], steps=250*)

class TwoModeQuadratureCorrelation(*state=None, theta1=0.0, theta2=0.0, extent=[[-5, 5], [-5, 5]], steps=250*)

> **update**(*state*)
> > calculate probability distribution for quadrature measurement outcomes given a two-mode wavefunction or density matrix

> **update_psi**(*psi*)
> > calculate probability distribution for quadrature measurement outcomes given a two-mode wavefunction

> **update_rho**(*rho*)
> > calculate probability distribution for quadrature measurement outcomes given a two-mode density matrix

class HarmonicOscillatorWaveFunction(*psi=None, omega=1.0, extent=[-5, 5], steps=250*)

> **update**(*psi*)
> > Calculate the wavefunction for the given state of an harmonic oscillator

class HarmonicOscillatorProbabilityFunction(*rho=None, omega=1.0, extent=[-5, 5], steps=250*)

> **update**(*rho*)
> > Calculate the probability function for the given state of an harmonic oscillator (as density matrix)

4.1.8 Quantum information processing

class Gate(*name, targets=None, controls=None, arg_value=None, arg_label=None*)
> Representation of a quantum gate, with its required parametrs, and target and control qubits.

class QubitCircuit(*N, input_states=None, output_states=None, reverse_states=True*)
> Representation of a quantum program/algorithm, maintaining a sequence of gates.

> **add_1q_gate**(*name, start=0, end=None, qubits=None, arg_value=None, arg_label=None*)
> > Adds a single qubit gate with specified parameters on a variable number of qubits in the circuit. By default, it applies the given gate to all the qubits in the register.

> > **Parameters** **name** : String
> > > Gate name.

> > > **start** : Integer
> > > > Starting location of qubits.

> > > **end** : Integer
> > > > Last qubit for the gate.

> > > **qubits** : List
> > > > Specific qubits for applying gates.

> > > **arg_value** : Float
> > > > Argument value(phi).

> > > **arg_label** : String
> > > > Label for gate representation.

> **add_circuit**(*qc, start=0*)
> > Adds a block of a qubit circuit to the main circuit. Globalphase gates are not added.

>**Parameters qc** : QubitCircuit

>>The circuit block to be added to the main circuit.

>>**start** : Integer

>>The qubit on which the first gate is applied.

add_gate(*gate*, *targets=None*, *controls=None*, *arg_value=None*, *arg_label=None*)
Adds a gate with specified parameters to the circuit.

>**Parameters gate: String or 'Gate'**

>>Gate name. If gate is an instance of *Gate*, parameters are unpacked and added.

>>**targets: List**

>>Gate targets.

>>**controls: List**

>>Gate controls.

>>**arg_value: Float**

>>Argument value(phi).

>>**arg_label: String**

>>Label for gate representation.

add_state(*state*, *targets=None*, *state_type='input'*)
Add an input or ouput state to the circuit. By default all the input and output states will be initialized to *None*. A particular state can be added by specifying the state and the qubit where it has to be added along with the type as input or output.

>**Parameters state: str**

>>The state that has to be added. It can be any string such as *0*, '+', "A", "Y"

>>**targets: list**

>>A list of qubit positions where the given state has to be added.

>>**state_type: str**

>>One of either "input" or "output". This specifies whether the state to be added is an input or output. default: "input"

adjacent_gates()
Method to resolve two qubit gates with non-adjacent control/s or target/s in terms of gates with adjacent interactions.

>**Returns qc** : QubitCircuit

>>Returns QubitCircuit of the gates for the qubit circuit with the resolved non-adjacent gates.

propagators()
Propagator matrix calculator for N qubits returning the individual steps as unitary matrices operating from left to right.

>**Returns U_list** : list

>>Returns list of unitary matrices for the qubit circuit.

remove_gate(*index=None*, *end=None*, *name=None*, *remove='first'*)
Removes a gate from a specific index or between two indexes or the first, last or all instances of a particular gate.

>**Parameters index** : Integer

>>Location of gate to be removed.

> **name** : String
>
> > Gate name to be removed.
>
> **remove** : String
>
> > If first or all gate are to be removed.

resolve_gates(*basis=['CNOT', 'RX', 'RY', 'RZ']*)

> Unitary matrix calculator for N qubits returning the individual steps as unitary matrices operating from left to right in the specified basis.
>
> > **Parameters basis** : list.
> >
> > > Basis of the resolved circuit.
> >
> > **Returns qc** : QubitCircuit
> >
> > > Returns QubitCircuit of resolved gates for the qubit circuit in the desired basis.

reverse_circuit()

> Reverses an entire circuit of unitary gates.
>
> > **Returns qc** : QubitCircuit
> >
> > > Returns QubitCircuit of resolved gates for the qubit circuit in the reverse order.

class CircuitProcessor(*N*, *correct_global_phase*)

> Base class for representation of the physical implementation of a quantum program/algorithm on a specified qubit system.

adjacent_gates(*qc*, *setup*)

> Function to take a quantum circuit/algorithm and convert it into the optimal form/basis for the desired physical system.
>
> > **Parameters qc: QubitCircuit**
> >
> > > Takes the quantum circuit to be implemented.
> >
> > > **setup: String**
> >
> > > Takes the nature of the spin chain; linear or circular.
> >
> > **Returns** qc: QubitCircuit
> >
> > > The resolved circuit representation.

get_ops_and_u()

> Returns the Hamiltonian operators and corresponding values by stacking them together.

get_ops_labels()

> Returns the Hamiltonian operators and corresponding labels by stacking them together.

load_circuit(*qc*)

> Translates an abstract quantum circuit to its corresponding Hamiltonian for a specific model.
>
> > **Parameters qc: QubitCircuit**
> >
> > > Takes the quantum circuit to be implemented.

optimize_circuit(*qc*)

> Function to take a quantum circuit/algorithm and convert it into the optimal form/basis for the desired physical system.
>
> > **Parameters qc: QubitCircuit**
> >
> > > Takes the quantum circuit to be implemented.
> >
> > **Returns** qc: QubitCircuit
> >
> > > The optimal circuit representation.

plot_pulses()

Maps the physical interaction between the circuit components for the desired physical system.

> **Returns** fig, ax: Figure
>
> Maps the physical interaction between the circuit components.

pulse_matrix()

Generates the pulse matrix for the desired physical system.

> **Returns** t, u, labels:
>
> Returns the total time and label for every operation.

run(*qc=None*)

Generates the propagator matrix by running the Hamiltonian for the appropriate time duration for the desired physical system.

> **Parameters** **qc: QubitCircuit**
>
> Takes the quantum circuit to be implemented.
>
> **Returns** U_list: list
>
> The propagator matrix obtained from the physical implementation.

run_state(*qc=None*, *states=None*)

Generates the propagator matrix by running the Hamiltonian for the appropriate time duration for the desired physical system with the given initial state of the qubit register.

> **Parameters** **qc: QubitCircuit**
>
> Takes the quantum circuit to be implemented.
>
> **states: Qobj**
>
> Initial state of the qubits in the register.
>
> **Returns** U_list: list
>
> The propagator matrix obtained from the physical implementation.

class **SpinChain**(*N*, *correct_global_phase=True*, *sx=None*, *sz=None*, *sxsy=None*)

Representation of the physical implementation of a quantum program/algorithm on a spin chain qubit system.

adjacent_gates(*qc*, *setup='linear'*)

Method to resolve 2 qubit gates with non-adjacent control/s or target/s in terms of gates with adjacent interactions for linear/circular spin chain system.

> **Parameters** **qc: QubitCircuit**
>
> The circular spin chain circuit to be resolved
>
> **setup: Boolean**
>
> Linear of Circular spin chain setup
>
> **Returns** qc: QubitCircuit
>
> Returns QubitCircuit of resolved gates for the qubit circuit in the desired basis.

class **LinearSpinChain**(*N*, *correct_global_phase=True*, *sx=None*, *sz=None*, *sxsy=None*)

Representation of the physical implementation of a quantum program/algorithm on a spin chain qubit system arranged in a linear formation. It is a sub-class of SpinChain.

class **CircularSpinChain**(*N*, *correct_global_phase=True*, *sx=None*, *sz=None*, *sxsy=None*)

Representation of the physical implementation of a quantum program/algorithm on a spin chain qubit system arranged in a circular formation. It is a sub-class of SpinChain.

class DispersivecQED (*N*, *correct_global_phase=True*, *Nres=None*, *deltamax=None*, *epsmax=None*, *w0=None*, *wq=None*, *eps=None*, *delta=None*, *g=None*)

Representation of the physical implementation of a quantum program/algorithm on a dispersive cavity-QED system.

dispersive_gate_correction (*qc1*, *rwa=True*)

Method to resolve ISWAP and SQRTISWAP gates in a cQED system by adding single qubit gates to get the correct output matrix.

> **Parameters qc: Qobj**
>
> > The circular spin chain circuit to be resolved
>
> **rwa: Boolean**
>
> > Specify if RWA is used or not.
>
> **Returns** qc: QubitCircuit
>
> > Returns QubitCircuit of resolved gates for the qubit circuit in the desired basis.

4.1.9 Optimal control

class Optimizer (*config*, *dyn*, *params=None*)

Base class for all control pulse optimisers. This class should not be instantiated, use its subclasses This class implements the fidelity, gradient and interation callback functions. All subclass objects must be initialised with a

> OptimConfig instance - various configuration options Dynamics instance - describes the dynamics of the (quantum) system
>
> > to be control optimised

Attributes

dumping

log_level	(integer) level of messaging output from the logger. Options are attributes of qutip.logging_utils, in decreasing levels of messaging, are: DEBUG_INTENSE, DEBUG_VERBOSE, DEBUG, INFO, WARN, ERROR, CRITICAL Anything WARN or above is effectively 'quiet' execution, assuming everything runs as expected. The default NOTSET implies that the level will be taken from the QuTiP settings file, which by default is WARN
params: Dictionary	The key value pairs are the attribute name and value Note: attributes are created if they do not exist already, and are overwritten if they do.
alg	(string) Algorithm to use in pulse optimisation. Options are: 'GRAPE' (default) - GRadient Ascent Pulse Engineering 'CRAB' - Chopped RAndom Basis
alg_params	(Dictionary) options that are specific to the pulse optim algorithm that is GRAPE or CRAB
disp_conv_msg	(bool) Set true to display a convergence message (for scipy.optimize.minimize methods anyway)
op-tim_method	(string) a scipy.optimize.minimize method that will be used to optimise the pulse for minimum fidelity error
method_params	(Dictionary) Options for the optim_method. Note that where there is an equivalent attribute of this instance or the termination_conditions (for example maxiter) it will override an value in these options
ap-prox_grad	(bool) If set True then the method will approximate the gradient itself (if it has requirement and facility for this) This will mean that the fid_err_grad_wrapper will not get called Note it should be left False when using the Dynamics to calculate approximate gradients Note it is set True automatically when the alg is CRAB
amp_lbound	(float or list of floats) lower boundaries for the control amplitudes Can be a scalar value applied to all controls or a list of bounds for each control
amp_ubound	(float or list of floats) upper boundaries for the control amplitudes Can be a scalar value applied to all controls or a list of bounds for each control
bounds	(List of floats) Bounds for the parameters. If not set before the run_optimization call then the list is built automatically based on the amp_lbound and amp_ubound attributes. Setting this attribute directly allows specific bounds to be set for individual parameters. Note: Only some methods use bounds
dynamics	(Dynamics (subclass instance)) describes the dynamics of the (quantum) system to be control optimised (see Dynamics classes for details)
config	(OptimConfig instance) various configuration options (see OptimConfig for details)
termina-tion_conditions	(TerminationCondition instance) attributes determine when the optimisation will end
pulse_generator	(PulseGen (subclass instance)) (can be) used to create initial pulses not used by the class, but set by pulseoptim.create_pulse_optimizer
stats	(Stats) attributes of which give performance stats for the optimisation set to None to reduce overhead of calculating stats. Note it is (usually) shared with the Dynamics instance
dump	(`dump.OptimDump`) Container for data dumped during the optimisation. Can be set by specifying the dumping level or set directly. Note this is mainly intended for user and a development debugging but could be used for status information during a long optimisation.
dump_to_file	(bool) If set True then data will be dumped to file during the optimisation dumping will be set to SUMMARY during init_optim if dump_to_file is True and dumping not set. Default is False
dump_dir	(string) Basically a link to dump.dump_dir. Exists so that it can be set through optim_params. If dump is None then will return None or will set dumping to SUMMARY when setting a path
iter_summary	(`OptimIterSummary`) Summary of the most recent iteration. Note this is only set if dummping is on

apply_method_params (*params=None*)

> Loops through all the method_params (either passed here or the method_params attribute) If the name matches an attribute of this object or the termination conditions object, then the value of this attribute is set. Otherwise it is assumed to a method_option for the scipy.optimize.minimize function

apply_params (*params=None*)
> Set object attributes based on the dictionary (if any) passed in the instantiation, or passed as a parameter This is called during the instantiation automatically. The key value pairs are the attribute name and value Note: attributes are created if they do not exist already, and are overwritten if they do.

dumping

> **The level of data dumping that will occur during the optimisation**
>
> > - NONE : No processing data dumped (Default)
> >
> > - SUMMARY : A summary at each iteration will be recorded
> >
> > - FULL : All logs will be generated and dumped
> >
> > - CUSTOM : Some customised level of dumping
>
> When first set to CUSTOM this is equivalent to SUMMARY. It is then up to the user to specify which logs are dumped

fid_err_func_wrapper (**args*)
> Get the fidelity error achieved using the ctrl amplitudes passed in as the first argument.
>
> This is called by generic optimisation algorithm as the func to the minimised. The argument is the current variable values, i.e. control amplitudes, passed as a flat array. Hence these are reshaped as [nTimeslots, n_ctrls] and then used to update the stored ctrl values (if they have changed)
>
> The error is checked against the target, and the optimisation is terminated if the target has been achieved.

fid_err_grad_wrapper (**args*)
> Get the gradient of the fidelity error with respect to all of the variables, i.e. the ctrl amplidutes in each timeslot
>
> This is called by generic optimisation algorithm as the gradients of func to the minimised wrt the variables. The argument is the current variable values, i.e. control amplitudes, passed as a flat array. Hence these are reshaped as [nTimeslots, n_ctrls] and then used to update the stored ctrl values (if they have changed)
>
> Although the optimisation algorithms have a check within them for function convergence, i.e. local minima, the sum of the squares of the normalised gradient is checked explicitly, and the optimisation is terminated if this is below the min_gradient_norm condition

init_optim (*term_conds*)
> Check optimiser attribute status and passed parameters before running the optimisation. This is called by run_optimization, but could called independently to check the configuration.

iter_step_callback_func (**args*)
> Check the elapsed wall time for the optimisation run so far. Terminate if this has exceeded the maximum allowed time

run_optimization (*term_conds=None*)
> This default function optimisation method is a wrapper to the scipy.optimize.minimize function.
>
> It will attempt to minimise the fidelity error with respect to some parameters, which are determined by _get_optim_var_vals (see below)
>
> The optimisation end when one of the passed termination conditions has been met, e.g. target achieved, wall time, or function call or iteration count exceeded. Note these conditions include gradient minimum met (local minima) for methods that use a gradient.
>
> The function minimisation method is taken from the optim_method attribute. Note that not all of these methods have been tested. Note that some of these use a gradient and some do not. See the scipy documentation for details. Options specific to the method can be passed setting the method_params attribute.
>
> If the parameter term_conds=None, then the termination_conditions attribute must already be set. It will be overwritten if the parameter is not None

The result is returned in an OptimResult object, which includes the final fidelity, time evolution, reason for termination etc

class `OptimizerBFGS`(*config*, *dyn*, *params=None*)

Implements the run_optimization method using the BFGS algorithm

> `run_optimization`(*term_conds=None*)
>
> Optimise the control pulse amplitudes to minimise the fidelity error using the BFGS (Broyden–Fletcher–Goldfarb–Shanno) algorithm The optimisation end when one of the passed termination conditions has been met, e.g. target achieved, gradient minimum met (local minima), wall time / iteration count exceeded.
>
> Essentially this is wrapper to the: scipy.optimize.fmin_bfgs function
>
> If the parameter term_conds=None, then the termination_conditions attribute must already be set. It will be overwritten if the parameter is not None
>
> The result is returned in an OptimResult object, which includes the final fidelity, time evolution, reason for termination etc

class `OptimizerLBFGSB`(*config*, *dyn*, *params=None*)

Implements the run_optimization method using the L-BFGS-B algorithm

Attributes

max_metric_corr *(integer)*	The maximum number of variable metric corrections used to define the limited memory matrix. That is the number of previous gradient values that are used to approximate the Hessian see the scipy.optimize.fmin_l_bfgs_b documentation for description of m argument

> `init_optim`(*term_conds*)
>
> Check optimiser attribute status and passed parameters before running the optimisation. This is called by run_optimization, but could called independently to check the configuration.
>
> `run_optimization`(*term_conds=None*)
>
> Optimise the control pulse amplitudes to minimise the fidelity error using the L-BFGS-B algorithm, which is the constrained (bounded amplitude values), limited memory, version of the Broyden–Fletcher–Goldfarb–Shanno algorithm.
>
> The optimisation end when one of the passed termination conditions has been met, e.g. target achieved, gradient minimum met (local minima), wall time / iteration count exceeded.
>
> Essentially this is wrapper to the: scipy.optimize.fmin_l_bfgs_b function This in turn is a warpper for well established implementation of the L-BFGS-B algorithm written in Fortran, which is therefore very fast. See SciPy documentation for credit and details on this function.
>
> If the parameter term_conds=None, then the termination_conditions attribute must already be set. It will be overwritten if the parameter is not None
>
> The result is returned in an OptimResult object, which includes the final fidelity, time evolution, reason for termination etc

class `OptimizerCrab`(*config*, *dyn*, *params=None*)

Optimises the pulse using the CRAB algorithm [1]. It uses the scipy.optimize.minimize function with the method specified by the optim_method attribute. See Optimizer.run_optimization for details It minimises the fidelity error function with respect to the CRAB basis function coefficients.

AJGP ToDo: Add citation here

> `init_optim`(*term_conds*)
>
> Check optimiser attribute status and passed parameters before running the optimisation. This is called by run_optimization, but could called independently to check the configuration.

class `OptimizerCrabFmin`(*config*, *dyn*, *params=None*)

Optimises the pulse using the CRAB algorithm [1, 2]. It uses the scipy.optimize.fmin function which is

effectively a wrapper for the Nelder-mead method. It minimises the fidelity error function with respect to the CRAB basis function coefficients. This is the default Optimizer for CRAB.

Notes

[1] P. Doria, T. Calarco & S. Montangero. Phys. Rev. Lett. 106, 190501 (2011).

[2] T. Caneva, T. Calarco, & S. Montangero. Phys. Rev. A 84, 022326 (2011).

run_optimization(*term_conds=None*)
　　This function optimisation method is a wrapper to the scipy.optimize.fmin function.

　　It will attempt to minimise the fidelity error with respect to some parameters, which are determined by _get_optim_var_vals which in the case of CRAB are the basis function coefficients

　　The optimisation end when one of the passed termination conditions has been met, e.g. target achieved, wall time, or function call or iteration count exceeded. Specifically to the fmin method, the optimisation will stop when change parameter values is less than xtol or the change in function value is below ftol.

　　If the parameter term_conds=None, then the termination_conditions attribute must already be set. It will be overwritten if the parameter is not None

　　The result is returned in an OptimResult object, which includes the final fidelity, time evolution, reason for termination etc

class OptimIterSummary
　　A summary of the most recent iteration of the pulse optimisation

Attributes

iter_num	(int) Iteration number of the pulse optimisation
fid_func_call_num	(int) Fidelity function call number of the pulse optimisation
grad_func_call_num	(int) Gradient function call number of the pulse optimisation
fid_err	(float) Fidelity error
grad_norm	(float) fidelity gradient (wrt the control parameters) vector norm that is the magnitude of the gradient
wall_time	(float) Time spent computing the pulse optimisation so far (in seconds of elapsed time)

class TerminationConditions
　　Base class for all termination conditions Used to determine when to stop the optimisation algorithm Note different subclasses should be used to match the type of optimisation being used

Attributes

fid_err_targ	(float) Target fidelity error
fid_goal	(float) goal fidelity, e.g. 1 - self.fid_err_targ It its typical to set this for unitary systems
max_wall_time	(float) # maximum time for optimisation (seconds)
min_gradient	(float) Minimum normalised gradient after which optimisation will terminate
max_iterations	(integer) Maximum iterations of the optimisation algorithm
max_fid_func_calls	(integer) Maximum number of calls to the fidelity function during the optimisation algorithm
accuracy_factor	(float) Determines the accuracy of the result. Typical values for accuracy_factor are: 1e12 for low accuracy; 1e7 for moderate accuracy; 10.0 for extremely high accuracy scipy.optimize.fmin_l_bfgs_b factr argument. Only set for specific methods (fmin_l_bfgs_b) that uses this Otherwise the same thing is passed as method_option ftol (although the scale is different) Hence it is not defined here, but may be set by the user

class `OptimResult`

> Attributes give the result of the pulse optimisation attempt

Attributes

termination_reason	(string) Description of the reason for terminating the optimisation
fidelity	(float) final (normalised) fidelity that was achieved
initial_fid_err	(float) fidelity error before optimisation starting
fid_err	(float) final fidelity error that was achieved
goal_achieved	(boolean) True is the fidely error achieved was below the target
grad_norm_final	(float) Final value of the sum of the squares of the (normalised) fidelity error gradients
grad_norm_min_reached	(float) True if the optimisation terminated due to the minimum value of the gradient being reached
num_iter	(integer) Number of iterations of the optimisation algorithm completed
max_iter_exceeded	(boolean) True if the iteration limit was reached
max_fid_func_exceeded	(boolean) True if the fidelity function call limit was reached
wall_time	(float) time elapsed during the optimisation
wall_time_limit_exceeded	(boolean) True if the wall time limit was reached
time	(array[num_tslots+1] of float) Time are the start of each timeslot with the final value being the total evolution time
initial_amps	(array[num_tslots, n_ctrls]) The amplitudes at the start of the optimisation
final_amps	(array[num_tslots, n_ctrls]) The amplitudes at the end of the optimisation
evo_full_final	(Qobj) The evolution operator from t=0 to t=T based on the final amps
stats	(Stats) Object contaning the stats for the run (if any collected)
optimizer	(Optimizer) Instance of the Optimizer used to generate the result

class `Dynamics` (*optimconfig*, *params=None*)

> This is a base class only. See subclass descriptions and choose an appropriate one for the application.
>
> Note that initialize_controls must be called before most of the methods can be used. init_timeslots can be called sometimes earlier in order to access timeslot related attributes
>
> This acts as a container for the operators that are used to calculate time evolution of the system under study. That is the dynamics generators (Hamiltonians, Lindbladians etc), the propagators from one timeslot to the next, and the evolution operators. Due to the large number of matrix additions and multiplications, for small systems at least, the optimisation performance is much better using ndarrays to represent these operators. However

Attributes

`num_ctrls`
`dyn_gen`
`prop`
`prop_grad`
`fwd_evo`
`onwd_evo`
`onto_evo`
`dumping`

log_level	(integer) level of messaging output from the logger. Options are attributes of qutip.logg
params: Dictionary	The key value pairs are the attribute name and value Note: attributes are created if they
stats	(Stats) Attributes of which give performance stats for the optimisation set to None to re

tslot_computer	(TimeslotComputer (subclass instance)) Used to manage when the timeslot dynamics g
prop_computer	(PropagatorComputer (subclass instance)) Used to compute the propagators and their g
fid_computer	(FidelityComputer (subclass instance)) Used to computer the fidelity error and the fidel
memory_optimization	(int) Level of memory optimisation. Setting to 0 (default) means that execution speed is
oper_dtype	(type) Data type for internal dynamics generators, propagators and time evolution opera
cache_phased_dyn_gen	(bool) If True then the dynamics generators will be saved with and without the propaga
cache_prop_grad	(bool) If the True then the propagator gradients (for exact gradients) will be computed v
cache_dyn_gen_eigenvectors_adj: bool	If True then DynamicsUnitary will cached the adjoint of the Hamiltion eignvector matr
sparse_eigen_decomp: bool	If True then DynamicsUnitary will use the sparse eigenvalue decomposition. Defaults t
num_tslots	(integer) Number of timeslots (aka timeslices)
evo_time	(float) Total time for the evolution
tau	(array[num_tslots] of float) Duration of each timeslot Note that if this is set before initi
time	(array[num_tslots+1] of float) Cumulative time for the evolution, that is the time at the
drift_dyn_gen	(Qobj or list of Qobj) Drift or system dynamics generator (Hamiltonian) Matrix definin
ctrl_dyn_gen	(List of Qobj) Control dynamics generator (Hamiltonians) List of matrices defining the
initial	(Qobj) Starting state / gate The matrix giving the initial state / gate, i.e. at time 0 Typica
target	(Qobj) Target state / gate: The matrix giving the desired state / gate for the evolution
ctrl_amps	(array[num_tslots, num_ctrls] of float) Control amplitudes The amplitude (scale factor)
initial_ctrl_scaling	(float) Scale factor applied to be applied the control amplitudes when they are initialise
initial_ctrl_offset	(float) Linear offset applied to be applied the control amplitudes when they are initialise
evo_current	(Boolean) Used to flag that the dynamics used to calculate the evolution operators is cu
fact_mat_round_prec	(float) Rounding precision used when calculating the factor matrix to determine if two
def_amps_fname	(string) Default name for the output used when save_amps is called
unitarity_check_level	(int) If > 0 then unitarity of the system evolution is checked at at evolution recomputati
unitarity_tol :	Tolerance used in checking if operator is unitary Default is 1e-10
dump	(dump.DynamicsDump) Store of historical calculation data. Set to None (Default) fo
dump_to_file	(bool) If set True then data will be dumped to file during the calculations dumping will
dump_dir	(string) Basically a link to dump.dump_dir. Exists so that it can be set through dyn_par

apply_params(*params=None*)
> Set object attributes based on the dictionary (if any) passed in the instantiation, or passed as a parameter This is called during the instantiation automatically. The key value pairs are the attribute name and value Note: attributes are created if they do not exist already, and are overwritten if they do.

combine_dyn_gen(*k*)
> Computes the dynamics generator for a given timeslot The is the combined Hamiltion for unitary systems

compute_evolution()
> Recalculate the time evolution operators Dynamics generators (e.g. Hamiltonian) and prop (propagators) are calculated as necessary Actual work is completed by the recompute_evolution method of the timeslot computer

dumping
> The level of data dumping that will occur during the time evolution calculation.

> - NONE : No processing data dumped (Default)
> - SUMMARY : A summary of each time evolution will be recorded
> - FULL : All operators used or created in the calculation dumped
> - CUSTOM : Some customised level of dumping

> When first set to CUSTOM this is equivalent to SUMMARY. It is then up to the user to specify which operators are dumped WARNING: FULL could consume a lot of memory!

dyn_gen
> List of combined dynamics generators (Qobj) for each timeslot

dyn_gen_phase
> Some preop that is applied to the dyn_gen before expontiating to get the propagator

flag_system_changed()
> Flag evolution, fidelity and gradients as needing recalculation

full_evo
> Full evolution - time evolution at final time slot

fwd_evo
> List of evolution operators (Qobj) from the initial to the given timeslot

get_ctrl_dyn_gen(*j*)
> Get the dynamics generator for the control Not implemented in the base class. Choose a subclass

get_drift_dim()
> Returns the size of the matrix that defines the drift dynamics that is assuming the drift is NxN, then this returns N

get_dyn_gen(*k*)
> Get the combined dynamics generator for the timeslot Not implemented in the base class. Choose a subclass

get_num_ctrls()
> calculate the of controls from the length of the control list sets the num_ctrls property, which can be used alternatively subsequently

init_timeslots()
> Generate the timeslot duration array 'tau' based on the evo_time and num_tslots attributes, unless the tau attribute is already set in which case this step in ignored Generate the cumulative time array 'time' based on the tau values

initialize_controls(*amps*, *init_tslots=True*)
> Set the initial control amplitudes and time slices Note this must be called after the configuration is complete before any dynamics can be calculated

num_ctrls
> calculate the of controls from the length of the control list sets the num_ctrls property, which can be used alternatively subsequently

onto_evo
> List of evolution operators (Qobj) from the initial to the given timeslot

onwd_evo
> List of evolution operators (Qobj) from the initial to the given timeslot

prop
> List of propagators (Qobj) for each timeslot

prop_grad
> Array of propagator gradients (Qobj) for each timeslot, control

refresh_drift_attribs()
> Reset the dyn_shape, dyn_dims and time_depend_drift attribs

save_amps(*file_name=None*, *times=None*, *amps=None*, *verbose=False*)
> Save a file with the current control amplitudes in each timeslot The first column in the file will be the start time of the slot

>> **Parameters** **file_name** : string

>>> Name of the file If None given the def_amps_fname attribuite will be used

>> **times** : List type (or string)

>>> List / array of the start times for each slot If None given this will be retrieved through get_amp_times() If 'exclude' then times will not be saved in the file, just the amplitudes

> > > **amps** : Array[num_tslots, num_ctrls]
> > >
> > > > Amplitudes to be saved If None given the ctrl_amps attribute will be used
> > >
> > > **verbose** : Boolean
> > >
> > > > If True then an info message will be logged

> > **unitarity_check**()
> > Checks whether all propagators are unitary

> > **update_ctrl_amps**(*new_amps*)
> > Determine if any amplitudes have changed. If so, then mark the timeslots as needing recalculation The actual work is completed by the compare_amps method of the timeslot computer

class DynamicsGenMat(*optimconfig*, *params=None*)
> This sub class can be used for any system where no additional operator is applied to the dynamics generator before calculating the propagator, e.g. classical dynamics, Lindbladian

class DynamicsUnitary(*optimconfig*, *params=None*)
> This is the subclass to use for systems with dynamics described by unitary matrices. E.g. closed systems with Hermitian Hamiltonians Note a matrix diagonalisation is used to compute the exponent The eigen decomposition is also used to calculate the propagator gradient. The method is taken from DYNAMO (see file header)

Attributes

drift_ham	(Qobj) This is the drift Hamiltonian for unitary dynamics It is mapped to drift_dyn_gen during initialize_controls
ctrl_ham	(List of Qobj) These are the control Hamiltonians for unitary dynamics It is mapped to ctrl_dyn_gen during initialize_controls
H	(List of Qobj) The combined drift and control Hamiltonians for each timeslot These are the dynamics generators for unitary dynamics. It is mapped to dyn_gen during initialize_controls

> **check_unitarity**()
> Checks whether all propagators are unitary For propagators found not to be unitary, the potential underlying causes are investigated.

class DynamicsSymplectic(*optimconfig*, *params=None*)
> Symplectic systems This is the subclass to use for systems where the dynamics is described by symplectic matrices, e.g. coupled oscillators, quantum optics

Attributes

omega	(array[drift_dyn_gen.shape]) matrix used in the calculation of propagators (time evolution) with symplectic systems.

> **dyn_gen_phase**
> The prephasing operator for the symplectic group generators usually refered to as Omega

class PropagatorComputer(*dynamics*, *params=None*)
> Base for all Propagator Computer classes that are used to calculate the propagators, and also the propagator gradient when exact gradient methods are used Note: they must be instantiated with a Dynamics object, that is the container for the data that the functions operate on This base class cannot be used directly. See subclass descriptions and choose the appropriate one for the application

Attributes

log_level	(integer) level of messaging output from the logger. Options are attributes of qutip_utils.logging, in decreasing levels of messaging, are: DEBUG_INTENSE, DEBUG_VERBOSE, DEBUG, INFO, WARN, ERROR, CRITICAL Anything WARN or above is effectively 'quiet' execution, assuming everything runs as expected. The default NOTSET implies that the level will be taken from the QuTiP settings file, which by default is WARN
grad_exact	(boolean) indicates whether the computer class instance is capable of computing propagator gradients. It is used to determine whether to create the Dynamics prop_grad array

apply_params (*params=None*)

> Set object attributes based on the dictionary (if any) passed in the instantiation, or passed as a parameter This is called during the instantiation automatically. The key value pairs are the attribute name and value Note: attributes are created if they do not exist already, and are overwritten if they do.

reset ()

> reset any configuration data

class PropCompApproxGrad (*dynamics*, *params=None*)

This subclass can be used when the propagator is calculated simply by expm of the dynamics generator, i.e. when gradients will be calculated using approximate methods.

reset ()

> reset any configuration data

class PropCompDiag (*dynamics*, *params=None*)

Coumputes the propagator exponentiation using diagonalisation of of the dynamics generator

reset ()

> reset any configuration data

class PropCompFrechet (*dynamics*, *params=None*)

Frechet method for calculating the propagator: exponentiating the combined dynamics generator

and the propagator gradient It should work for all systems, e.g. unitary, open, symplectic There are other PropagatorComputer subclasses that may be more efficient

class FidelityComputer (*dynamics*, *params=None*)

Base class for all Fidelity Computers. This cannot be used directly. See subclass descriptions and choose one appropriate for the application Note: this must be instantiated with a Dynamics object, that is the container for the data that the methods operate on

Attributes

log_level	(integer) level of messaging output from the logger. Options are attributes of qutip.logging_utils, in decreasing levels of messaging, are: DEBUG_INTENSE, DEBUG_VERBOSE, DEBUG, INFO, WARN, ERROR, CRITICAL Anything WARN or above is effectively 'quiet' execution, assuming everything runs as expected. The default NOTSET implies that the level will be taken from the QuTiP settings file, which by default is WARN
dimensional_norm	(float) Normalisation constant
fid_norm_func	(function) Used to normalise the fidelity See SU and PSU options for the unitary dynamics
grad_norm_func	(function) Used to normalise the fidelity gradient See SU and PSU options for the unitary dynamics
uses_onwd_evo	(boolean) flag to specify whether the onwd_evo evolution operator (see Dynamics) is used by the FidelityComputer
uses_onto_evo	(boolean) flag to specify whether the onto_evo evolution operator (see Dynamics) is used by the FidelityComputer
fid_err	(float) Last computed value of the fidelity error
fidelity	(float) Last computed value of the normalised fidelity
fidelity_current	(boolean) flag to specify whether the fidelity / fid_err are based on the current amplitude values. Set False when amplitudes change
fid_err_grad: array[num_tslot, num_ctrls] of float	Last computed values for the fidelity error gradients wrt the control in the timeslot
grad_norm	(float) Last computed value for the norm of the fidelity error gradients (sqrt of the sum of the squares)
fid_err_grad_current	(boolean) flag to specify whether the fidelity / fid_err are based on the current amplitude values. Set False when amplitudes change

apply_params(*params=None*)

> Set object attributes based on the dictionary (if any) passed in the instantiation, or passed as a parameter This is called during the instantiation automatically. The key value pairs are the attribute name and value Note: attributes are created if they do not exist already, and are overwritten if they do.

clear()

> clear any temporarily held status data

flag_system_changed()

> Flag fidelity and gradients as needing recalculation

get_fid_err()

> returns the absolute distance from the maximum achievable fidelity

get_fid_err_gradient()

> Returns the normalised gradient of the fidelity error in a (nTimeslots x n_ctrls) array wrt the timeslot control amplitude

init_comp()

> initialises the computer based on the configuration of the Dynamics

reset()

> reset any configuration data and clear any temporarily held status data

class FidCompUnitary(*dynamics*, *params=None*)

> Computes fidelity error and gradient assuming unitary dynamics, e.g. closed qubit systems Note fidelity and gradient calculations were taken from DYNAMO (see file header)

Attributes

phase_option	(string) determines how global phase is treated in fidelity calculations: PSU - global phase ignored SU - global phase included
fidelity_prenorm	(complex) Last computed value of the fidelity before it is normalised It is stored to use in the gradient normalisation calculation
fidelity_prenorm_current	(boolean) flag to specify whether fidelity_prenorm are based on the current amplitude values. Set False when amplitudes change

compute_fid_grad()
> Calculates exact gradient of function wrt to each timeslot control amplitudes. Note these gradients are not normalised These are returned as a (nTimeslots x n_ctrls) array

flag_system_changed()
> Flag fidelity and gradients as needing recalculation

get_fid_err()
> Gets the absolute error in the fidelity

get_fid_err_gradient()
> Returns the normalised gradient of the fidelity error in a (nTimeslots x n_ctrls) array The gradients are cached in case they are requested mutliple times between control updates (although this is not typically found to happen)

get_fidelity()
> Gets the appropriately normalised fidelity value The normalisation is determined by the fid_norm_func pointer which should be set in the config

get_fidelity_prenorm()
> Gets the current fidelity value prior to normalisation Note the gradient function uses this value The value is cached, because it is used in the gradient calculation

init_comp()
> Check configuration and initialise the normalisation

init_normalization()
> Calc norm of <Ufinal | Ufinal> to scale subsequent norms When considering unitary time evolution operators, this basically results in calculating the trace of the identity matrix and is hence equal to the size of the target matrix There may be situations where this is not the case, and hence it is not assumed to be so. The normalisation function called should be set to either the PSU - global phase ignored SU - global phase respected

normalize_PSU(*A*)

normalize_SU(*A*)

normalize_gradient_PSU(*grad*)
> Normalise the gradient matrix passed as grad This PSU version is independent of global phase

normalize_gradient_SU(*grad*)
> Normalise the gradient matrix passed as grad This SU version respects global phase

set_phase_option(*phase_option=None*)
> Deprecated - use phase_option Phase options are SU - global phase important PSU - global phase is not important

class **FidCompTraceDiff**(*dynamics*, *params=None*)
> Computes fidelity error and gradient for general system dynamics by calculating the the fidelity error as the trace of the overlap of the difference between the target and evolution resulting from the pulses with the transpose of the same. This should provide a distance measure for dynamics described by matrices Note the gradient calculation is taken from: 'Robust quantum gates for open systems via optimal control: Markovian versus non-Markovian dynamics' Frederik F Floether, Pierre de Fouquieres, and Sophie G Schirmer

Attributes

scale_factor	(float) The fidelity error calculated is of some arbitary scale. This factor can be used to scale the fidelity error such that it may represent some physical measure If None is given then it is caculated as 1/2N, where N is the dimension of the drift, when the Dynamics are initialised.

compute_fid_err_grad()
> Calculate exact gradient of the fidelity error function wrt to each timeslot control amplitudes. Uses the trace difference norm fidelity These are returned as a (nTimeslots x n_ctrls) array

get_fid_err()
> Gets the absolute error in the fidelity

get_fid_err_gradient()
> Returns the normalised gradient of the fidelity error in a (nTimeslots x n_ctrls) array The gradients are cached in case they are requested mutliple times between control updates (although this is not typically found to happen)

init_comp()
> initialises the computer based on the configuration of the Dynamics Calculates the scale_factor is not already set

class FidCompTraceDiffApprox(*dynamics*, *params=None*)
> As FidCompTraceDiff, except uses the finite difference method to compute approximate gradients

Attributes

ep-silon	(float) control amplitude offset to use when approximating the gradient wrt a timeslot control amplitude

compute_fid_err_grad()
> Calculates gradient of function wrt to each timeslot control amplitudes. Note these gradients are not normalised They are calulated These are returned as a (nTimeslots x n_ctrls) array

class TimeslotComputer(*dynamics*, *params=None*)
> Base class for all Timeslot Computers Note: this must be instantiated with a Dynamics object, that is the container for the data that the methods operate on

Attributes

log_level	(integer) level of messaging output from the logger. Options are attributes of qutip.logging_utils, in decreasing levels of messaging, are: DEBUG_INTENSE, DEBUG_VERBOSE, DEBUG, INFO, WARN, ERROR, CRITICAL Anything WARN or above is effectively 'quiet' execution, assuming everything runs as expected. The default NOTSET implies that the level will be taken from the QuTiP settings file, which by default is WARN
evo_comp_summary	(EvoCompSummary) A summary of the most recent evolution computation Used in the stats and dump Will be set to None if neither stats or dump are set

apply_params(*params=None*)
> Set object attributes based on the dictionary (if any) passed in the instantiation, or passed as a parameter This is called during the instantiation automatically. The key value pairs are the attribute name and value Note: attributes are created if they do not exist already, and are overwritten if they do.

dump_current()
> Store a copy of the current time evolution

class TSlotCompUpdateAll(*dynamics*, *params=None*)
> Timeslot Computer - Update All Updates all dynamics generators, propagators and evolutions when ctrl amplitudes are updated

compare_amps(*new_amps*)

> Determine if any amplitudes have changed. If so, then mark the timeslots as needing recalculation Returns: True if amplitudes are the same, False if they have changed

get_timeslot_for_fidelity_calc()

> Returns the timeslot index that will be used calculate current fidelity value. This (default) method simply returns the last timeslot

recompute_evolution()

> Recalculates the evolution operators. Dynamics generators (e.g. Hamiltonian) and prop (propagators) are calculated as necessary

class PulseGen(*dyn=None*, *params=None*)

> Pulse generator Base class for all Pulse generators The object can optionally be instantiated with a Dynamics object, in which case the timeslots and amplitude scaling and offset are copied from that. Otherwise the class can be used independently by setting: tau (array of timeslot durations) or num_tslots and pulse_time for equally spaced timeslots

Attributes

num_tslots	(integer) Number of timeslots, aka timeslices (copied from Dynamics if given)
pulse_time	(float) total duration of the pulse (copied from Dynamics.evo_time if given)
scaling	(float) linear scaling applied to the pulse (copied from Dynamics.initial_ctrl_scaling if given)
offset	(float) linear offset applied to the pulse (copied from Dynamics.initial_ctrl_offset if given)
tau	(array[num_tslots] of float) Duration of each timeslot (copied from Dynamics if given)
lbound	(float) Lower boundary for the pulse amplitudes Note that the scaling and offset attributes can be used to fully bound the pulse for all generators except some of the random ones This bound (if set) may result in additional shifting / scaling Default is -Inf
ubound	(float) Upper boundary for the pulse amplitudes Note that the scaling and offset attributes can be used to fully bound the pulse for all generators except some of the random ones This bound (if set) may result in additional shifting / scaling Default is Inf
periodic	(boolean) True if the pulse generator produces periodic pulses
random	(boolean) True if the pulse generator produces random pulses
log_level	(integer) level of messaging output from the logger. Options are attributes of qutip.logging_utils, in decreasing levels of messaging, are: DEBUG_INTENSE, DEBUG_VERBOSE, DEBUG, INFO, WARN, ERROR, CRITICAL Anything WARN or above is effectively 'quiet' execution, assuming everything runs as expected. The default NOTSET implies that the level will be taken from the QuTiP settings file, which by default is WARN

apply_params(*params=None*)

> Set object attributes based on the dictionary (if any) passed in the instantiation, or passed as a parameter This is called during the instantiation automatically. The key value pairs are the attribute name and value

gen_pulse()

> returns the pulse as an array of vales for each timeslot Must be implemented by subclass

init_pulse()

> Initialise the pulse parameters

reset()

> reset attributes to default values

class PulseGenRandom(*dyn=None*, *params=None*)

> Generates random pulses as simply random values for each timeslot

gen_pulse()

    Generate a pulse of random values between 1 and -1 Values are scaled using the scaling property and shifted using the offset property Returns the pulse as an array of vales for each timeslot

class **PulseGenZero**(*dyn=None*, *params=None*)

    Generates a flat pulse

gen_pulse()

    Generate a pulse with the same value in every timeslot. The value will be zero, unless the offset is not zero, in which case it will be the offset

class **PulseGenLinear**(*dyn=None*, *params=None*)

    Generates linear pulses

Attributes

gradient	(float) Gradient of the line. Note this is calculated from the start_val and end_val if these are given
start_val	(float) Start point of the line. That is the starting amplitude
end_val	(float) End point of the line. That is the amplitude at the start of the last timeslot

gen_pulse(*gradient=None*, *start_val=None*, *end_val=None*)

    Generate a linear pulse using either the gradient and start value or using the end point to calulate the gradient Note that the scaling and offset parameters are still applied, so unless these values are the default 1.0 and 0.0, then the actual gradient etc will be different Returns the pulse as an array of vales for each timeslot

init_pulse(*gradient=None*, *start_val=None*, *end_val=None*)

    Calculate the gradient if pulse is defined by start and end point values

reset()

    reset attributes to default values

class **PulseGenPeriodic**(*dyn=None*, *params=None*)

    Intermediate class for all periodic pulse generators All of the periodic pulses range from -1 to 1 All have a start phase that can be set between 0 and 2pi

Attributes

num_waves	(float) Number of complete waves (cycles) that occur in the pulse. wavelen and freq calculated from this if it is given
wavelen	(float) Wavelength of the pulse (assuming the speed is 1) freq is calculated from this if it is given
freq	(float) Frequency of the pulse
start_phase	(float) Phase of the pulse signal when t=0

init_pulse(*num_waves=None*, *wavelen=None*, *freq=None*, *start_phase=None*)

    Calculate the wavelength, frequency, number of waves etc from the each other and the other parameters If num_waves is given then the other parameters are worked from this Otherwise if the wavelength is given then it is the driver Otherwise the frequency is used to calculate wavelength and num_waves

reset()

    reset attributes to default values

class **PulseGenSine**(*dyn=None*, *params=None*)

    Generates sine wave pulses

gen_pulse(*num_waves=None*, *wavelen=None*, *freq=None*, *start_phase=None*)

    Generate a sine wave pulse If no params are provided then the class object attributes are used. If they are provided, then these will reinitialise the object attribs. returns the pulse as an array of vales for each timeslot

class **PulseGenSquare** (*dyn=None*, *params=None*)
> Generates square wave pulses
>
> **gen_pulse** (*num_waves=None*, *wavelen=None*, *freq=None*, *start_phase=None*)
> > Generate a square wave pulse If no parameters are pavided then the class object attributes are used. If they are provided, then these will reinitialise the object attribs

class **PulseGenSaw** (*dyn=None*, *params=None*)
> Generates saw tooth wave pulses
>
> **gen_pulse** (*num_waves=None*, *wavelen=None*, *freq=None*, *start_phase=None*)
> > Generate a saw tooth wave pulse If no parameters are pavided then the class object attributes are used. If they are provided, then these will reinitialise the object attribs

class **PulseGenTriangle** (*dyn=None*, *params=None*)
> Generates triangular wave pulses
>
> **gen_pulse** (*num_waves=None*, *wavelen=None*, *freq=None*, *start_phase=None*)
> > Generate a sine wave pulse If no parameters are pavided then the class object attributes are used. If they are provided, then these will reinitialise the object attribs

class **PulseGenGaussian** (*dyn=None*, *params=None*)
> Generates pulses with a Gaussian profile
>
> **gen_pulse** (*mean=None*, *variance=None*)
> > Generate a pulse with Gaussian shape. The peak is centre around the mean and the variance determines the breadth The scaling and offset attributes are applied as an amplitude and fixed linear offset. Note that the maximum amplitude will be scaling + offset.
>
> **reset** ()
> > reset attributes to default values

class **PulseGenGaussianEdge** (*dyn=None*, *params=None*)
> Generate pulses with inverted Gaussian ramping in and out It's intended use for a ramping modulation, which is often required in experimental setups.

Attributes

de-cay_time	(float) Determines the ramping rate. It is approximately the time required to bring the pulse to full amplitude It is set to 1/10 of the pulse time by default

> **gen_pulse** (*decay_time=None*)
> > Generate a pulse that starts and ends at zero and 1.0 in between then apply scaling and offset The tailing in and out is an inverted Gaussian shape
>
> **reset** ()
> > reset attributes to default values

class **PulseGenCrab** (*dyn=None*, *num_coeffs=None*, *params=None*)
> Base class for all CRAB pulse generators Note these are more involved in the optimisation process as they are used to produce piecewise control amplitudes each time new optimisation parameters are tried

Attributes

num_coeffs	(integer) Number of coefficients used for each basis function
num_basis_funcs	(integer) Number of basis functions In this case set at 2 and should not be changed
coeffs	(float array[num_coeffs, num_basis_funcs]) The basis coefficient values
random-ize_coeffs	(bool) If True (default) then the coefficients are set to some random values when initialised, otherwise they will all be equal to self.scaling

estimate_num_coeffs(*dim*)

    Estimate the number coefficients based on the dimensionality of the system. :returns: **num_coeffs** – estimated number of coefficients :rtype: int

get_optim_var_vals()

    Get the parameter values to be optimised :returns: :rtype: list (or 1d array) of floats

init_coeffs(*num_coeffs=None*)

    Generate the initial ceofficent values.

        Parameters num_coeffs : integer

            Number of coefficients used for each basis function If given this overides the default and sets the attribute of the same name.

init_pulse(*num_coeffs=None*)

    Set the initial freq and coefficient values

reset()

    reset attributes to default values

set_optim_var_vals(*param_vals*)

    Set the values of the any of the pulse generation parameters based on new values from the optimisation method Typically this will be the basis coefficients

class **PulseGenCrabFourier**(*dyn=None*, *num_coeffs=None*, *params=None*)

    Generates a pulse using the Fourier basis functions, i.e. sin and cos

Attributes

freqs	(float array[num_coeffs]) Frequencies for the basis functions
randomize_freqs	(bool) If True (default) the some random offset is applied to the frequencies

gen_pulse(*coeffs=None*)

    Generate a pulse using the Fourier basis with the freqs and coeffs attributes.

        Parameters coeffs : float array[num_coeffs, num_basis_funcs]

            The basis coefficient values If given this overrides the default and sets the attribute of the same name.

init_freqs()

    Generate the frequencies These are the Fourier harmonics with a uniformly distributed random offset

init_pulse(*num_coeffs=None*)

    Set the initial freq and coefficient values

reset()

    reset attributes to default values

class **Stats**

    Base class for all optimisation statistics Used for configurations where all timeslots are updated each iteration e.g. exact gradients Note that all times are generated using timeit.default_timer() and are in seconds

Attributes

dyn_gen_name	(string) Text used in some report functions. Makes sense to set it to 'Hamiltonian' when using unitary dynamics Default is simply 'dynamics generator'
num_iter	(integer) Number of iterations of the optimisation algorithm
wall_time_optim_start	(float) Start time for the optimisation
wall_time_optim_end	(float) End time for the optimisation
wall_time_optim	(float) Time elasped during the optimisation
wall_time_dyn_gen_compute	(float) Total wall (elasped) time computing combined dynamics generator (for example combining drift and control Hamiltonians)
wall_time_prop_compute	(float) Total wall (elasped) time computing propagators, that is the time evolution from one timeslot to the next Includes calculating the propagator gradient for exact gradients
wall_time_fwd_prop_compute	(float) Total wall (elasped) time computing combined forward propagation, that is the time evolution from the start to a specific timeslot. Excludes calculating the propagators themselves
wall_time_onwd_prop_compute	(float) Total wall (elasped) time computing combined onward propagation, that is the time evolution from a specific timeslot to the end time. Excludes calculating the propagators themselves
wall_time_gradient_compute	(float) Total wall (elasped) time computing the fidelity error gradient. Excludes calculating the propagator gradients (in exact gradient methods)
num_fidelity_func_calls	(integer) Number of calls to fidelity function by the optimisation algorithm
num_grad_func_calls	(integer) Number of calls to gradient function by the optimisation algorithm
num_tslot_recompute	(integer) Number of time the timeslot evolution is recomputed (It is only computed if any amplitudes changed since the last call)
num_fidelity_computes	(integer) Number of time the fidelity is computed (It is only computed if any amplitudes changed since the last call)
num_grad_computes	(integer) Number of time the gradient is computed (It is only computed if any amplitudes changed since the last call)
num_ctrl_amp_updates	(integer) Number of times the control amplitudes are updated
mean_num_ctrl_amp_updates	(float) Mean number of control amplitude updates per iteration
num_timeslot_changes	(integer) Number of times the amplitudes of a any control in a timeslot changes
mean_num_timeslot_changes	(float) Mean average number of timeslot amplitudes that are changed per update
num_ctrl_amp_changes	(integer) Number of times individual control amplitudes that are changed
mean_num_ctrl_amp_changes	(float) Mean average number of control amplitudes that are changed per update

calculate()
> Perform the calculations (e.g. averages) that are required on the stats Should be called before calling report

report()
> Print a report of the stats to the console

class Dump
> A container for dump items. The lists for dump items is depends on the type Note: abstract class

Attributes

level

parent	(some control object (Dynamics or Optimizer)) aka the host. Object that generates the data that is dumped and is host to this dump object.
dump_dir	(str) directory where files (if any) will be written out the path and be relative or absolute use ~/ to specify user home directory Note: files are only written when write_to_file is True of writeout is called explicitly Defaults to ~/.qtrl_dump
write_to_file	(bool) When set True data and summaries (as configured) will be written interactively to file during the processing Set during instantiation by the host based on its dump_to_file attrib
dump_file_ext	(str) Default file extension for any file names that are auto generated
fname_base	(str) First part of any auto generated file names. This is usually overridden in the subclass
dump_summary	(bool) If True a summary is recorded each time a new item is added to the the dump. Default is True
summary_sep	(str) delimiter for the summary file. default is a space
data_sep	(str) delimiter for the data files (arrays saved to file). default is a space
summary_file	(str) File path for summary file. Automatically generated. Can be set specifically

create_dump_dir()
> Checks dump directory exists, creates it if not

level

> **The level of data dumping that will occur**
>
> > • SUMMARY : A summary will be recorded
> >
> > • FULL : All possible dumping
> >
> > • CUSTOM : Some customised level of dumping
>
> When first set to CUSTOM this is equivalent to SUMMARY. It is then up to the user to specify what specifically is dumped

class **OptimDump**(*optim*, *level='SUMMARY'*)
> A container for dumps of optimisation data generated during the pulse optimisation.

Attributes

dump_summary	(bool) When True summary items are appended to the iter_summary
iter_summary	(list of `optimizer.OptimIterSummary`) Summary at each iteration
dump_fid_err	(bool) When True values are appended to the fid_err_log
fid_err_log	(list of float) Fidelity error at each call of the fid_err_func
dump_grad_norm	(bool) When True values are appended to the fid_err_log
grad_norm_log	(list of float) Gradient norm at each call of the grad_norm_log
dump_grad	(bool) When True values are appended to the grad_log
grad_log	(list of ndarray) Gradients at each call of the fid_grad_func

add_iter_summary()
> add copy of current optimizer iteration summary

dump_all
> True if everything (ignoring the summary) is to be dumped

dump_any
> True if anything other than the summary is to be dumped

update_fid_err_log(*fid_err*)
> add an entry to the fid_err log

update_grad_log(*grad*)
> add an entry to the grad log

update_grad_norm_log(*grad_norm*)
> add an entry to the grad_norm log

writeout (*f=None*)

> write all the logs and the summary out to file(s)

> > **Parameters f** : filename or filehandle

> > > If specified then all summary and object data will go in one file. If None is specified then type specific files will be generated in the dump_dir If a filehandle is specified then it must be a byte mode file as numpy.savetxt is used, and requires this.

class DynamicsDump (*dynamics*, *level='SUMMARY'*)

> A container for dumps of dynamics data. Mainly time evolution calculations

Attributes

dump_summary	(bool) If True a summary is recorded
evo_summary	(list of :class:'tslotcomp.EvoCompSummary') Summary items are appended if dump_summary is True at each recomputation of the evolution.
dump_amps	(bool) If True control amplitudes are dumped
dump_dyn_gen	(bool) If True the dynamics generators (Hamiltonians) are dumped
dump_prop	(bool) If True propagators are dumped
dump_prop_grad	(bool) If True propagator gradients are dumped
dump_fwd_evo	(bool) If True forward evolution operators are dumped
dump_onwd_evo	(bool) If True onward evolution operators are dumped
dump_onto_evo	(bool) If True onto (or backward) evolution operators are dumped
evo_dumps	(list of EvoCompDumpItem) A new dump item is appended at each recomputation of the evolution. That is if any of the calculation objects are to be dumped.

add_evo_comp_summary (*dump_item_idx=None*)

> add copy of current evo comp summary

add_evo_dump ()

> Add dump of current time evolution generating objects

dump_all

> True if all of the calculation objects are to be dumped

dump_any

> True if any of the calculation objects are to be dumped

writeout (*f=None*)

> write all the dump items and the summary out to file(s) :param f: If specified then all summary and object data will go in one file.

> > If None is specified then type specific files will be generated in the dump_dir If a filehandle is specified then it must be a byte mode file as numpy.savetxt is used, and requires this.

class DumpItem

> An item in a dump list

class EvoCompDumpItem (*dump*)

> A copy of all objects generated to calculate one time evolution Note the attributes are only set if the corresponding DynamicsDump **dump_** attribute is set.

writeout (*f=None*)

> write all the objects out to files

> > **Parameters f** : filename or filehandle

> > > If specified then all object data will go in one file. If None is specified then type specific files will be generated in the dump_dir If a filehandle is specified then it must be a byte mode file as numpy.savetxt is used, and requires this.

class **DumpSummaryItem**

> A summary of the most recent iteration Abstract class only

> Attributes: idx : int

>> Index in the summary list in which this is stored

4.2 Functions

4.2.1 Manipulation and Creation of States and Operators

Quantum States

basis(*N*, *n=0*, *offset=0*)

> Generates the vector representation of a Fock state.

>> **Parameters** **N** : int

>>> Number of Fock states in Hilbert space.

>> **n** : int

>>> Integer corresponding to desired number state, defaults to 0 if omitted.

>> **offset** : int (default 0)

>>> The lowest number state that is included in the finite number state representation of the state.

>> **Returns** **state** : qobj

>>> Qobj representing the requested number state `|n>`.

Notes

A subtle incompatibility with the quantum optics toolbox: In QuTiP:

```
basis(N, 0) = ground state
```

but in the 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]]
```

bell_state(*state='00'*)

> Returns the Bell state:

>> |B00> = 1 / sqrt(2)*[|0>|0>+|1>|1>] |B01> = 1 / sqrt(2)*[|0>|0>-|1>|1>] |B10> = 1 / sqrt(2)*[|0>|1>+|1>|0>] |B11> = 1 / sqrt(2)*[|0>|1>-|1>|0>]

Returns Bell_state : qobj

Bell state

bra(*seq*, *dim=2*)

Produces a multiparticle bra state for a list or string, where each element stands for state of the respective particle.

Parameters seq : str / list of ints or characters

Each element defines state of the respective particle. (e.g. [1,1,0,1] or a string "1101"). For qubits it is also possible to use the following conventions: - 'g'/'e' (ground and excited state) - 'u'/'d' (spin up and down) - 'H'/'V' (horizontal and vertical polarization) Note: for dimension > 9 you need to use a list.

dim : int (default: 2) / list of ints

Space dimension for each particle: int if there are the same, list if they are different.

Returns bra : qobj

Examples

```
>>> bra("10")
Quantum object: dims = [[1, 1], [2, 2]], shape = [1, 4], type = bra
Qobj data =
[[ 0.  0.  1.  0.]]
```

```
>>> bra("Hue")
Quantum object: dims = [[1, 1, 1], [2, 2, 2]], shape = [1, 8], type = bra
Qobj data =
[[ 0.  1.  0.  0.  0.  0.  0.  0.]]
```

```
>>> bra("12", 3)
Quantum object: dims = [[1, 1], [3, 3]], shape = [1, 9], type = bra
Qobj data =
[[ 0.  0.  0.  0.  0.  1.  0.  0.  0.]]
```

```
>>> bra("31", [5, 2])
Quantum object: dims = [[1, 1], [5, 2]], shape = [1, 10], type = bra
Qobj data =
[[ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]]
```

coherent(*N*, *alpha*, *offset=0*, *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.

offset : int (default 0)

The lowest number state that is included in the finite number state representation of the state. Using a non-zero offset will make the default method 'analytic'.

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*, *offset=0*, *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.

offset : int (default 0)

> The lowest number state that is included in the finite number state representation of the 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         ]]
```

enr_state_dictionaries(*dims*, *excitations*)

Return the number of states, and lookup-dictionaries for translating a state tuple to a state index, and vice versa, for a system with a given number of components and maximum number of excitations.

Parameters dims: list

A list with the number of states in each sub-system.

excitations : integer

The maximum numbers of dimension

Returns nstates, state2idx, idx2state: integer, dict, dict

The number of states *nstates*, a dictionary for looking up state indices from a state tuple, and a dictionary for looking up state state tuples from state indices.

enr_thermal_dm(*dims*, *excitations*, *n*)

Generate the density operator for a thermal state in the excitation-number- restricted state space defined by the *dims* and *exciations* arguments. See the documentation for enr_fock for a more detailed description of these arguments. The temperature of each mode in dims is specified by the average number of excitatons *n*.

Parameters dims : list

A list of the dimensions of each subsystem of a composite quantum system.

excitations : integer

The maximum number of excitations that are to be included in the state space.

n : integer

The average number of exciations in the thermal state. *n* can be a float (which then applies to each mode), or a list/array of the same length as dims, in which each element corresponds specifies the temperature of the corresponding mode.

Returns dm : Qobj

Thermal state density matrix.

enr_fock(*dims*, *excitations*, *state*)

Generate the Fock state representation in a excitation-number restricted state space. The *dims* argument is a list of integers that define the number of quantums states of each component of a composite quantum system, and the *excitations* specifies the maximum number of excitations for the basis states that are to be included in the state space. The *state* argument is a tuple of integers that specifies the state (in the number basis representation) for which to generate the Fock state representation.

Parameters dims : list

A list of the dimensions of each subsystem of a composite quantum system.

excitations : integer

The maximum number of excitations that are to be included in the state space.

state : list of integers

The state in the number basis representation.

Returns ket : Qobj

A Qobj instance that represent a Fock state in the exication-number- restricted state space defined by *dims* and *exciations*.

fock (*N*, *n=0*, *offset=0*)

Bosonic Fock (number) state.

Same as `qutip.states.basis`.

> **Parameters** **N** : int
>
> > Number of states in the Hilbert space.
>
> **n** : int
>
> > `int` for desired number state, defaults to 0 if omitted.
>
> **Returns** Requested number state $|n\rangle$.

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*, *n=0*, *offset=0*)

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.
>
> **n** : 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]]
```

ghz_state (*N=3*)

Returns the N-qubit GHZ-state.

> **Parameters** **N** : int (default=3)
>
> > Number of qubits in state
>
> **Returns** **G** : qobj
>
> > N-qubit GHZ-state

maximally_mixed_dm(*N*)

Returns the maximally mixed density matrix for a Hilbert space of dimension N.

> **Parameters N** : int
>
> > Number of basis states in Hilbert space.
>
> **Returns dm** : qobj
>
> > Thermal state density matrix.

ket(*seq*, *dim=2*)

Produces a multiparticle ket state for a list or string, where each element stands for state of the respective particle.

> **Parameters seq** : str / list of ints or characters
>
> > Each element defines state of the respective particle. (e.g. [1,1,0,1] or a string "1101"). For qubits it is also possible to use the following conventions: - 'g'/'e' (ground and excited state) - 'u'/'d' (spin up and down) - 'H'/'V' (horizontal and vertical polarization) Note: for dimension > 9 you need to use a list.
>
> > **dim** : int (default: 2) / list of ints
> >
> > > Space dimension for each particle: int if there are the same, list if they are different.
>
> **Returns ket** : qobj

Examples

```
>>> ket("10")
Quantum object: dims = [[2, 2], [1, 1]], shape = [4, 1], type = ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 1.]
 [ 0.]]
```

```
>>> ket("Hue")
Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = [8, 1], type = ket
Qobj data =
[[ 0.]
 [ 1.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]]
```

```
>>> ket("12", 3)
Quantum object: dims = [[3, 3], [1, 1]], shape = [9, 1], type = ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 1.]
 [ 0.]
 [ 0.]
 [ 0.]]
```

```
>>> ket("31", [5, 2])
Quantum object: dims = [[5, 2], [1, 1]], shape = [10, 1], type = ket
Qobj data =
[[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 1.]
 [ 0.]
 [ 0.]]
```

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]]
```

phase_basis(*N*, *m*, *phi0=0*)

Basis vector for the mth phase of the Pegg-Barnett phase operator.

> **Parameters** **N** : int
>
>> Number of basis vectors in Hilbert space.
>
> **m** : int
>
>> Integer corresponding to the mth discrete phase phi_m=phi0+2*pi*m/N
>
> **phi0** : float (default=0)
>
>> Reference phase angle.
>
> **Returns** **state** : qobj
>
>> Ket vector for mth Pegg-Barnett phase operator basis state.

Notes

The Pegg-Barnett basis states form a complete set over the truncated Hilbert space.

projection(*N*, *n*, *m*, *offset=0*)

The projection operator that projects state $|m>$ on state $|n>$.

> **Parameters** **N** : int
>
>> Number of basis states in Hilbert space.

> **n, m** : float
>
>> The number states in the projection.
>
> **offset** : int (default 0)
>
>> The lowest number state that is included in the finite number state representation of the projector.
>
> **Returns oper** : qobj
>
>> Requested projection operator.

qutrit_basis()
    Basis states for a three level system (qutrit)

> **Returns qstates** : array
>
>> Array of qutrit basis vectors

singlet_state()
    Returns the two particle singlet-state:

> |S>=1/sqrt(2)*[|0>|1>-|1>|0>]

that is identical to the fourth bell state.

> **Returns Bell_state** : qobj
>
>> |B11> Bell state

spin_state(*j*, *m*, *type='ket'*)
    Generates the spin state |j, m>, i.e. the eigenstate of the spin-j Sz operator with eigenvalue m.

> **Parameters j** : float
>
>> The spin of the state ().
>
> **m** : int
>
>> Eigenvalue of the spin-j Sz operator.
>
> **type** : string {'ket', 'bra', 'dm'}
>
>> Type of state to generate.
>
> **Returns state** : qobj
>
>> Qobj quantum object for spin state

spin_coherent(*j*, *theta*, *phi*, *type='ket'*)
    Generates the spin state |j, m>, i.e. the eigenstate of the spin-j Sz operator with eigenvalue m.

> **Parameters j** : float
>
>> The spin of the state.
>
> **theta** : float
>
>> Angle from z axis.
>
> **phi** : float
>
>> Angle from x axis.
>
> **type** : string {'ket', 'bra', 'dm'}
>
>> Type of state to generate.
>
> **Returns state** : qobj
>
>> Qobj quantum object for spin coherent state

state_number_enumerate(*dims*, *excitations=None*, *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.
>
> **excitations** : integer (None)
>
>> Restrict state space to states with excitation numbers below or equal to this value.
>
> **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
```

> **Parameters dims** : list or array
>
>> The quantum state dimensions array, as it would appear in a Qobj.
>
> **state** : list
>
>> State number array.
>
> **Returns idx** : int
>
>> 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.

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 aggressively.

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]]
```

zero_ket(*N*, *dims=None*)

Creates the zero ket vector with shape Nx1 and dimensions *dims*.

> **Parameters** **N** : int
>
>> Hilbert space dimensionality
>
> **dims** : list
>
>> Optional dimensions if ket corresponds to a composite Hilbert space.
>
> **Returns** **zero_ket** : qobj
>
>> Zero ket on given Hilbert space.

Quantum Operators

This module contains functions for generating Qobj representation of a variety of commonly occuring quantum operators.

charge(*Nmax*, *Nmin=None*, *frac=1*)

Generate the diagonal charge operator over charge states from Nmin to Nmax.

> **Parameters** **Nmax** : int
>
>> Maximum charge state to consider.
>
> **Nmin** : int (default = -Nmax)
>
>> Lowest charge state to consider.
>
> **frac** : float (default = 1)
>
>> Specify fractional charge if needed.
>
> **Returns** **C** : Qobj

Charge operator over [Nmin,Nmax].

Notes

New in version 3.2.

commutator(*A*, *B*, *kind='normal'*)

Return the commutator of kind *kind* (normal, anti) of the two operators A and B.

create(*N*, *offset=0*)

Creation (raising) operator.

> **Parameters** **N** : int
>
> > Dimension of Hilbert space.
>
> **Returns** **oper** : qobj
>
> > Qobj for raising operator.
>
> **offset** : int (default 0)
>
> > The lowest number state that is included in the finite number state representation of
> > the 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*, *offset=0*)

Destruction (lowering) operator.

> **Parameters** **N** : int
>
> > Dimension of Hilbert space.
>
> **offset** : int (default 0)
>
> > The lowest number state that is included in the finite number state representation of
> > the operator.
>
> **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*, *offset=0*)

Single-mode displacement operator.

Parameters N : int

> Dimension of Hilbert space.

alpha : float/complex

> Displacement amplitude.

offset : int (default 0)

> The lowest number state that is included in the finite number state representation of the operator.

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]]
```

enr_destroy(*dims*, *excitations*)

> Generate annilation operators for modes in a excitation-number-restricted state space. For example, consider a system consisting of 4 modes, each with 5 states. The total hilbert space size is 5**4 = 625. If we are only interested in states that contain up to 2 excitations, we only need to include states such as
>
> > (0, 0, 0, 0) (0, 0, 0, 1) (0, 0, 0, 2) (0, 0, 1, 0) (0, 0, 1, 1) (0, 0, 2, 0) ...
>
> This function creates annihilation operators for the 4 modes that act within this state space:
>
> > a1, a2, a3, a4 = enr_destroy([5, 5, 5, 5], excitations=2)
>
> From this point onwards, the annihiltion operators a1, ..., a4 can be used to setup a Hamiltonian, collapse operators and expectation-value operators, etc., following the usual pattern.
>
> **Parameters dims** : list
>
> > A list of the dimensions of each subsystem of a composite quantum system.
>
> **excitations** : integer
>
> > The maximum number of excitations that are to be included in the state space.
>
> **Returns a_ops** : list of qobj
>
> > A list of annihilation operators for each mode in the composite quantum system described by dims.

enr_identity(*dims*, *excitations*)

> Generate the identity operator for the excitation-number restricted state space defined by the *dims* and *exciations* arguments. See the docstring for enr_fock for a more detailed description of these arguments.
>
> **Parameters dims** : list
>
> > A list of the dimensions of each subsystem of a composite quantum system.
>
> **excitations** : integer
>
> > The maximum number of excitations that are to be included in the state space.
>
> **state** : list of integers
>
> > The state in the number basis representation.

Returns op : Qobj

A Qobj instance that represent the identity operator in the exication-number-restricted state space defined by *dims* and *exciations*.

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 / ndarray

`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*, *offset=0*)

Quantum object for number operator.

Parameters N : int

The dimension of the Hilbert space.

offset : int (default 0)

The lowest number state that is included in the finite number state representation of the operator.

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 or list of ints

> > > Dimension of Hilbert space. If provided as a list of ints, then the dimension is the product over this list, but the `dims` property of the new Qobj are set to this list.

> > **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 or list of ints

> > > Dimension of Hilbert space. If provided as a list of ints, then the dimension is the product over this list, but the `dims` property of the new Qobj are set to this list.

> > **Returns oper** : qobj

> > > Identity operator Qobj.

momentum (*N*, *offset=0*)

> Momentum operator p=-1j/sqrt(2)*(a-a.dag())

> > **Parameters N** : int

> > > Number of Fock states in Hilbert space.

> > **offset** : int (default 0)

> > > The lowest number state that is included in the finite number state representation of the operator.

> > **Returns oper** : qobj

> > > Momentum operator as Qobj.

phase (*N*, *phi0=0*)

> Single-mode Pegg-Barnett phase operator.

> > **Parameters N** : int

> > > Number of basis states in Hilbert space.

> > **phi0** : float

Reference phase.

> **Returns oper** : qobj
>
>> Phase operator with respect to reference phase.

Notes

The Pegg-Barnett phase operator is Hermitian on a truncated Hilbert space.

position (*N*, *offset=0*)

Position operator x=1/sqrt(2)*(a+a.dag())

> **Parameters N** : int
>
>> Number of Fock states in Hilbert space.
>
> **offset** : int (default 0)
>
>> The lowest number state that is included in the finite number state representation of the operator.
>
> **Returns oper** : qobj
>
>> Position operator as Qobj.

qdiags (*diagonals*, *offsets*, *dims=None*, *shape=None*)

Constructs an operator from an array of diagonals.

> **Parameters diagonals** : sequence of array_like
>
>> Array of elements to place along the selected diagonals.
>
> **offsets** : sequence of ints
>
>> **Sequence for diagonals to be set:**
>>
>> - k=0 main diagonal
>> - k>0 kth upper diagonal
>> - k<0 kth lower diagonal
>
> **dims** : list, optional
>
>> Dimensions for operator
>
> **shape** : list, tuple, optional
>
>> Shape of operator. If omitted, a square operator large enough to contain the diagonals is generated.

See also:

`scipy.sparse.diags`

Notes

This function requires SciPy 0.11+.

Examples

```
>>> qdiags(sqrt(range(1,4)),1)
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.         ]]
```

qutrit_ops()
> Operators for a three level system (qutrit).
>
>> **Returns** opers: array
>>
>>> *array* of qutrit operators.

qzero(*N*)
> Zero operator
>
>> **Parameters** **N** : int or list of ints
>>
>>> Dimension of Hilbert space. If provided as a list of ints, then the dimension is the product over this list, but the `dims` property of the new Qobj are set to this list.
>>
>> **Returns** **qzero** : qobj
>>
>>> Zero operator Qobj.

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.]]
```

spin_Jx(*j*)

Spin-j x operator

> **Parameters j** : float
>
> > Spin of operator
>
> **Returns op** : Qobj
>
> > qobj representation of the operator.

spin_Jy(*j*)

Spin-j y operator

> **Parameters j** : float
>
> > Spin of operator
>
> **Returns op** : Qobj
>
> > qobj representation of the operator.

spin_Jz(*j*)

Spin-j z operator

> **Parameters j** : float
>
> > Spin of operator
>
> **Returns op** : Qobj
>
> > qobj representation of the operator.

spin_Jm(*j*)

Spin-j annihilation operator

> **Parameters j** : float
>
> > Spin of operator
>
> **Returns op** : Qobj
>
> > qobj representation of the operator.

spin_Jp(*j*)

Spin-j creation operator

> **Parameters j** : float

Spin of operator

Returns op : Qobj

`qobj` representation of the operator.

squeeze(*N, z, offset=0*)

Single-mode Squeezing operator.

Parameters N : int

Dimension of hilbert space.

z : float/complex

Squeezing parameter.

offset : int (default 0)

The lowest number state that is included in the finite number state representation of the operator.

Returns oper : `qutip.qobj.Qobj`

Squeezing operator.

Examples

```
>>> squeeze(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]]
```

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.

tunneling(*N, m=1*)

Tunneling operator with elements of the form $\sum |N\rangle\langle N+m| + |N+m\rangle\langle N|$.

Parameters N : int

Number of basis states in Hilbert space.

m : int (default = 1)

Number of excitations in tunneling event.

Returns T : Qobj

Tunneling operator.

Notes

New in version 3.2.

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, ndarray, list
>
> > If int, then shape of output operator. If list/ndarray then eigenvalues of generated density matrix.
>
> **density** : float
>
> > Density between [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 matrices., choosing a low density will result in an error as no diagonal elements will be generated such that $Tr(\rho) = 1$.

rand_dm_ginibre (*N=2*, *rank=None*, *dims=None*)

Returns a Ginibre random density operator of dimension `dim` and rank `rank` by using the algorithm of *[BCSZ08]*. If `rank` is *None*, a full-rank (Hilbert-Schmidt ensemble) random density operator will be returned.

> **Parameters** **N** : int
>
> > Dimension of the density operator to be returned.
>
> **dims** : list
>
> > Dimensions of quantum object. Used for specifying tensor structure. Default is dims=[[N],[N]].
>
> **rank** : int or None
>
> > Rank of the sampled density operator. If None, a full-rank density operator is generated.
>
> **Returns** **rho** : Qobj
>
> > An N × N density operator sampled from the Ginibre or Hilbert-Schmidt distribution.

rand_dm_hs (*N=2*, *dims=None*)

Returns a Hilbert-Schmidt random density operator of dimension `dim` and rank `rank` by using the algorithm of *[BCSZ08]*.

> **Parameters** **N** : int
>
>> Dimension of the density operator to be returned.
>
> **dims** : list
>
>> Dimensions of quantum object. Used for specifying tensor structure. Default is dims=[[N],[N]].
>
> **Returns** **rho** : Qobj
>
>> A dim × dim density operator sampled from the Ginibre or Hilbert-Schmidt distribution.

rand_herm(*N*, *density=0.75*, *dims=None*, *pos_def=False*)

> Creates a random NxN sparse Hermitian quantum object.
>
> If 'N' is an integer, uses $H = 0.5 * (X + X^+)$ where X is a randomly generated quantum operator with a given *density*. Else uses complex Jacobi rotations when 'N' is given by an array.
>
> **Parameters** **N** : int, list/ndarray
>
>> If int, then shape of output operator. If list/ndarray then eigenvalues of generated 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]].
>
> **pos_def** : bool (default=False)
>
>> Return a positive semi-definite matrix (by diagonal dominance).
>
> **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
>
>> Left-dimensions of quantum object. Used for specifying tensor structure. Default is dims=[[N]].
>
> **Returns** **oper** : qobj
>
>> Nx1 ket state quantum operator.

rand_ket_haar(*N=2*, *dims=None*)

> Returns a Haar random pure state of dimension `dim` by applying a Haar random unitary to a fixed pure state.
>
> **Parameters** **N** : int
>
>> Dimension of the state vector to be returned.
>
> **dims** : list of ints, or None
>
>> Left-dimensions of the resultant quantum object. If None, [N] is used.

Returns psi : Qobj

A random state vector drawn from the Haar measure.

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.

rand_unitary_haar(*N=2*, *dims=None*)

Returns a Haar random unitary matrix of dimension `dim`, using the algorithm of *[Mez07]*.

Parameters N : int

Dimension of the unitary to be returned.

dims : list of lists of int, or None

Dimensions of quantum object. Used for specifying tensor structure. Default is dims=[[N],[N]].

Returns U : Qobj

Unitary of dims `[[dim], [dim]]` drawn from the Haar measure.

rand_super(*N=5*, *dims=None*)

Returns a randomly drawn superoperator acting on operators acting on N dimensions.

Parameters N : int

Square root of the dimension of the superoperator to be returned.

dims : list

Dimensions of quantum object. Used for specifying tensor structure. Default is dims=[[[N],[N]], [[N],[N]]].

rand_super_bcsz(*N=2*, *enforce_tp=True*, *rank=None*, *dims=None*)

Returns a random superoperator drawn from the Bruzda et al ensemble for CPTP maps *[BCSZ08]*. Note that due to finite numerical precision, for ranks less than full-rank, zero eigenvalues may become slightly negative, such that the returned operator is not actually completely positive.

Parameters N : int

Square root of the dimension of the superoperator to be returned.

enforce_tp : bool

If True, the trace-preserving condition of *[BCSZ08]* is enforced; otherwise only complete positivity is enforced.

rank : int or None

Rank of the sampled superoperator. If None, a full-rank superoperator is generated.

dims : list

Dimensions of quantum object. Used for specifying tensor structure. Default is dims=[[[N],[N]], [[N],[N]]].

Returns rho : Qobj

A superoperator acting on vectorized dim × dim density operators, sampled from the BCSZ distribution.

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:

```
Ladder:            Lambda:                  Vee:
                                |two>                         |three>
     -------|three>         -------                       -------
        |                  / \          |one>              /
        |                 /   \        -------            /
        |                /     \          \              /
     -------|two>       /       \          \            /
        |             /         \          \          /
        |            /           \          \  /
        |           /    -------          \ /
     -------|one>    -------   |three>      -------
                    |one>                  |two>
```
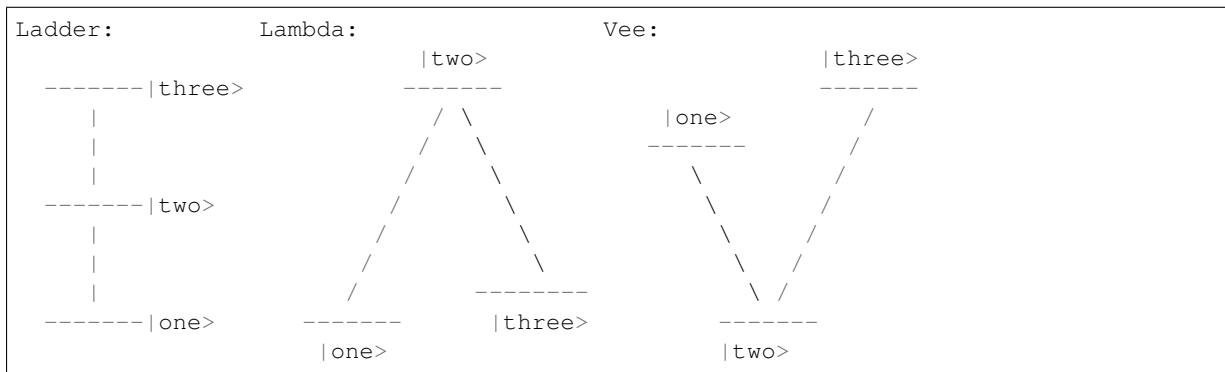
References

The naming of qutip operators follows the convention in *[R1]* .

___

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.

Superoperators and Liouvillians

operator_to_vector(*op*)
   Create a vector representation of a quantum operator given the matrix representation.

vector_to_operator(*op*)
   Create a matrix representation given a quantum operator in vector form.

liouvillian(*H*, *c_ops=[]*, *data_only=False*, *chi=None*)

Assembles the Liouvillian superoperator from a Hamiltonian and a `list` of collapse operators. Like liouvillian, but with an experimental implementation which avoids creating extra Qobj instances, which can be advantageous for large systems.

>**Parameters** **H** : qobj
>
>>System Hamiltonian.
>
>**c_ops** : array_like
>
>>A `list` or `array` of collapse operators.
>
>**Returns** **L** : qobj
>
>>Liouvillian 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.

sprepost(*A*, *B*)

Superoperator formed from pre-multiplication by operator A and post- multiplication of operator B.

>**Parameters** **A** : Qobj
>
>>Quantum operator for pre-multiplication.
>
>**B** : Qobj
>
>>Quantum operator for post-multiplication.
>
>**Returns** **super** : Qobj
>
>>Superoperator formed from input quantum objects.

lindblad_dissipator(*a*, *b=None*, *data_only=False*)

Lindblad dissipator (generalized) for a single pair of collapse operators (a, b), or for a single collapse operator (a) when b is not specified:

$$\mathcal{D}[a, b]\rho = a\rho b^\dagger - \frac{1}{2}a^\dagger b\rho - \frac{1}{2}\rho a^\dagger b$$

>**Parameters** **a** : qobj
>
>>Left part of collapse operator.
>
>**b** : qobj (optional)
>
>>Right part of collapse operator. If not specified, b defaults to a.
>
>**Returns** **D** : qobj
>
>>Lindblad dissipator superoperator.

Superoperator Representations

This module implements transformations between superoperator representations, including supermatrix, Kraus, Choi and Chi (process) matrix formalisms.

to_choi(*q_oper*)

> Converts a Qobj representing a quantum map to the Choi representation, such that the trace of the returned operator is equal to the dimension of the system.

> > **Parameters q_oper** : Qobj
> >
> > > Superoperator to be converted to Choi representation. If `q_oper` is `type="oper"`, then it is taken to act by conjugation, such that `to_choi(A) == to_choi(sprepost(A, A.dag()))`.
> >
> > **Returns choi** : Qobj
> >
> > > A quantum object representing the same map as `q_oper`, such that `choi. superrep == "choi"`.
> >
> > **Raises TypeError: if the given quantum object is not a map, or cannot be converted**
> >
> > > to Choi representation.

to_super(*q_oper*)

> Converts a Qobj representing a quantum map to the supermatrix (Liouville) representation.

> > **Parameters q_oper** : Qobj
> >
> > > Superoperator to be converted to supermatrix representation. If `q_oper` is `type="oper"`, then it is taken to act by conjugation, such that `to_super(A) == sprepost(A, A.dag())`.
> >
> > **Returns superop** : Qobj
> >
> > > A quantum object representing the same map as `q_oper`, such that `superop. superrep == "super"`.
> >
> > **Raises TypeError**
> >
> > > If the given quantum object is not a map, or cannot be converted to supermatrix representation.

to_kraus(*q_oper*)

> Converts a Qobj representing a quantum map to a list of quantum objects, each representing an operator in the Kraus decomposition of the given map.

> > **Parameters q_oper** : Qobj
> >
> > > Superoperator to be converted to Kraus representation. If `q_oper` is `type="oper"`, then it is taken to act by conjugation, such that `to_kraus(A) == to_kraus(sprepost(A, A.dag())) == [A]`.
> >
> > **Returns kraus_ops** : list of Qobj
> >
> > > A list of quantum objects, each representing a Kraus operator in the decomposition of `q_oper`.
> >
> > **Raises TypeError: if the given quantum object is not a map, or cannot be**
> >
> > > decomposed into Kraus operators.

4.2.2 Functions acting on states and operators

Expectation Values

expect(*oper*, *state*)

> Calculates the expectation value for operator(s) and state(s).

Parameters oper : qobj/array-like

A single or a *list* or operators for expectation value.

state : qobj/array-like

A single or a *list* of quantum states or density matrices.

Returns expt : float/complex/array-like

Expectation value. `real` if *oper* is Hermitian, `complex` otherwise. A (nested) array of expectaction values of state or operator are arrays.

Examples

```
>>> expect(num(4), basis(4, 3))
3
```

variance(*oper*, *state*)

Variance of an operator for the given state vector or density matrix.

Parameters oper : qobj

Operator for expectation value.

state : qobj/list

A single or *list* of quantum states or density matrices..

Returns var : float

Variance of operator 'oper' for given state.

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]]
```

super_tensor(**args*)

Calculates the tensor product of input superoperators, by tensoring together the underlying Hilbert spaces on which each vectorized operator acts.

Parameters args : array_like

list or array of quantum objects with type="super".

> **Returns obj** : qobj
>
> > A composite quantum object.

composite(**args*)

> Given two or more operators, kets or bras, returns the Qobj corresponding to a composite system over each argument. For ordinary operators and vectors, this is the tensor product, while for superoperators and vectorized operators, this is the column-reshuffled tensor product.
>
> If a mix of Qobjs supported on Hilbert and Liouville spaces are passed in, the former are promoted. Ordinary operators are assumed to be unitaries, and are promoted using to_super, while kets and bras are promoted by taking their projectors and using operator_to_vector(ket2dm(arg)).

tensor_contract(*qobj*, **pairs*)

> Contracts a qobj along one or more index pairs. Note that this uses dense representations and thus should *not* be used for very large Qobjs.
>
> > **Parameters pairs** : tuple
> >
> > > One or more tuples (i, j) indicating that the i and j dimensions of the original qobj should be contracted.
> >
> > **Returns cqobj** : Qobj
> >
> > > The original Qobj with all named index pairs contracted away.

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:** qutip.qobj
> >
> > > A density matrix with the selected subsystems transposed.

Entropy Functions

concurrence(*rho*)

> Calculate the concurrence entanglement measure for a two-qubit state.
>
> > **Parameters state** : qobj
> >
> > > Ket, bra, or density matrix for a two-qubit state.
> >
> > **Returns concur** : float
> >
> > > Concurrence

References

[R2]

entropy_conditional(*rho*, *selB*, *base=2.718281828459045*, *sparse=False*)

Calculates the conditional entropy $S(A|B) = S(A, B) - S(B)$ of a selected 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
```

Density Matrix Metrics

This module contains a collection of functions for calculating metrics (distance measures) between states and operators.

fidelity (*A*, *B*)

    Calculates the fidelity (pseudo-metric) between two density matrices. See: Nielsen & Chuang, "Quantum Computation and Quantum Information"

> **Parameters A** : qobj
>
> > Density matrix or state vector.
>
> **B** : qobj
>
> > Density matrix or state vector 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 matrices.. See: Nielsen & Chuang, "Quantum Computation and Quantum Information"

> **Parameters A** : qobj
>
> > Density matrix or state vector.
>
> **B** : qobj
>
> > Density matrix or state vector with same dimensions as A.

tol : float

> Tolerance used by sparse eigensolver, if used. (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
```

bures_dist(*A*, *B*)

> Returns the Bures distance between two density matrices A & B.
>
> The Bures distance ranges from 0, for states with unit fidelity, to sqrt(2).
>
> > **Parameters A** : qobj
> >
> > > Density matrix or state vector.
> >
> > **B** : qobj
> >
> > > Density matrix or state vector with same dimensions as A.
> >
> > **Returns dist** : float
> >
> > > Bures distance between density matrices.

bures_angle(*A*, *B*)

> Returns the Bures Angle between two density matrices A & B.
>
> The Bures angle ranges from 0, for states with unit fidelity, to pi/2.
>
> > **Parameters A** : qobj
> >
> > > Density matrix or state vector.
> >
> > **B** : qobj
> >
> > > Density matrix or state vector with same dimensions as A.
> >
> > **Returns angle** : float
> >
> > > Bures angle between density matrices.

hilbert_dist(*A*, *B*)

> Returns the Hilbert-Schmidt distance between two density matrices A & B.
>
> > **Parameters A** : qobj
> >
> > > Density matrix or state vector.
> >
> > **B** : qobj
> >
> > > Density matrix or state vector with same dimensions as A.
> >
> > **Returns dist** : float
> >
> > > Hilbert-Schmidt distance between density matrices.

Notes

See V. Vedral and M. B. Plenio, Phys. Rev. A 57, 1619 (1998).

average_gate_fidelity(*oper*, *target=None*)

Given a Qobj representing the supermatrix form of a map, returns the average gate fidelity (pseudo-metric) of that map.

> **Parameters A** : Qobj
>
>> Quantum object representing a superoperator.
>
> **target** : Qobj
>
>> Quantum object representing the target unitary; the inverse is applied before evaluating the fidelity.
>
> **Returns fid** : float
>
>> Fidelity pseudo-metric between A and the identity superoperator, or between A and the target superunitary.

process_fidelity(*U1*, *U2*, *normalize=True*)

Calculate the process fidelity given two process operators.

Continous Variables

This module contains a collection functions for calculating continuous 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
>
>> List of operators that defines the basis for the correlation matrix.
>
> **rho** : 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** : ndarray
>
>> 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
>
>> List of operators that defines the basis for the covariance matrix.
>
> **rho** : Qobj
>
>> Density matrix for which to calculate the covariance matrix.
>
> **symmetrized** : bool {True, False}

Flag indicating whether the symmetrized (default) or non-symmetrized correlation matrix is to be calculated.

> **Returns corr_mat** : ndarray
>
> > A 2-dimensional array of covariance values.

correlation_matrix_field (*a1*, *a2*, *rho=None*)

Calculates 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** : Qobj
>
> > Field operator for mode 1.
>
> **a2** : Qobj
>
> > Field operator for mode 2.
>
> **rho** : Qobj
>
> > Density matrix for which to calculate the covariance matrix.
>
> **Returns cov_mat** : ndarray
>
> > Array of complex numbers or Qobj's 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** : Qobj
>
> > Field operator for mode 1.
>
> **a2** : Qobj
>
> > Field operator for mode 2.
>
> **rho** : Qobj
>
> > Density matrix for which to calculate the covariance matrix.
>
> **Returns corr_mat** : ndarray
>
> > Array of complex numbers or Qobj's 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*)

Calculates 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 annihilation operators *a1* and *a2* for the two modes are supplied instead, the quadrature correlation matrix is constructed from the annihilation operators before then the covariance matrix is calculated.

> **Parameters a1** : Qobj
>
> > Field operator for mode 1.
>
> **a2** : Qobj
>
> > Field operator for mode 2.
>
> **R** : ndarray
>
> > The quadrature correlation matrix.
>
> **rho** : Qobj
>
> > Density matrix for which to calculate the covariance matrix.

> **Returns cov_mat** : ndarray
>
>> A 2-dimensional array of covariance values.

logarithmic_negativity(*V*)

Calculates the logarithmic negativity given a 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 Gaussian state that is described by the the Wigner covariance matrix V.

4.2.3 Dynamics and Time-Evolution

Schrödinger Equation

This module provides solvers for the unitary Schrodinger equation.

sesolve(*H*, *rho0*, *tlist*, *e_ops=[]*, *args={}*, *options=None*, *progress_bar=<qutip.ui.progressbar.BaseProgressBar object>*, *_safe_mode=True*)

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 (*e_ops*). If e_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.
>
> **e_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:** `qutip.solver`
>
>> An instance of the class `qutip.solver`, 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 *e_ops* is an empty list], or nothing if a callback function was given inplace of operators for which to calculate the expectation values.

Master Equation

This module provides solvers for the Lindblad master equation and von Neumann equation.

mesolve(*H*, *rho0*, *tlist*, *c_ops=[]*, *e_ops=[]*, *args={}*, *options=None*, *progress_bar=None*, *_safe_mode=True*)

Master equation evolution of a density matrix for a given Hamiltonian and set of collapse operators, or a Liouvillian.

Evolve the state vector or density matrix (*rho0*) using a given Hamiltonian (*H*) and an [optional] set of collapse operators (*c_ops*), by integrating the set of ordinary differential equations that define the system. In the absence of collapse 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 (*e_ops*). If e_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.

If either *H* or the Qobj elements in *c_ops* are superoperators, they will be treated as direct contributions to the total system Liouvillian. This allows to solve master equations that are not on standard Lindblad form by passing a custom Liouvillian in place of either the *H* or *c_ops* elements.

Time-dependent operators

For time-dependent problems, *H* and *c_ops* can be callback functions that takes two arguments, time and *args*, and returns the Hamiltonian or Liouvillian 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*), a callback function (*list callback format*) that evaluates to the time-dependent coefficient for the corresponding operator, or a NumPy array (*list array format*) which specifies the value of the coefficient to the corresponding operator for each value of t in tlist.

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).

  H = [[H0, np.sin(w*tlist)], [H1, np.sin(2*w*tlist)]]

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.

Additional options

Additional options to mesolve can be set via the *options* argument, which should be an instance of `qutip.solver.Options`. Many ODE integration options can be set this way, and the *store_states* and *store_final_state* options can be used to store states even though expectation values are requested via the *e_ops* argument.

Note: If an element in the list-specification of the Hamiltonian or the list of collapse operators are in superoperator form 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. mesolve 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 in scipy which will raise an NotImplemented exception.

> **Parameters** **H** : `qutip.Qobj`
>
> > System Hamiltonian, or a callback function for time-dependent Hamiltonians, or alternatively a system Liouvillian.
> >
> > **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, or a list of Liouvillian super-operators.
> >
> > **e_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.Options`
> >
> > with options for the solver.
> >
> > **progress_bar** : BaseProgressBar
> >
> > Optional instance of BaseProgressBar, or a subclass thereof, for showing the progress of the simulation.
>
> **Returns** result: `qutip.Result`
>
> > An instance of the class `qutip.Result`, which contains either an *array result.expect* of expectation values for the times specified by *tlist*, or an *array result.states* of state vectors or density matrices corresponding to the times in *tlist* [if *e_ops* is an empty list], or nothing if a callback function was given in place of operators for which to calculate the expectation values.

Monte Carlo Evolution

mcsolve (*H*, *psi0*, *tlist*, *c_ops=[]*, *e_ops=[]*, *ntraj=None*, *args={}*, *options=None*, *progress_bar=True*, *map_func=None*, *map_kwargs=None*, *_safe_mode=True*)
    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 Options 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:

```python
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_ops = [C0, [C1, C1_coeff]]

args={'a': A, 'w': W}
```

or in String (Cython) format we could write:

```python
H = [H0, [H1, 'sin(w*t)']]

c_ops = [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** : *qutip.Qobj*
>
> > System Hamiltonian.
>
> **psi0** : *qutip.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** : Options
>
> > Instance of ODE solver options.
>
> **progress_bar: BaseProgressBar**
>
> > Optional instance of BaseProgressBar, or a subclass thereof, for showing the progress of the simulation. Set to None to disable the progress bar.
>
> **map_func: function**
>
> > A map function for managing the calls to the single-trajactory solver.
>
> **map_kwargs: dictionary**
>
> > Optional keyword arguments to the map_func function.
>
> **Returns** **results** : *qutip.solver.Result*
>
> > Object storing all results from the simulation.

> **Note:** It is possible to reuse the random number seeds from a previous run of the mcsolver by passing the output Result object seeds via the Options class, i.e. Options(seeds=prev_result.seeds).

Exponential Series

essolve (*H*, *rho0*, *tlist*, *c_op_list*, *e_ops*)

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 (*e_ops*).

    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.

    e_ops : 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.

Bloch-Redfield Master Equation

brmesolve (*H*, *psi0*, *tlist*, *a_ops=[]*, *e_ops=[]*, *c_ops=[]*, *args={}*, *use_secular=True*, *sec_cutoff=0.1*, *tol=1e-12*, *spectra_cb=None*, *options=None*, *progress_bar=None*, *_safe_mode=True*)

Solves for the dynamics of a system using the Bloch-Redfield master equation, given an input Hamiltonian, Hermitian bath-coupling terms and their associated spectrum functions, as well as possible Lindblad collapse operators.

For time-independent systems, the Hamiltonian must be given as a Qobj, whereas the bath-coupling terms (a_ops), must be written as a nested list of operator - spectrum function pairs, where the frequency is specified by the *w* variable.

Example

> a_ops = [[a+a.dag(),lambda w: 0.2*(w>=0)]]

For time-dependent systems, the Hamiltonian, a_ops, and Lindblad collapse operators (c_ops), can be specified in the QuTiP string-based time-dependent format. For the a_op spectra, the frequency variable must be *w*, and the string cannot contain any other variables other than the possibility of having a time-dependence through the time variable *t*:

Example

> a_ops = [[a+a.dag(), '0.2*exp(-t)*(w>=0)']]

It is also possible to use Cubic_Spline objects for time-dependence. In the case of a_ops, Cubic_Splines must be passed as a tuple:

Example

> a_ops = [[a+a.dag(), (f(w), g(t)]]

where f(w) and g(t) are strings or Cubic_spline objects for the bath spectrum and time-dependence, respectively.

Finally, if one has bath-couplimg terms of the form H = f(t)*a + conj[f(t)]*a.dag(), then the correct input format is

Example

> a_ops = [[(a,a.dag()), (f(w), g1(t), g2(t))],...]

where f(w) is the spectrum of the operators while g1(t) and g2(t) are the time-dependence of the operators *a* and *a.dag()*, respectively

Parameters **H** : Qobj / list

> System Hamiltonian given as a Qobj or nested list in string-based format.

psi0: Qobj

> Initial density matrix or state vector (ket).

tlist : array_like

> List of times for evaluating evolution

a_ops : list

> Nested list of Hermitian system operators that couple to the bath degrees of freedom, along with their associated spectra.

e_ops : list

> List of operators for which to evaluate expectation values.

c_ops : list

> List of system collapse operators, or nested list in string-based format.

args : dict (not implimented)

> Placeholder for future implementation, kept for API consistency.

use_secular : bool {True}

> Use secular approximation when evaluating bath-coupling terms.

sec_cutoff : float {0.1}

> Cutoff for secular approximation.

---

tol : float {qutip.setttings.atol}

Tolerance used for removing small values after basis transformation.

spectra_cb : list

DEPRECIATED. Do not use.

options : `qutip.solver.Options`

Options for the solver.

progress_bar : BaseProgressBar

Optional instance of BaseProgressBar, or a subclass thereof, for showing the progress of the simulation.

Returns result: `qutip.solver.Result`

An instance of the class `qutip.solver.Result`, which contains either an array of expectation values, for operators given in e_ops, or a list of states for the times specified by *tlist*.

bloch_redfield_tensor(*H*, *a_ops*, *spectra_cb=None*, *c_ops=[]*, *use_secular=True*, *sec_cutoff=0.1*)

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.

Note: This tensor generation requires a time-independent Hamiltonian.

Parameters **H** : `qutip.qobj`

System Hamiltonian.

a_ops : list of `qutip.qobj`

List of system operators that couple to the environment.

spectra_cb : list of callback functions

List of callback functions that evaluate the noise power spectrum at a given frequency.

c_ops : list of `qutip.qobj`

List of system collapse operators.

use_secular : bool

Flag (True of False) that indicates if the secular approximation should be used.

sec_cutoff : float {0.1}

Threshold for secular approximation.

Returns R, kets: `qutip.Qobj`, list of `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*, *progress_bar=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: `qutip.solver`

> An instance of the class `qutip.solver`, 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.solver.Options object>*, *floquet_basis=True*, *kmax=5*, *_safe_mode=True*)
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.solver`

options for the ODE solver.

k_max : int

The truncation of the number of sidebands (default 5).

Returns output : `qutip.solver`

An instance of the class `qutip.solver`, which contains either an *array* of expectation values for the times specified by *tlist*.

floquet_modes (*H*, *T*, *args=None*, *sort=False*, *U=None*)

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.

U : `qutip.qobj`

The propagator for the time-dependent Hamiltonian with period *T*. If U is *None* (default), it will be calculated from the Hamiltonian *H* using `qutip.propagator.propagator`.

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.solver.Result`

    An instance of the class `qutip.solver.Result`, which contains either an *array* of expectation values or an array of state vectors, for the times specified by *tlist*.

Stochastic Schrödinger Equation and Master Equation

This module contains functions for solving stochastic schrodinger and master equations. The API should not be considered stable, and is subject to change when we work more on optimizing this module for performance and features.

smesolve(*H*, *rho0*, *times*, *c_ops=[]*, *sc_ops=[]*, *e_ops=[]*, *_safe_mode=True*, ***kwargs*)

    Solve stochastic master equation. Dispatch to specific solvers depending on the value of the *solver* keyword argument.

    Parameters H : `qutip.Qobj`

        System Hamiltonian.

    rho0 : `qutip.Qobj`

        Initial density matrix or state vector (ket).

    times : *list* / *array*

        List of times for t. Must be uniformly spaced.

    c_ops : list of `qutip.Qobj`

        Deterministic collapse operator which will contribute with a standard Lindblad type of dissipation.

    sc_ops : list of `qutip.Qobj`

        List of stochastic collapse operators. Each stochastic collapse operator will give a deterministic and stochastic contribution to the eqaution of motion according to how the d1 and d2 functions are defined.

    e_ops : list of `qutip.Qobj` / callback function single

        single operator or list of operators for which to evaluate expectation values.

    kwargs : *dictionary*

Optional keyword arguments. See *qutip.stochastic.
StochasticSolverOptions*.

Returns output: `qutip.solver.SolverResult`

An instance of the class `qutip.solver.SolverResult`.

ssesolve (*H*, *psi0*, *times*, *sc_ops=[]*, *e_ops=[]*, *_safe_mode=True*, ***kwargs*)
Solve the stochastic Schrödinger equation. Dispatch to specific solvers depending on the value of the *solver* keyword argument.

Parameters H : *qutip.Qobj*

System Hamiltonian.

psi0 : *qutip.Qobj*

Initial state vector (ket).

times : *list / array*

List of times for t. Must be uniformly spaced.

sc_ops : list of *qutip.Qobj*

List of stochastic collapse operators. Each stochastic collapse operator will give a deterministic and stochastic contribution to the equation of motion according to how the d1 and d2 functions are defined.

e_ops : list of *qutip.Qobj*

Single operator or list of operators for which to evaluate expectation values.

kwargs : *dictionary*

Optional keyword arguments. See *qutip.stochastic.
StochasticSolverOptions*.

Returns output: `qutip.solver.SolverResult`

An instance of the class `qutip.solver.SolverResult`.

smepdpsolve (*H*, *rho0*, *times*, *c_ops*, *e_ops*, ***kwargs*)
A stochastic (piecwese deterministic process) PDP solver for density matrix evolution.

Parameters H : *qutip.Qobj*

System Hamiltonian.

rho0 : *qutip.Qobj*

Initial density matrix.

times : *list / array*

List of times for t. Must be uniformly spaced.

c_ops : list of *qutip.Qobj*

Deterministic collapse operator which will contribute with a standard Lindblad type of dissipation.

sc_ops : list of *qutip.Qobj*

List of stochastic collapse operators. Each stochastic collapse operator will give a deterministic and stochastic contribution to the eqaution of motion according to how the d1 and d2 functions are defined.

e_ops : list of *qutip.Qobj* / callback function single

single operator or list of operators for which to evaluate expectation values.

kwargs : *dictionary*

Optional keyword arguments. See *qutip.stochastic.*
StochasticSolverOptions.

Returns output: `qutip.solver.SolverResult`

An instance of the class `qutip.solver.SolverResult`.

ssepdpsolve(*H*, *psi0*, *times*, *c_ops*, *e_ops*, ***kwargs*)

A stochastic (piecewse deterministic process) PDP solver for wavefunction evolution. For most purposes, use *qutip.mcsolve* instead for quantum trajectory simulations.

Parameters H : *qutip.Qobj*

System Hamiltonian.

psi0 : *qutip.Qobj*

Initial state vector (ket).

times : *list / array*

List of times for t. Must be uniformly spaced.

c_ops : list of *qutip.Qobj*

Deterministic collapse operator which will contribute with a standard Lindblad type of dissipation.

e_ops : list of *qutip.Qobj* / callback function single

single operator or list of operators for which to evaluate expectation values.

kwargs : *dictionary*

Optional keyword arguments. See *qutip.stochastic.*
StochasticSolverOptions.

Returns output: `qutip.solver.SolverResult`

An instance of the class `qutip.solver.SolverResult`.

Correlation Functions

correlation(*H*, *state0*, *tlist*, *taulist*, *c_ops*, *a_op*, *b_op*, *solver='me'*, *reverse=False*, *args={}*, *options=<qutip.solver.Options object>*)

Calculate the two-operator two-time correlation function: $\langle A(t+\tau)B(t)\rangle$ along two time axes using the quantum regression theorem and the evolution solver indicated by the *solver* parameter.

Parameters H : Qobj

system Hamiltonian, may be time-dependent for solver choice of *me* or *mc*.

state0 : Qobj

Initial state density matrix $\rho(t_0)$ or state vector $\psi(t_0)$. If 'state0' is 'None', then the steady state will be used as the initial state. The 'steady-state' is only implemented for the *me* and *es* solvers.

tlist : array_like

list of times for t. tlist must be positive and contain the element *0*. When taking steady-steady correlations only one tlist value is necessary, i.e. when $t \to \infty$; here tlist is automatically set, ignoring user input.

taulist : array_like

list of times for τ. taulist must be positive and contain the element *0*.

c_ops : list

list of collapse operators, may be time-dependent for solver choice of *me* or *mc*.

a_op : Qobj

operator A.

b_op : Qobj

operator B.

reverse : *bool*

If *True*, calculate $\langle A(t)B(t+\tau)\rangle$ instead of $\langle A(t+\tau)B(t)\rangle$.

solver : str

choice of solver (*me* for master-equation, *mc* for Monte Carlo, and *es* for exponential series).

options : Options

solver options class. *ntraj* is taken as a two-element list because the *mc* correlator calls *mcsolve()* recursively; by default, *ntraj=[20, 100]*. *mc_corr_eps* prevents divide-by-zero errors in the *mc* correlator; by default, *mc_corr_eps=1e-10*.

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.

correlation_ss(*H*, *taulist*, *c_ops*, *a_op*, *b_op*, *solver='me'*, *reverse=False*, *args={}*, *options=<qutip.solver.Options object>*)

Calculate the two-operator two-time correlation function:

$$\lim_{t\to\infty} \langle A(t+\tau)B(t)\rangle$$

along one time axis (given steady-state initial conditions) using the quantum regression theorem and the evolution solver indicated by the *solver* parameter.

Parameters H : Qobj

system Hamiltonian.

taulist : array_like

list of times for τ. taulist must be positive and contain the element *0*.

c_ops : list

list of collapse operators.

a_op : Qobj

operator A.

b_op : Qobj

operator B.

reverse : *bool*

If *True*, calculate $\lim_{t\to\infty} \langle A(t)B(t+\tau)\rangle$ instead of $\lim_{t\to\infty} \langle A(t+\tau)B(t)\rangle$.

solver : str

choice of solver (*me* for master-equation and *es* for exponential series).

options : Options

solver options class. *ntraj* is taken as a two-element list because the *mc* correlator calls *mcsolve()* recursively; by default, *ntraj=[20, 100]*. *mc_corr_eps* prevents divide-by-zero errors in the *mc* correlator; by default, *mc_corr_eps=1e-10*.

>   **Returns corr_vec** : array

>> An array of correlation values for the times specified by *tlist*.

References

See, Gardiner, Quantum Noise, Section 5.2.

correlation_2op_1t (*H*, *state0*, *taulist*, *c_ops*, *a_op*, *b_op*, *solver='me'*, *reverse=False*, *args={}*, *options=<qutip.solver.Options object>*)

Calculate the two-operator two-time correlation function: $\langle A(t+\tau)B(t) \rangle$ along one time axis using the quantum regression theorem and the evolution solver indicated by the *solver* parameter.

>   **Parameters H** : Qobj

>> system Hamiltonian, may be time-dependent for solver choice of *me* or *mc*.

>   **state0** : Qobj

>> Initial state density matrix $\rho(t_0)$ or state vector $\psi(t_0)$. If 'state0' is 'None', then the steady state will be used as the initial state. The 'steady-state' is only implemented for the *me* and *es* solvers.

>   **taulist** : array_like

>> list of times for τ. taulist must be positive and contain the element *0*.

>   **c_ops** : list

>> list of collapse operators, may be time-dependent for solver choice of *me* or *mc*.

>   **a_op** : Qobj

>> operator A.

>   **b_op** : Qobj

>> operator B.

>   **reverse** : bool {False, True}

>> If *True*, calculate $\langle A(t)B(t+\tau) \rangle$ instead of $\langle A(t+\tau)B(t) \rangle$.

>   **solver** : str {'me', 'mc', 'es'}

>> choice of solver (*me* for master-equation, *mc* for Monte Carlo, and *es* for exponential series).

>   **options** : Options

>> Solver options class. *ntraj* is taken as a two-element list because the *mc* correlator calls *mcsolve()* recursively; by default, *ntraj=[20, 100]*. *mc_corr_eps* prevents divide-by-zero errors in the *mc* correlator; by default, *mc_corr_eps=1e-10*.

>   **Returns corr_vec** : ndarray

>> An array of correlation values for the times specified by *tlist*.

References

See, Gardiner, Quantum Noise, Section 5.2.

correlation_2op_2t (*H*, *state0*, *tlist*, *taulist*, *c_ops*, *a_op*, *b_op*, *solver='me'*, *reverse=False*, *args={}, options=<qutip.solver.Options object>*)

Calculate the two-operator two-time correlation function: $\langle A(t+\tau)B(t)\rangle$ along two time axes using the quantum regression theorem and the evolution solver indicated by the *solver* parameter.

> **Parameters** **H** : Qobj
>
>> system Hamiltonian, may be time-dependent for solver choice of *me* or *mc*.
>
> **state0** : Qobj
>
>> Initial state density matrix ρ_0 or state vector ψ_0. If 'state0' is 'None', then the steady state will be used as the initial state. The 'steady-state' is only implemented for the *me* and *es* solvers.
>
> **tlist** : array_like
>
>> list of times for t. tlist must be positive and contain the element *0*. When taking steady-steady correlations only one tlist value is necessary, i.e. when $t \rightarrow \infty$; here tlist is automatically set, ignoring user input.
>
> **taulist** : array_like
>
>> list of times for τ. taulist must be positive and contain the element *0*.
>
> **c_ops** : list
>
>> list of collapse operators, may be time-dependent for solver choice of *me* or *mc*.
>
> **a_op** : Qobj
>
>> operator A.
>
> **b_op** : Qobj
>
>> operator B.
>
> **reverse** : bool {False, True}
>
>> If *True*, calculate $\langle A(t)B(t+\tau)\rangle$ instead of $\langle A(t+\tau)B(t)\rangle$.
>
> **solver** : str
>
>> choice of solver (*me* for master-equation, *mc* for Monte Carlo, and *es* for exponential series).
>
> **options** : Options
>
>> solver options class. *ntraj* is taken as a two-element list because the *mc* correlator calls *mcsolve()* recursively; by default, *ntraj=[20, 100]*. *mc_corr_eps* prevents divide-by-zero errors in the *mc* correlator; by default, *mc_corr_eps=1e-10*.
>
> **Returns** **corr_mat** : ndarray
>
>> 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.

correlation_3op_1t (*H*, *state0*, *taulist*, *c_ops*, *a_op*, *b_op*, *c_op*, *solver='me'*, *args={}, options=<qutip.solver.Options object>*)

Calculate the three-operator two-time correlation function: $\langle A(t)B(t+\tau)C(t)\rangle$ along one time axis using the quantum regression theorem and the evolution solver indicated by the *solver* parameter.

Note: it is not possibly to calculate a physically meaningful correlation of this form where $\tau < 0$.

> **Parameters** **H** : Qobj

system Hamiltonian, may be time-dependent for solver choice of *me* or *mc*.

rho0 : Qobj

Initial state density matrix $\rho(t_0)$ or state vector $\psi(t_0)$. If 'state0' is 'None', then the steady state will be used as the initial state. The 'steady-state' is only implemented for the *me* and *es* solvers.

taulist : array_like

list of times for τ. taulist must be positive and contain the element *0*.

c_ops : list

list of collapse operators, may be time-dependent for solver choice of *me* or *mc*.

a_op : Qobj

operator A.

b_op : Qobj

operator B.

c_op : Qobj

operator C.

solver : str

choice of solver (*me* for master-equation, *mc* for Monte Carlo, and *es* for exponential series).

options : Options

solver options class. *ntraj* is taken as a two-element list because the *mc* correlator calls *mcsolve()* recursively; by default, *ntraj=[20, 100]*. *mc_corr_eps* prevents divide-by-zero errors in the *mc* correlator; by default, *mc_corr_eps=1e-10*.

Returns corr_vec : array

An array of correlation values for the times specified by *taulist*

References

See, Gardiner, Quantum Noise, Section 5.2.

correlation_3op_2t (*H*, *state0*, *tlist*, *taulist*, *c_ops*, *a_op*, *b_op*, *c_op*, *solver='me'*, *args={}*, *options=<qutip.solver.Options object>*)
Calculate the three-operator two-time correlation function: $\langle A(t)B(t + \tau)C(t)\rangle$ along two time axes using the quantum regression theorem and the evolution solver indicated by the *solver* parameter.

Note: it is not possibly to calculate a physically meaningful correlation of this form where $\tau < 0$.

Parameters H : Qobj

system Hamiltonian, may be time-dependent for solver choice of *me* or *mc*.

rho0 : Qobj

Initial state density matrix ρ_0 or state vector ψ_0. If 'state0' is 'None', then the steady state will be used as the initial state. The 'steady-state' is only implemented for the *me* and *es* solvers.

tlist : array_like

list of times for t. tlist must be positive and contain the element *0*. When taking steady-steady correlations only one tlist value is necessary, i.e. when $t \to \infty$; here tlist is automatically set, ignoring user input.

taulist : array_like

list of times for τ. taulist must be positive and contain the element *0*.

c_ops : list

list of collapse operators, may be time-dependent for solver choice of *me* or *mc*.

a_op : Qobj

operator A.

b_op : Qobj

operator B.

c_op : Qobj

operator C.

solver : str

choice of solver (*me* for master-equation, *mc* for Monte Carlo, and *es* for exponential series).

options : Options

solver options class. *ntraj* is taken as a two-element list because the *mc* correlator calls *mcsolve()* recursively; by default, *ntraj=[20, 100]*. *mc_corr_eps* prevents divide-by-zero errors in the *mc* correlator; by default, *mc_corr_eps=1e-10*.

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.

correlation_4op_1t (*H*, *state0*, *taulist*, *c_ops*, *a_op*, *b_op*, *c_op*, *d_op*, *solver='me'*, *args={}*, *options=<qutip.solver.Options object>*)
Calculate the four-operator two-time correlation function: $\langle A(t)B(t+\tau)C(t+\tau)D(t)\rangle$ along one time axis using the quantum regression theorem and the evolution solver indicated by the *solver* parameter.

Note: it is not possibly to calculate a physically meaningful correlation of this form where $\tau < 0$.

Parameters H : Qobj

system Hamiltonian, may be time-dependent for solver choice of *me* or *mc*.

rho0 : Qobj

Initial state density matrix $\rho(t_0)$ or state vector $\psi(t_0)$. If 'state0' is 'None', then the steady state will be used as the initial state. The 'steady-state' is only implemented for the *me* and *es* solvers.

taulist : array_like

list of times for τ. taulist must be positive and contain the element *0*.

c_ops : list

list of collapse operators, may be time-dependent for solver choice of *me* or *mc*.

a_op : Qobj

operator A.

b_op : Qobj

operator B.

c_op : Qobj

operator C.

d_op : Qobj

operator D.

solver : str

choice of solver (*me* for master-equation, *mc* for Monte Carlo, and *es* for exponential series).

options : Options

solver options class. *ntraj* is taken as a two-element list because the *mc* correlator calls *mcsolve()* recursively; by default, *ntraj=[20, 100]*. *mc_corr_eps* prevents divide-by-zero errors in the *mc* correlator; by default, *mc_corr_eps=1e-10*.

Returns corr_vec : array

An array of correlation values for the times specified by *taulist*.

References

See, Gardiner, Quantum Noise, Section 5.2.

Note: Deprecated in QuTiP 3.1 Use correlation_3op_1t() instead.

correlation_4op_2t (*H, state0, tlist, taulist, c_ops, a_op, b_op, c_op, d_op, solver='me', args={}, options=<qutip.solver.Options object>*)

Calculate the four-operator two-time correlation function: $\langle A(t)B(t+\tau)C(t+\tau)D(t)\rangle$ along two time axes using the quantum regression theorem and the evolution solver indicated by the *solver* parameter.

Note: it is not possibly to calculate a physically meaningful correlation of this form where $\tau < 0$.

Parameters H : Qobj

system Hamiltonian, may be time-dependent for solver choice of *me* or *mc*.

rho0 : Qobj

Initial state density matrix ρ_0 or state vector ψ_0. If 'state0' is 'None', then the steady state will be used as the initial state. The 'steady-state' is only implemented for the *me* and *es* solvers.

tlist : array_like

list of times for t. tlist must be positive and contain the element *0*. When taking steady-steady correlations only one tlist value is necessary, i.e. when $t \to \infty$; here tlist is automatically set, ignoring user input.

taulist : array_like

list of times for τ. taulist must be positive and contain the element *0*.

c_ops : list

list of collapse operators, may be time-dependent for solver choice of *me* or *mc*.

a_op : Qobj

operator A.

b_op : Qobj

operator B.

c_op : Qobj

> operator C.

d_op : Qobj

> operator D.

solver : str

> choice of solver (*me* for master-equation, *mc* for Monte Carlo, and *es* for exponential series).

options : Options

> solver options class. *ntraj* is taken as a two-element list because the *mc* correlator calls *mcsolve()* recursively; by default, *ntraj=[20, 100]*. *mc_corr_eps* prevents divide-by-zero errors in the *mc* correlator; by default, *mc_corr_eps=1e-10*.

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.

spectrum(*H, wlist, c_ops, a_op, b_op, solver='es', use_pinv=False*)
Calculate the spectrum of the correlation function $\lim_{t \to \infty} \langle A(t+\tau)B(t) \rangle$, i.e., the Fourier transform of the correlation function:

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

using the solver indicated by the *solver* parameter. Note: this spectrum is only defined for stationary statistics (uses steady state rho0)

Parameters H : qutip.qobj

> system Hamiltonian.

wlist : array_like

> list of frequencies for ω.

c_ops : list

> list of collapse operators.

a_op : Qobj

> operator A.

b_op : Qobj

> operator B.

solver : str

> choice of solver (*es* for exponential series and *pi* for psuedo-inverse).

use_pinv : bool

> For use with the *pi* solver: if *True* use numpy's pinv method, otherwise use a generic solver.

Returns spectrum : array

> An array with spectrum $S(\omega)$ for the frequencies specified in *wlist*.

spectrum_ss (*H*, *wlist*, *c_ops*, *a_op*, *b_op*)

Calculate the spectrum of the correlation function $\lim_{t \to \infty} \langle A(t + \tau)B(t) \rangle$, i.e., the Fourier transform of the correlation function:

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

using an eseries based solver Note: this spectrum is only defined for stationary statistics (uses steady state rho0).

Parameters **H** : `qutip.qobj`

system Hamiltonian.

wlist : array_like

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.

use_pinv : *bool*

If *True* use numpy's *pinv* method, otherwise use a generic solver.

Returns **spectrum** : array

An array with spectrum $S(\omega)$ for the frequencies specified in *wlist*.

spectrum_pi (*H*, *wlist*, *c_ops*, *a_op*, *b_op*, *use_pinv=False*)

Calculate the spectrum of the correlation function $\lim_{t \to \infty} \langle A(t + \tau)B(t) \rangle$, i.e., the Fourier transform of the correlation function:

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

using a psuedo-inverse method. Note: this spectrum is only defined for stationary statistics (uses steady state rho0)

Parameters **H** : `qutip.qobj`

system Hamiltonian.

wlist : array_like

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.

use_pinv : *bool*

If *True* use numpy's pinv method, otherwise use a generic solver.

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** : array_like
>
>> list/array of times t which the correlation function is given.
>
> **y** : array_like
>
>> 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*, *state0*, *taulist*, *c_ops*, *a_op*, *solver='me'*, *args={}*, *options=<qutip.solver.Options object>*)

Calculate the normalized first-order quantum coherence function:

$$g^{(1)}(\tau) = \frac{\langle A^\dagger(\tau)A(0)\rangle}{\sqrt{\langle A^\dagger(\tau)A(\tau)\rangle\langle A^\dagger(0)A(0)\rangle}}$$

using the quantum regression theorem and the evolution solver indicated by the *solver* parameter.

> **Parameters** **H** : Qobj
>
>> system Hamiltonian, may be time-dependent for solver choice of *me* or *mc*.
>
> **state0** : Qobj
>
>> Initial state density matrix $\rho(t_0)$ or state vector $\psi(t_0)$. If 'state0' is 'None', then the steady state will be used as the initial state. The 'steady-state' is only implemented for the *me* and *es* solvers.
>
> **taulist** : array_like
>
>> list of times for τ. taulist must be positive and contain the element *0*.
>
> **c_ops** : list
>
>> list of collapse operators, may be time-dependent for solver choice of *me* or *mc*.
>
> **a_op** : Qobj
>
>> operator A.
>
> **solver** : str
>
>> choice of solver (*me* for master-equation and *es* for exponential series).
>
> **options** : Options
>
>> solver options class. *ntraj* is taken as a two-element list because the *mc* correlator calls *mcsolve()* recursively; by default, *ntraj=[20, 100]*. *mc_corr_eps* prevents divide-by-zero errors in the *mc* correlator; by default, *mc_corr_eps=1e-10*.
>
> **Returns** **g1, G1** : tuple
>
>> The normalized and unnormalized second-order coherence function.

coherence_function_g2 (*H*, *state0*, *taulist*, *c_ops*, *a_op*, *solver='me'*, *args={}*, *options=<qutip.solver.Options object>*)

Calculate the normalized second-order quantum coherence function:

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

using the quantum regression theorem and the evolution solver indicated by the *solver* parameter.

Parameters **H** : Qobj

>> system Hamiltonian, may be time-dependent for solver choice of *me* or *mc*.

> **state0** : Qobj

>> Initial state density matrix $\rho(t_0)$ or state vector $\psi(t_0)$. If 'state0' is 'None', then the steady state will be used as the initial state. The 'steady-state' is only implemented for the *me* and *es* solvers.

> **taulist** : array_like

>> list of times for τ. taulist must be positive and contain the element *0*.

> **c_ops** : list

>> list of collapse operators, may be time-dependent for solver choice of *me* or *mc*.

> **a_op** : Qobj

>> operator A.

> **solver** : str

>> choice of solver (*me* for master-equation and *es* for exponential series).

> **options** : Options

>> solver options class. *ntraj* is taken as a two-element list because the *mc* correlator calls *mcsolve()* recursively; by default, *ntraj=[20, 100]*. *mc_corr_eps* prevents divide-by-zero errors in the *mc* correlator; by default, *mc_corr_eps=1e-10*.

Returns **g2, G2** : tuple

>> The normalized and unnormalized second-order coherence function.

Steady-state Solvers

Module contains functions for solving for the steady state density matrix of open quantum systems defined by a Liouvillian or Hamiltonian and a list of collapse operators.

steadystate(*A*, *c_op_list=[]*, ***kwargs*)

> Calculates the steady state for quantum evolution subject to the supplied Hamiltonian or Liouvillian operator and (if given a Hamiltonian) a list of collapse operators.

> If the user passes a Hamiltonian then it, along with the list of collapse operators, will be converted into a Liouvillian operator in Lindblad form.

> **Parameters** **A** : qobj

>> A Hamiltonian or Liouvillian operator.

> **c_op_list** : list

>> A list of collapse operators.

> **method** : str {'direct', 'eigen', 'iterative-gmres',

>> 'iterative-lgmres', 'iterative-bicgstab', 'svd', 'power', 'power-gmres', 'power-lgmres', 'power-bicgstab'}

>> Method for solving the underlying linear equation. Direct LU solver 'direct' (default), sparse eigenvalue problem 'eigen', iterative GMRES method 'iterative-gmres', iterative LGMRES method 'iterative-lgmres', iterative BICGSTAB method 'iterative-bicgstab', SVD 'svd' (dense), or inverse-power method 'power'. The iterative power methods 'power-gmres', 'power-lgmres', 'power-bicgstab' use the same solvers as their direct counterparts.

> **return_info** : bool, optional, default = False

Return a dictionary of solver-specific infomation about the solution and how it was obtained.

sparse : bool, optional, default = True

Solve for the steady state using sparse algorithms. If set to False, the underlying Liouvillian operator will be converted into a dense matrix. Use only for 'smaller' systems.

use_rcm : bool, optional, default = False

Use reverse Cuthill-Mckee reordering to minimize fill-in in the LU factorization of the Liouvillian.

use_wbm : bool, optional, default = False

Use Weighted Bipartite Matching reordering to make the Liouvillian diagonally dominant. This is useful for iterative preconditioners only, and is set to `True` by default when finding a preconditioner.

weight : float, optional

Sets the size of the elements used for adding the unity trace condition to the linear solvers. This is set to the average abs value of the Liouvillian elements if not specified by the user.

x0 : ndarray, optional

ITERATIVE ONLY. Initial guess for solution vector.

maxiter : int, optional, default=1000

ITERATIVE ONLY. Maximum number of iterations to perform.

tol : float, optional, default=1e-12

ITERATIVE ONLY. Tolerance used for terminating solver.

permc_spec : str, optional, default='COLAMD'

ITERATIVE ONLY. Column ordering used internally by superLU for the 'direct' LU decomposition method. Options include 'COLAMD' and 'NATURAL'. If using RCM then this is set to 'NATURAL' automatically unless explicitly specified.

use_precond : bool optional, default = False

ITERATIVE ONLY. Use an incomplete sparse LU decomposition as a preconditioner for the 'iterative' GMRES and BICG solvers. Speeds up convergence time by orders of magnitude in many cases.

M : {sparse matrix, dense matrix, LinearOperator}, optional

ITERATIVE ONLY. Preconditioner for A. The preconditioner should approximate the inverse of A. Effective preconditioning can dramatically improve the rate of convergence for iterative methods. If no preconditioner is given and `use_precond = True`, then one is generated automatically.

fill_factor : float, optional, default = 100

ITERATIVE ONLY. Specifies the fill ratio upper bound (>=1) of the iLU preconditioner. Lower values save memory at the cost of longer execution times and a possible singular factorization.

drop_tol : float, optional, default = 1e-4

ITERATIVE ONLY. Sets the threshold for the magnitude of preconditioner elements that should be dropped. Can be reduced for a courser factorization at the cost of an increased number of iterations, and a possible singular factorization.

diag_pivot_thresh : float, optional, default = None

ITERATIVE ONLY. Sets the threshold between [0,1] for which diagonal elements are considered acceptable pivot points when using a preconditioner. A value of zero forces the pivot to be the diagonal element.

ILU_MILU : str, optional, default = 'smilu_2'

ITERATIVE ONLY. Selects the incomplete LU decomposition method algoithm used in creating the preconditoner. Should only be used by advanced users.

Returns dm : qobj

Steady state density matrix.

info : dict, optional

Dictionary containing solver-specific information about the solution.

Notes

The SVD method works only for dense operators (i.e. small systems).

build_preconditioner(*A*, *c_op_list=[]*, ***kwargs*)
Constructs a iLU preconditioner necessary for solving for the steady state density matrix using the iterative linear solvers in the 'steadystate' function.

Parameters A : qobj

A Hamiltonian or Liouvillian operator.

c_op_list : list

A list of collapse operators.

return_info : bool, optional, default = False

Return a dictionary of solver-specific infomation about the solution and how it was obtained.

use_rcm : bool, optional, default = False

Use reverse Cuthill-Mckee reordering to minimize fill-in in the LU factorization of the Liouvillian.

use_wbm : bool, optional, default = False

Use Weighted Bipartite Matching reordering to make the Liouvillian diagonally dominant. This is useful for iterative preconditioners only, and is set to `True` by default when finding a preconditioner.

weight : float, optional

Sets the size of the elements used for adding the unity trace condition to the linear solvers. This is set to the average abs value of the Liouvillian elements if not specified by the user.

method : str, default = 'iterative'

Tells the preconditioner what type of Liouvillian to build for iLU factorization. For direct iterative methods use 'iterative'. For power iterative methods use 'power'.

permc_spec : str, optional, default='COLAMD'

Column ordering used internally by superLU for the 'direct' LU decomposition method. Options include 'COLAMD' and 'NATURAL'. If using RCM then this is set to 'NATURAL' automatically unless explicitly specified.

fill_factor : float, optional, default = 100

Specifies the fill ratio upper bound (>=1) of the iLU preconditioner. Lower values save memory at the cost of longer execution times and a possible singular factorization.

drop_tol : float, optional, default = 1e-4

Sets the threshold for the magnitude of preconditioner elements that should be dropped. Can be reduced for a courser factorization at the cost of an increased number of iterations, and a possible singular factorization.

diag_pivot_thresh : float, optional, default = None

Sets the threshold between [0,1] for which diagonal elements are considered acceptable pivot points when using a preconditioner. A value of zero forces the pivot to be the diagonal element.

ILU_MILU : str, optional, default = 'smilu_2'

Selects the incomplete LU decomposition method algoithm used in creating the preconditoner. Should only be used by advanced users.

Returns **lu** : object

Returns a SuperLU object representing iLU preconditioner.

info : dict, optional

Dictionary containing solver-specific information.

Propagators

propagator(*H*, *t*, *c_op_list=[]*, *args={}*, *options=None*, *unitary_mode='batch'*, *parallel=False*, *progress_bar=None*, ***kwargs*)

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 ρ_{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.

args : list/array/dictionary

Parameters to callback functions for time-dependent Hamiltonians and collapse operators.

options : `qutip.Options`

with options for the ODE solver.

unitary_mode = str ('batch', 'single')

Solve all basis vectors simulaneously ('batch') or individually ('single').

parallel : bool {False, True}

Run the propagator in parallel mode. This will override the unitary_mode settings if set to True.

progress_bar: BaseProgressBar

Optional instance of BaseProgressBar, or a subclass thereof, for showing the progress of the simulation. By default no progress bar is used, and if set to True a TextProgressBar will be used.

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.

Time-dependent problems

rhs_generate(*H, c_ops, args={}, options=<qutip.solver.Options object>, name=None, cleanup=True*)

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 : Options

Instance of ODE solver options.

name: str

Name of generated RHS

cleanup: bool

Whether the generated cython file should be automatically removed or not.

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 config module.

4.2.4 Visualization

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].

spin_q_function (*rho*, *theta*, *phi*)

Husimi Q-function for spins.

Parameters state : qobj

A state vector or density matrix for a spin-j quantum system.

theta : array_like

theta-coordinates at which to calculate the Q function.

phi : array_like

phi-coordinates at which to calculate the Q function.

Returns Q, THETA, PHI : 2d-array

Values representing the spin Q function at the values specified by THETA and PHI.

spin_wigner (*rho*, *theta*, *phi*)

Wigner function for spins on the Bloch sphere.

Parameters state : qobj

A state vector or density matrix for a spin-j quantum system.

theta : array_like

theta-coordinates at which to calculate the Q function.

phi : array_like

phi-coordinates at which to calculate the Q function.

Returns W, THETA, PHI : 2d-array

Values representing the spin Wigner function at the values specified by THETA and PHI.

Notes

Experimental.

wigner (*psi*, *xvec*, *yvec*, *method='clenshaw'*, *g=1.4142135623730951*, *sparse=False*, *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. Does not apply to the 'fft' method.
>
> > **g** : float
> >
> > > Scaling factor for *a = 0.5 * g * (x + iy)*, default *g = sqrt(2)*.
> >
> > **method** : string {'clenshaw', 'iterative', 'laguerre', 'fft'}
> >
> > > Select method 'clenshaw' 'iterative', 'laguerre', or 'fft', where 'clenshaw' and 'iterative' use an 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 'fft' method evaluates the Fourier transform of the density matrix. 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). The 'clenshaw' method is the preferred method for dealing with density matrices that have a large number of excitations (>~50). 'clenshaw' is a fast and numerically stable method.
> >
> > **sparse** : bool {False, True}
> >
> > > Tells the default solver whether or not to keep the input density matrix in sparse format. As the dimensions of the density matrix grow, setthing this flag can result in increased performance.
> >
> > **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].
>
> > **yvex** : array
> >
> > > FFT ONLY. Returns the y-coordinate values calculated via the Fourier transform.

Notes

The 'fft' method accepts only an xvec input for the x-coordinate. The y-coordinates are calculated internally.

References

Ulf Leonhardt, Measuring the Quantum State of Light, (Cambridge University Press, 1997)

Graphs and Visualization

Functions for visualizing results of quantum dynamics simulations, visualizations of quantum states and processes.

hinton (*rho*, *xlabels=None*, *ylabels=None*, *title=None*, *ax=None*, *cmap=None*, *label_top=True*)
> Draws a Hinton diagram for visualizing a density matrix or superoperator.
>
> > **Parameters rho** : qobj
> >
> > > Input density matrix or superoperator.
> >
> > **xlabels** : list of strings or False
> >
> > > list of x labels
> >
> > **ylabels** : list of strings or False

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.

cmap : a matplotlib colormap instance

Color map to use when plotting.

label_top : bool

If True, x-axis labels will be placed on top, otherwise they will appear below the plot.

Returns fig, ax : tuple

A tuple of the matplotlib figure and axes instances used to produce the figure.

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 fig, ax : tuple

A tuple of the matplotlib figure and axes instances used to produce the figure.

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*, *threshold=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.

threshold: float (None)

Threshold for when bars of smaller height should be transparent. If not set, all bars are colored according to the color map.

Returns fig, ax : tuple

A tuple of the matplotlib figure and axes instances used to produce the figure.

Raises ValueError

Input argument is not valid.

plot_energy_levels(*H_list*, *N=0*, *labels=None*, *show_ylabels=False*, *figsize=(8, 12)*, *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

show_ylabels [Bool (default False)] Show y labels to the left of energy levels of the initial Hamiltonian.

N [int] The number of energy levels to plot

figsize [tuple (int,int)] The size of the figure (width, height).

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.

Returns fig, ax : tuple

A tuple of the matplotlib figure and axes instances used to produce the figure.

Raises ValueError

Input argument is not valid.

plot_fock_distribution(*rho*, *offset=0*, *fig=None*, *ax=None*, *figsize=(8, 6)*, *title=None*, *unit_y_range=True*)
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 fig, ax : tuple

A tuple of the matplotlib figure and axes instances used to produce the figure.

plot_wigner_fock_distribution(*rho*, *fig=None*, *axes=None*, *figsize=(8, 4)*, *cmap=None*, *alpha_max=7.5*, *colorbar=False*, *method='iterative'*, *projection='2d'*)

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.

axes : a list of two matplotlib axes instances

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.

method : string {'iterative', 'laguerre', 'fft'}

The method used for calculating the wigner function. See the documentation for qutip.wigner for details.

projection: string {'2d', '3d'}

Specify whether the Wigner function is to be plotted as a contour graph ('2d') or surface plot ('3d').

Returns fig, ax : tuple

A tuple of the matplotlib figure and axes instances used to produce the figure.

plot_wigner(*rho*, *fig=None*, *ax=None*, *figsize=(8*, *4)*, *cmap=None*, *alpha_max=7.5*, *colorbar=False*,
method='iterative', *projection='2d'*)

Plot the 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.
>
> **method** : string {'iterative', 'laguerre', 'fft'}
>
> > The method used for calculating the wigner function. See the documentation for
> > qutip.wigner for details.
>
> **projection: string {'2d', '3d'}**
>
> > Specify whether the Wigner function is to be plotted as a contour graph ('2d') or
> > surface plot ('3d').
>
> **Returns fig, ax** : tuple
>
> > A tuple of the matplotlib figure and axes instances used to produce the figure.

sphereplot(*theta*, *phi*, *values*, *fig=None*, *ax=None*, *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
>
> **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.
>
> **save** : bool {False , True}
>
> > Whether to save the figure or not

Returns fig, ax : tuple

A tuple of the matplotlib figure and axes instances used to produce the figure.

plot_schmidt (*ket*, *splitting=None*, *labels_iteration=(3, 2)*, *theme='light'*, *fig=None*, *ax=None*, *figsize=(6, 6)*)

Plotting scheme related to Schmidt decomposition. Converts a state into a matrix (A_ij -> A_i^j), where rows are first particles and columns - last.

See also: plot_qubism with how='before_after' for a similar plot.

Parameters ket : Qobj

Pure state for plotting.

splitting : int

Plot for a number of first particles versus the rest. If not given, it is (number of particles + 1) // 2.

theme : 'light' (default) or 'dark'

Set coloring theme for mapping complex values into colors. See: complex_array_to_rgb.

labels_iteration : int or pair of ints (default (3,2))

Number of particles to be shown as tick labels, for first (vertical) and last (horizontal) particles, respectively.

fig : a matplotlib figure instance

The figure canvas on which the plot will be drawn.

ax : a matplotlib axis instance

The axis 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).

Returns fig, ax : tuple

A tuple of the matplotlib figure and axes instances used to produce the figure.

plot_qubism (*ket*, *theme='light'*, *how='pairs'*, *grid_iteration=1*, *legend_iteration=0*, *fig=None*, *ax=None*, *figsize=(6, 6)*)

Qubism plot for pure states of many qudits. Works best for spin chains, especially with even number of particles of the same dimension. Allows to see entanglement between first 2*k particles and the rest.

More information:

J. Rodriguez-Laguna, P. Migdal, M. Ibanez Berganza, M. Lewenstein, G. Sierra, "Qubism: self-similar visualization of many-body wavefunctions", New J. Phys. 14 053028 (2012), arXiv:1112.3560, http://dx.doi.org/10.1088/1367-2630/14/5/053028 (open access)

Parameters ket : Qobj

Pure state for plotting.

theme : 'light' (default) or 'dark'

Set coloring theme for mapping complex values into colors. See: complex_array_to_rgb.

how : 'pairs' (default), 'pairs_skewed' or 'before_after'

Type of Qubism plotting. Options:

'pairs' - typical coordinates, 'pairs_skewed' - for ferromagnetic/antriferromagnetic plots, 'before_after' - related to Schmidt plot (see also: plot_schmidt).

grid_iteration : int (default 1)

Helper lines to be drawn on plot. Show tiles for 2*grid_iteration particles vs all others.

legend_iteration : int (default 0) or 'grid_iteration' or 'all'

Show labels for first 2*legend_iteration particles. Option 'grid_iteration' sets the same number of particles

as for grid_iteration.

Option 'all' makes label for all particles. Typically it should be 0, 1, 2 or perhaps 3.

fig : a matplotlib figure instance

The figure canvas on which the plot will be drawn.

ax : a matplotlib axis instance

The axis 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).

Returns fig, ax : tuple

A tuple of the matplotlib figure and axes instances used to produce the figure.

---

plot_expectation_values (*results*, *ylabels=[]*, *title=None*, *show_legend=False*, *fig=None*, *axes=None*, *figsize=(8, 4)*)

Visualize the results (expectation values) for an evolution solver. *results* is assumed to be an instance of Result, or a list of Result instances.

Parameters results : (list of) `qutip.solver.Result`

List of results objects returned by any of the QuTiP evolution solvers.

ylabels : list of strings

The y-axis labels. List should be of the same length as *results*.

title : string

The title of the figure.

show_legend : bool

Whether or not to show the legend.

fig : a matplotlib Figure instance

The Figure canvas in which the plot will be drawn.

axes : 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).

Returns fig, ax : tuple

A tuple of the matplotlib figure and axes instances used to produce the figure.

plot_spin_distribution_2d(*P*, *THETA*, *PHI*, *fig=None*, *ax=None*, *figsize=(8, 8)*)

Plot a spin distribution function (given as meshgrid data) with a 2D projection where the surface of the unit sphere is mapped on the unit disk.

> **Parameters P** : matrix
>
> > Distribution values as a meshgrid matrix.
> >
> > **THETA** : matrix
> >
> > Meshgrid matrix for the theta coordinate.
> >
> > **PHI** : matrix
> >
> > Meshgrid matrix for the phi coordinate.
> >
> > **fig** : a matplotlib figure instance
> >
> > The figure canvas on which the plot will be drawn.
> >
> > **ax** : a matplotlib axis instance
> >
> > The axis 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).
>
> **Returns fig, ax** : tuple
>
> > A tuple of the matplotlib figure and axes instances used to produce the figure.

plot_spin_distribution_3d(*P*, *THETA*, *PHI*, *fig=None*, *ax=None*, *figsize=(8, 6)*)

Plots a matrix of values on a sphere

> **Parameters P** : matrix
>
> > Distribution values as a meshgrid matrix.
> >
> > **THETA** : matrix
> >
> > Meshgrid matrix for the theta coordinate.
> >
> > **PHI** : matrix
> >
> > Meshgrid matrix for the phi coordinate.
> >
> > **fig** : a matplotlib figure instance
> >
> > The figure canvas on which the plot will be drawn.
> >
> > **ax** : a matplotlib axis instance
> >
> > The axis 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).
>
> **Returns fig, ax** : tuple
>
> > A tuple of the matplotlib figure and axes instances used to produce the figure.

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

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** **fig, ax** : tuple
>
>> A tuple of the matplotlib figure and axes instances used to produce the figure.

qpt_plot_combined (*chi*, *lbls_list*, *title=None*, *fig=None*, *ax=None*, *figsize=(8, 6)*, *threshold=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).

threshold: float (None)

Threshold for when bars of smaller height should be transparent. If not set, all bars are colored according to the color map.

Returns fig, ax : tuple

A tuple of the matplotlib figure and axes instances used to produce the figure.

4.2.5 Quantum Information Processing

Gates

rx (*phi*, *N=None*, *target=0*)
Single-qubit rotation for operator sigmax with angle phi.

> **Returns result** : qobj
>
> Quantum object for operator describing the rotation.

ry (*phi*, *N=None*, *target=0*)
Single-qubit rotation for operator sigmay with angle phi.

> **Returns result** : qobj
>
> Quantum object for operator describing the rotation.

rz (*phi*, *N=None*, *target=0*)
Single-qubit rotation for operator sigmaz with angle phi.

> **Returns result** : qobj
>
> Quantum object for operator describing the rotation.

sqrtnot (*N=None*, *target=0*)
Single-qubit square root NOT gate.

> **Returns result** : qobj
>
> Quantum object for operator describing the square root NOT gate.

snot (*N=None*, *target=0*)
Quantum object representing the SNOT (Hadamard) gate.

> **Returns snot_gate** : qobj
>
> Quantum object representation of SNOT 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]]
```

phasegate (*theta*, *N=None*, *target=0*)

  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]]
```

cphase (*theta*, *N=2*, *control=0*, *target=1*)

  Returns quantum object representing the controlled phase shift gate.

    Parameters theta : float

      Phase rotation angle.

    N : integer

      The number of qubits in the target space.

    control : integer

      The index of the control qubit.

    target : integer

      The index of the target qubit.

    Returns U : qobj

      Quantum object representation of controlled phase gate.

cnot (*N=None*, *control=0*, *target=1*)

  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]]
```

csign (*N=None*, *control=0*, *target=1*)

    Quantum object representing the CSIGN gate.

        Returns csign_gate : qobj

            Quantum object representation of CSIGN gate

Examples

```
>>> csign()
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  1.+0.j  0.+0.j]
     [ 0.+0.j  0.+0.j  0.+0.j  -1.+0.j]]
```

berkeley (*N=None*, *targets=[0, 1]*)

    Quantum object representing the Berkeley gate.

        Returns berkeley_gate : qobj

            Quantum object representation of Berkeley gate

Examples

```
>>> berkeley()
Quantum object: dims = [[2, 2], [2, 2]], shape = [4, 4], type = oper, isHerm =␣
↪True
Qobj data =
    [[ cos(pi/8).+0.j   0.+0.j          0.+0.j          0.+sin(pi/8).j]
     [ 0.+0.j          cos(3pi/8).+0.j  0.+sin(3pi/8).j  0.+0.j]
     [ 0.+0.j          0.+sin(3pi/8).j  cos(3pi/8).+0.j  0.+0.j]
     [ 0.+sin(pi/8).j  0.+0.j          0.+0.j          cos(pi/8).+0.j]]
```

swapalpha (*alpha, N=None, targets=[0, 1]*)

    Quantum object representing the SWAPalpha gate.

        Returns swapalpha_gate : qobj

            Quantum object representation of SWAPalpha gate

Examples

```
>>> swapalpha(alpha)
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  0.5*(1 + exp(j*pi*alpha)  0.5*(1 - exp(j*pi*alpha)  0.+0.j]
 [ 0.+0.j  0.5*(1 - exp(j*pi*alpha)  0.5*(1 + exp(j*pi*alpha)  0.+0.j]
 [ 0.+0.j  0.+0.j                   0.+0.j                   1.+0.j]]
```

swap (*N=None*, *targets=[0, 1]*)

    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]]
```

iswap (*N=None, targets=[0, 1]*)

Quantum object representing the iSWAP gate.

> **Returns iswap_gate** : qobj
>
> > Quantum object representation of iSWAP gate

Examples

```
>>> iswap()
Quantum object: dims = [[2, 2], [2, 2]], shape = [4, 4], type = oper, isHerm =
↪False
Qobj data =
[[ 1.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+1.j  0.+0.j]
 [ 0.+0.j  0.+1.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j  1.+0.j]]
```

sqrtswap (*N=None, targets=[0, 1]*)

Quantum object representing the square root SWAP gate.

> **Returns sqrtswap_gate** : qobj
>
> > Quantum object representation of square root SWAP gate

sqrtiswap (*N=None, targets=[0, 1]*)

Quantum object representing the square root iSWAP gate.

> **Returns sqrtiswap_gate** : qobj
>
> > Quantum object representation of square root iSWAP gate

Examples

```
>>> sqrtiswap()
Quantum object: dims = [[2, 2], [2, 2]], shape = [4, 4], type = oper, isHerm =
↪False
Qobj data =
[[ 1.00000000+0.j   0.00000000+0.j          0.00000000+0.j          0.
↪00000000+0.j]
 [ 0.00000000+0.j   0.70710678+0.j          0.00000000-0.70710678j  0.
↪00000000+0.j]
 [ 0.00000000+0.j   0.00000000-0.70710678j      0.70710678+0.j          0.
↪00000000+0.j]
 [ 0.00000000+0.j   0.00000000+0.j          0.00000000+0.j          1.
↪00000000+0.j]]
```

fredkin (*N=None, control=0, targets=[1, 2]*)

Quantum object representing the Fredkin gate.

> **Returns fredkin_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]]
```

toffoli (*N=None, controls=[0, 1], target=2*)

> 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]]
```

rotation (*op*, *phi*, *N=None, target=0*)

> Single-qubit rotation for operator op with angle phi.

> **Returns result** : qobj

>> Quantum object for operator describing the rotation.

controlled_gate (*U*, *N=2, control=0, target=1, control_value=1*)

> Create an N-qubit controlled gate from a single-qubit gate U with the given control and target qubits.

> **Parameters U** : Qobj

>> Arbitrary single-qubit gate.

> **N** : integer

>> The number of qubits in the target space.

> **control** : integer

>> The index of the first control qubit.

> **target** : integer

The index of the target qubit.

control_value : integer (1)

The state of the control qubit that activates the gate U.

Returns result : qobj

Quantum object representing the controlled-U gate.

globalphase(*theta*, *N=1*)

Returns quantum object representing the global phase shift gate.

Parameters theta : float

Phase rotation angle.

Returns phase_gate : qobj

Quantum object representation of global phase shift gate.

Examples

```
>>> phasegate(pi/4)
Quantum object: dims = [[2], [2]], shape = [2, 2], type = oper, isHerm = False
Qobj data =
[[ 0.70710678+0.70710678j         0.00000000+0.j]
 [ 0.00000000+0.j         0.70710678+0.70710678j]]
```

hadamard_transform(*N=1*)

Quantum object representing the N-qubit Hadamard gate.

Returns q : qobj

Quantum object representation of the N-qubit Hadamard gate.

gate_sequence_product(*U_list*, *left_to_right=True*)

Calculate the overall unitary matrix for a given list of unitary operations

Parameters U_list : list

List of gates implementing the quantum circuit.

left_to_right : Boolean

Check if multiplication is to be done from left to right.

Returns U_overall : qobj

Overall unitary matrix of a given quantum circuit.

gate_expand_1toN(*U*, *N*, *target*)

Create a Qobj representing a one-qubit gate that act on a system with N qubits.

Parameters U : Qobj

The one-qubit gate

N : integer

The number of qubits in the target space.

target : integer

The index of the target qubit.

Returns gate : qobj

Quantum object representation of N-qubit gate.

gate_expand_2toN (*U*, *N*, *control=None*, *target=None*, *targets=None*)

   Create a Qobj representing a two-qubit gate that act on a system with N qubits.

> **Parameters** **U** : Qobj
>
>> The two-qubit gate
>
>> **N** : integer
>>
>>> The number of qubits in the target space.
>
>> **control** : integer
>>
>>> The index of the control qubit.
>
>> **target** : integer
>>
>>> The index of the target qubit.
>
>> **targets** : list
>>
>>> List of target qubits.
>
> **Returns** **gate** : qobj
>
>> Quantum object representation of N-qubit gate.

gate_expand_3toN (*U, N, controls=[0, 1], target=2*)

   Create a Qobj representing a three-qubit gate that act on a system with N qubits.

> **Parameters** **U** : Qobj
>
>> The three-qubit gate
>
>> **N** : integer
>>
>>> The number of qubits in the target space.
>
>> **controls** : list
>>
>>> The list of the control qubits.
>
>> **target** : integer
>>
>>> The index of the target qubit.
>
> **Returns** **gate** : qobj
>
>> Quantum object representation of N-qubit gate.

Qubits

qubit_states (*N=1, states=[0]*)

   Function to define initial state of the qubits.

> **Parameters** **N** : Integer
>
>> Number of qubits in the register.
>
>> **states** : List
>>
>>> Initial state of each qubit.
>
> **Returns** **qstates** : Qobj
>
>> List of qubits.

Algorithms

This module provides the circuit implementation for Quantum Fourier Transform.

qft (*N=1*)
> Quantum Fourier Transform operator on N qubits.

>> **Parameters N** : int

>>> Number of qubits.

>> **Returns** QFT: qobj

>>> Quantum Fourier transform operator.

qft_steps (*N=1*, *swapping=True*)
> Quantum Fourier Transform operator on N qubits returning the individual steps as unitary matrices operating from left to right.

>> **Parameters N: int**

>>> Number of qubits.

>> **swap: boolean**

>>> Flag indicating sequence of swap gates to be applied at the end or not.

>> **Returns** U_step_list: list of qobj

>>> List of Hadamard and controlled rotation gates implementing QFT.

qft_gate_sequence (*N=1*, *swapping=True*)
> Quantum Fourier Transform operator on N qubits returning the gate sequence.

>> **Parameters N: int**

>>> Number of qubits.

>> **swap: boolean**

>>> Flag indicating sequence of swap gates to be applied at the end or not.

>> **Returns** qc: instance of QubitCircuit

>>> Gate sequence of Hadamard and controlled rotation gates implementing QFT.

4.2.6 non-Markovian Solvers

This module contains an implementation of the non-Markovian transfer tensor method (TTM), introduced in [1].

[1] Javier Cerrillo and Jianshu Cao, Phys. Rev. Lett 112, 110401 (2014)

ttmsolve (*dynmaps*, *rho0*, *times*, *e_ops=[]*, *learningtimes=None*, *tensors=None*, ***kwargs*)
> Solve time-evolution using the Transfer Tensor Method, based on a set of precomputed dynamical maps.

>> **Parameters dynmaps** : list of `qutip.Qobj`

>>> List of precomputed dynamical maps (superoperators), or a callback function that returns the superoperator at a given time.

>> **rho0** : `qutip.Qobj`

>>> Initial density matrix or state vector (ket).

>> **times** : array_like

>>> list of times t_n at which to compute $\rho(t_n)$. Must be uniformily spaced.

>> **e_ops** : list of `qutip.Qobj` / callback function

>>> single operator or list of operators for which to evaluate expectation values.

> **learningtimes** : array_like
>
>> list of times t_k for which we have knowledge of the dynamical maps $E(t_k)$.
>
> **tensors** : array_like
>
>> optional list of precomputed tensors T_k
>
> **kwargs** : dictionary
>
>> Optional keyword arguments. See `qutip.nonmarkov.ttm.TTMSolverOptions`.
>
> **Returns** output: `qutip.solver.Result`
>
>> An instance of the class `qutip.solver.Result`.

4.2.7 Optimal control

Wrapper functions that will manage the creation of the objects, build the configuration, and execute the algorithm required to optimise a set of ctrl pulses for a given (quantum) system. The fidelity error is some measure of distance of the system evolution from the given target evolution in the time allowed for the evolution. The functions minimise this fidelity error wrt the piecewise control amplitudes in the timeslots

There are currently two quantum control pulse optmisations algorithms implemented in this library. There are accessible through the methods in this module. Both the algorithms use the scipy.optimize methods to minimise the fidelity error with respect to to variables that define the pulse.

GRAPE

The default algorithm (as it was implemented here first) is GRAPE GRadient Ascent Pulse Engineering [1][2]. It uses a gradient based method such as BFGS to minimise the fidelity error. This makes convergence very quick when an exact gradient can be calculated, but this limits the factors that can taken into account in the fidelity.

CRAB

The CRAB [3][4] algorithm was developed at the University of Ulm. In full it is the Chopped RAndom Basis algorithm. The main difference is that it reduces the number of optimisation variables by defining the control pulses by expansions of basis functions, where the variables are the coefficients. Typically a Fourier series is chosen, i.e. the variables are the Fourier coefficients. Therefore it does not need to compute an explicit gradient. By default it uses the Nelder-Mead method for fidelity error minimisation.

References

1. N Khaneja et. al. Optimal control of coupled spin dynamics: Design of NMR pulse sequences by gradient ascent algorithms. J. Magn. Reson. 172, 296–305 (2005).

2. Shai Machnes et.al DYNAMO - Dynamic Framework for Quantum Optimal Control arXiv.1011.4874

3. Doria, P., Calarco, T. & Montangero, S. Optimal Control Technique for Many-Body Quantum Dynamics. Phys. Rev. Lett. 106, 1–4 (2011).

4. Caneva, T., Calarco, T. & Montangero, S. Chopped random-basis quantum optimization. Phys. Rev. A - At. Mol. Opt. Phys. 84, (2011).

optimize_pulse(*drift*, *ctrls*, *initial*, *target*, *num_tslots=None*, *evo_time=None*, *tau=None*, *amp_lbound=None*, *amp_ubound=None*, *fid_err_targ=1e-10*, *min_grad=1e-10*, *max_iter=500*, *max_wall_time=180*, *alg='GRAPE'*, *alg_params=None*, *optim_params=None*, *optim_method='DEF'*, *method_params=None*, *optim_alg=None*, *max_metric_corr=None*, *accuracy_factor=None*, *dyn_type='GEN_MAT'*, *dyn_params=None*, *prop_type='DEF'*, *prop_params=None*, *fid_type='DEF'*, *fid_params=None*, *phase_option=None*, *fid_err_scale_factor=None*, *tslot_type='DEF'*, *tslot_params=None*, *amp_update_mode=None*, *init_pulse_type='DEF'*, *init_pulse_params=None*, *pulse_scaling=1.0*, *pulse_offset=0.0*, *ramping_pulse_type=None*, *ramping_pulse_params=None*, *log_level=0*, *out_file_ext=None*, *gen_stats=False*)

Optimise a control pulse to minimise the fidelity error. The dynamics of the system in any given timeslot are governed by the combined dynamics generator, i.e. the sum of the drift+ctrl_amp[j]*ctrls[j] The control pulse is an [n_ts, n_ctrls)] array of piecewise amplitudes Starting from an intital (typically random) pulse, a multivariable optimisation algorithm attempts to determines the optimal values for the control pulse to minimise the fidelity error The fidelity error is some measure of distance of the system evolution from the given target evolution in the time allowed for the evolution.

    Parameters drift : Qobj or list of Qobj

        the underlying dynamics generator of the system can provide list (of length num_tslots) for time dependent drift

    ctrls : List of Qobj or array like [num_tslots, evo_time]

        a list of control dynamics generators. These are scaled by the amplitudes to alter the overall dynamics Array like imput can be provided for time dependent control generators

    initial : Qobj

        starting point for the evolution. Typically the identity matrix

    target : Qobj

        target transformation, e.g. gate or state, for the time evolution

    num_tslots : integer or None

        number of timeslots. None implies that timeslots will be given in the tau array

    evo_time : float or None

        total time for the evolution None implies that timeslots will be given in the tau array

    tau : array[num_tslots] of floats or None

        durations for the timeslots. if this is given then num_tslots and evo_time are derived from it None implies that timeslot durations will be equal and calculated as evo_time/num_tslots

    amp_lbound : float or list of floats

        lower boundaries for the control amplitudes Can be a scalar value applied to all controls or a list of bounds for each control

    amp_ubound : float or list of floats

        upper boundaries for the control amplitudes Can be a scalar value applied to all controls or a list of bounds for each control

    fid_err_targ : float

        Fidelity error target. Pulse optimisation will terminate when the fidelity error falls below this value

    mim_grad : float

Minimum gradient. When the sum of the squares of the gradients wrt to the control amplitudes falls below this value, the optimisation terminates, assuming local minima

max_iter : integer

Maximum number of iterations of the optimisation algorithm

max_wall_time : float

Maximum allowed elapsed time for the optimisation algorithm

alg : string

Algorithm to use in pulse optimisation. Options are:

'GRAPE' (default) - GRadient Ascent Pulse Engineering 'CRAB' - Chopped RAndom Basis

alg_params : Dictionary

options that are specific to the algorithm see above

optim_params : Dictionary

The key value pairs are the attribute name and value used to set attribute values Note: attributes are created if they do not exist already, and are overwritten if they do. Note: method_params are applied afterwards and so may override these

optim_method : string

a scipy.optimize.minimize method that will be used to optimise the pulse for minimum fidelity error Note that FMIN, FMIN_BFGS & FMIN_L_BFGS_B will all result in calling these specific scipy.optimize methods Note the LBFGSB is equivalent to FMIN_L_BFGS_B for backwards capatibility reasons. Supplying DEF will given alg dependent result:

GRAPE - Default optim_method is FMIN_L_BFGS_B CRAB - Default optim_method is FMIN

method_params : dict

Parameters for the optim_method. Note that where there is an attribute of the Optimizer object or the termination_conditions matching the key that attribute. Otherwise, and in some case also, they are assumed to be method_options for the scipy.optimize.minimize method.

optim_alg : string

Deprecated. Use optim_method.

max_metric_corr : integer

Deprecated. Use method_params instead

accuracy_factor : float

Deprecated. Use method_params instead

dyn_type : string

Dynamics type, i.e. the type of matrix used to describe the dynamics. Options are UNIT, GEN_MAT, SYMPL (see Dynamics classes for details)

dyn_params : dict

Parameters for the Dynamics object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

prop_type : string

Propagator type i.e. the method used to calculate the propagtors and propagtor gradient for each timeslot options are DEF, APPROX, DIAG, FRECHET, AUG_MAT DEF will use the default for the specific dyn_type (see PropagatorComputer classes for details)

prop_params : dict

Parameters for the PropagatorComputer object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

fid_type : string

Fidelity error (and fidelity error gradient) computation method Options are DEF, UNIT, TRACEDIFF, TD_APPROX DEF will use the default for the specific dyn_type (See FidelityComputer classes for details)

fid_params : dict

Parameters for the FidelityComputer object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

phase_option : string

Deprecated. Pass in fid_params instead.

fid_err_scale_factor : float

Deprecated. Use scale_factor key in fid_params instead.

tslot_type : string

Method for computing the dynamics generators, propagators and evolution in the timeslots. Options: DEF, UPDATE_ALL, DYNAMIC UPDATE_ALL is the only one that currently works (See TimeslotComputer classes for details)

tslot_params : dict

Parameters for the TimeslotComputer object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

amp_update_mode : string

Deprecated. Use tslot_type instead.

init_pulse_type : string

type / shape of pulse(s) used to initialise the the control amplitudes. Options (GRAPE) include:

   RND, LIN, ZERO, SINE, SQUARE, TRIANGLE, SAW

DEF is RND (see PulseGen classes for details) For the CRAB the this the guess_pulse_type.

init_pulse_params : dict

Parameters for the initial / guess pulse generator object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

pulse_scaling : float

Linear scale factor for generated initial / guess pulses By default initial pulses are generated with amplitudes in the range (-1.0, 1.0). These will be scaled by this parameter

pulse_offset : float

Linear offset for the pulse. That is this value will be added to any initial / guess pulses generated.

ramping_pulse_type : string

Type of pulse used to modulate the control pulse. It's intended use for a ramping modulation, which is often required in experimental setups. This is only currently implemented in CRAB. GAUSSIAN_EDGE was added for this purpose.

ramping_pulse_params : dict

Parameters for the ramping pulse generator object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

log_level : integer

level of messaging output from the logger. Options are attributes of qutip.logging_utils, in decreasing levels of messaging, are: DEBUG_INTENSE, DEBUG_VERBOSE, DEBUG, INFO, WARN, ERROR, CRITICAL Anything WARN or above is effectively 'quiet' execution, assuming everything runs as expected. The default NOTSET implies that the level will be taken from the QuTiP settings file, which by default is WARN

out_file_ext : string or None

files containing the initial and final control pulse amplitudes are saved to the current directory. The default name will be postfixed with this extension Setting this to None will suppress the output of files

gen_stats : boolean

if set to True then statistics for the optimisation run will be generated - accessible through attributes of the stats object

Returns opt : OptimResult

Returns instance of OptimResult, which has attributes giving the reason for termination, final fidelity error, final evolution final amplitudes, statistics etc

optimize_pulse_unitary(*H_d*, *H_c*, *U_0*, *U_targ*, *num_tslots=None*, *evo_time=None*, *tau=None*, *amp_lbound=None*, *amp_ubound=None*, *fid_err_targ=1e-10*, *min_grad=1e-10*, *max_iter=500*, *max_wall_time=180*, *alg='GRAPE'*, *alg_params=None*, *optim_params=None*, *optim_method='DEF'*, *method_params=None*, *optim_alg=None*, *max_metric_corr=None*, *accuracy_factor=None*, *phase_option='PSU'*, *dyn_params=None*, *prop_params=None*, *fid_params=None*, *tslot_type='DEF'*, *tslot_params=None*, *amp_update_mode=None*, *init_pulse_type='DEF'*, *init_pulse_params=None*, *pulse_scaling=1.0*, *pulse_offset=0.0*, *ramping_pulse_type=None*, *ramping_pulse_params=None*, *log_level=0*, *out_file_ext=None*, *gen_stats=False*)

Optimise a control pulse to minimise the fidelity error, assuming that the dynamics of the system are generated by unitary operators. This function is simply a wrapper for optimize_pulse, where the appropriate options for unitary dynamics are chosen and the parameter names are in the format familiar to unitary dynamics The dynamics of the system in any given timeslot are governed by the combined Hamiltonian, i.e. the sum of the H_d + ctrl_amp[j]*H_c[j] The control pulse is an [n_ts, n_ctrls] array of piecewise amplitudes Starting from an intital (typically random) pulse, a multivariable optimisation algorithm attempts to determines the optimal values for the control pulse to minimise the fidelity error The maximum fidelity for a unitary system is 1, i.e. when the time evolution resulting from the pulse is equivalent to the target. And therefore the fidelity error is 1 - fidelity

Parameters H_d : Qobj or list of Qobj

Drift (aka system) the underlying Hamiltonian of the system can provide list (of length num_tslots) for time dependent drift

H_c : List of Qobj or array like [num_tslots, evo_time]

a list of control Hamiltonians. These are scaled by the amplitudes to alter the overall dynamics Array like imput can be provided for time dependent control generators

U_0 : Qobj

starting point for the evolution. Typically the identity matrix

U_targ : Qobj

target transformation, e.g. gate or state, for the time evolution

num_tslots : integer or None

number of timeslots. None implies that timeslots will be given in the tau array

evo_time : float or None

total time for the evolution None implies that timeslots will be given in the tau array

tau : array[num_tslots] of floats or None

durations for the timeslots. if this is given then num_tslots and evo_time are derived from it None implies that timeslot durations will be equal and calculated as evo_time/num_tslots

amp_lbound : float or list of floats

lower boundaries for the control amplitudes Can be a scalar value applied to all controls or a list of bounds for each control

amp_ubound : float or list of floats

upper boundaries for the control amplitudes Can be a scalar value applied to all controls or a list of bounds for each control

fid_err_targ : float

Fidelity error target. Pulse optimisation will terminate when the fidelity error falls below this value

mim_grad : float

Minimum gradient. When the sum of the squares of the gradients wrt to the control amplitudes falls below this value, the optimisation terminates, assuming local minima

max_iter : integer

Maximum number of iterations of the optimisation algorithm

max_wall_time : float

Maximum allowed elapsed time for the optimisation algorithm

alg : string

Algorithm to use in pulse optimisation. Options are:

'GRAPE' (default) - GRadient Ascent Pulse Engineering 'CRAB' - Chopped RAndom Basis

alg_params : Dictionary

options that are specific to the algorithm see above

optim_params : Dictionary

The key value pairs are the attribute name and value used to set attribute values Note: attributes are created if they do not exist already, and are overwritten if they do. Note: method_params are applied afterwards and so may override these

optim_method : string

a scipy.optimize.minimize method that will be used to optimise the pulse for minimum fidelity error Note that FMIN, FMIN_BFGS & FMIN_L_BFGS_B will all result in calling these specific scipy.optimize methods Note the LBFGSB is equivalent to FMIN_L_BFGS_B for backwards capatibility reasons. Supplying DEF will given alg dependent result:

> GRAPE - Default optim_method is FMIN_L_BFGS_B CRAB - Default optim_method is FMIN

method_params : dict

> Parameters for the optim_method. Note that where there is an attribute of the Optimizer object or the termination_conditions matching the key that attribute. Otherwise, and in some case also, they are assumed to be method_options for the scipy.optimize.minimize method.

optim_alg : string

> Deprecated. Use optim_method.

max_metric_corr : integer

> Deprecated. Use method_params instead

accuracy_factor : float

> Deprecated. Use method_params instead

phase_option : string

> determines how global phase is treated in fidelity calculations (fid_type='UNIT' only). Options:
>
> > PSU - global phase ignored SU - global phase included

dyn_params : dict

> Parameters for the Dynamics object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

prop_params : dict

> Parameters for the PropagatorComputer object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

fid_params : dict

> Parameters for the FidelityComputer object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

tslot_type : string

> Method for computing the dynamics generators, propagators and evolution in the timeslots. Options: DEF, UPDATE_ALL, DYNAMIC UPDATE_ALL is the only one that currently works (See TimeslotComputer classes for details)

tslot_params : dict

> Parameters for the TimeslotComputer object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

amp_update_mode : string

> Deprecated. Use tslot_type instead.

init_pulse_type : string

> type / shape of pulse(s) used to initialise the the control amplitudes. Options (GRAPE) include:
>
> > RND, LIN, ZERO, SINE, SQUARE, TRIANGLE, SAW DEF is RND
>
> (see PulseGen classes for details) For the CRAB the this the guess_pulse_type.

init_pulse_params : dict

> Parameters for the initial / guess pulse generator object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

pulse_scaling : float

>Linear scale factor for generated initial / guess pulses By default initial pulses are generated with amplitudes in the range (-1.0, 1.0). These will be scaled by this parameter

pulse_offset : float

>Linear offset for the pulse. That is this value will be added to any initial / guess pulses generated.

ramping_pulse_type : string

>Type of pulse used to modulate the control pulse. It's intended use for a ramping modulation, which is often required in experimental setups. This is only currently implemented in CRAB. GAUSSIAN_EDGE was added for this purpose.

ramping_pulse_params : dict

>Parameters for the ramping pulse generator object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

log_level : integer

>level of messaging output from the logger. Options are attributes of qutip.logging_utils, in decreasing levels of messaging, are: DEBUG_INTENSE, DEBUG_VERBOSE, DEBUG, INFO, WARN, ERROR, CRITICAL Anything WARN or above is effectively 'quiet' execution, assuming everything runs as expected. The default NOTSET implies that the level will be taken from the QuTiP settings file, which by default is WARN

out_file_ext : string or None

>files containing the initial and final control pulse amplitudes are saved to the current directory. The default name will be postfixed with this extension Setting this to None will suppress the output of files

gen_stats : boolean

>if set to True then statistics for the optimisation run will be generated - accessible through attributes of the stats object

Returns opt : OptimResult

>Returns instance of OptimResult, which has attributes giving the reason for termination, final fidelity error, final evolution final amplitudes, statistics etc

create_pulse_optimizer(*drift*, *ctrls*, *initial*, *target*, *num_tslots=None*, *evo_time=None*, *tau=None*, *amp_lbound=None*, *amp_ubound=None*, *fid_err_targ=1e-10*, *min_grad=1e-10*, *max_iter=500*, *max_wall_time=180*, *alg='GRAPE'*, *alg_params=None*, *optim_params=None*, *optim_method='DEF'*, *method_params=None*, *optim_alg=None*, *max_metric_corr=None*, *accuracy_factor=None*, *dyn_type='GEN_MAT'*, *dyn_params=None*, *prop_type='DEF'*, *prop_params=None*, *fid_type='DEF'*, *fid_params=None*, *phase_option=None*, *fid_err_scale_factor=None*, *tslot_type='DEF'*, *tslot_params=None*, *amp_update_mode=None*, *init_pulse_type='DEF'*, *init_pulse_params=None*, *pulse_scaling=1.0*, *pulse_offset=0.0*, *ramping_pulse_type=None*, *ramping_pulse_params=None*, *log_level=0*, *gen_stats=False*)

Generate the objects of the appropriate subclasses required for the pulse optmisation based on the parameters given Note this method may be preferable to calling optimize_pulse if more detailed configuration is required before running the optmisation algorthim, or the algorithm will be run many times, for instances when trying to finding global the optimum or minimum time optimisation

>**Parameters drift** : Qobj or list of Qobj

the underlying dynamics generator of the system can provide list (of length num_tslots) for time dependent drift

ctrls : List of Qobj or array like [num_tslots, evo_time]

a list of control dynamics generators. These are scaled by the amplitudes to alter the overall dynamics Array like imput can be provided for time dependent control generators

initial : Qobj

starting point for the evolution. Typically the identity matrix

target : Qobj

target transformation, e.g. gate or state, for the time evolution

num_tslots : integer or None

number of timeslots. None implies that timeslots will be given in the tau array

evo_time : float or None

total time for the evolution None implies that timeslots will be given in the tau array

tau : array[num_tslots] of floats or None

durations for the timeslots. if this is given then num_tslots and evo_time are derived from it None implies that timeslot durations will be equal and calculated as evo_time/num_tslots

amp_lbound : float or list of floats

lower boundaries for the control amplitudes Can be a scalar value applied to all controls or a list of bounds for each control

amp_ubound : float or list of floats

upper boundaries for the control amplitudes Can be a scalar value applied to all controls or a list of bounds for each control

fid_err_targ : float

Fidelity error target. Pulse optimisation will terminate when the fidelity error falls below this value

mim_grad : float

Minimum gradient. When the sum of the squares of the gradients wrt to the control amplitudes falls below this value, the optimisation terminates, assuming local minima

max_iter : integer

Maximum number of iterations of the optimisation algorithm

max_wall_time : float

Maximum allowed elapsed time for the optimisation algorithm

alg : string

Algorithm to use in pulse optimisation. Options are:

'GRAPE' (default) - GRadient Ascent Pulse Engineering 'CRAB' - Chopped RAndom Basis

alg_params : Dictionary

options that are specific to the algorithm see above

optim_params : Dictionary

The key value pairs are the attribute name and value used to set attribute values Note: attributes are created if they do not exist already, and are overwritten if they do. Note: method_params are applied afterwards and so may override these

optim_method : string

a scipy.optimize.minimize method that will be used to optimise the pulse for minimum fidelity error Note that FMIN, FMIN_BFGS & FMIN_L_BFGS_B will all result in calling these specific scipy.optimize methods Note the LBFGSB is equivalent to FMIN_L_BFGS_B for backwards capatibility reasons. Supplying DEF will given alg dependent result:

- GRAPE - Default optim_method is FMIN_L_BFGS_B

- CRAB - Default optim_method is Nelder-Mead

method_params : dict

Parameters for the optim_method. Note that where there is an attribute of the Optimizer object or the termination_conditions matching the key that attribute. Otherwise, and in some case also, they are assumed to be method_options for the scipy.optimize.minimize method.

optim_alg : string

Deprecated. Use optim_method.

max_metric_corr : integer

Deprecated. Use method_params instead

accuracy_factor : float

Deprecated. Use method_params instead

dyn_type : string

Dynamics type, i.e. the type of matrix used to describe the dynamics. Options are UNIT, GEN_MAT, SYMPL (see Dynamics classes for details)

dyn_params : dict

Parameters for the Dynamics object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

prop_type : string

Propagator type i.e. the method used to calculate the propagtors and propagtor gradient for each timeslot options are DEF, APPROX, DIAG, FRECHET, AUG_MAT DEF will use the default for the specific dyn_type (see PropagatorComputer classes for details)

prop_params : dict

Parameters for the PropagatorComputer object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

fid_type : string

Fidelity error (and fidelity error gradient) computation method Options are DEF, UNIT, TRACEDIFF, TD_APPROX DEF will use the default for the specific dyn_type (See FidelityComputer classes for details)

fid_params : dict

Parameters for the FidelityComputer object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

phase_option : string

Deprecated. Pass in fid_params instead.

4.2. Functions

fid_err_scale_factor : float

>   Deprecated. Use scale_factor key in fid_params instead.

tslot_type : string

>   Method for computing the dynamics generators, propagators and evolution in the timeslots. Options: DEF, UPDATE_ALL, DYNAMIC UPDATE_ALL is the only one that currently works (See TimeslotComputer classes for details)

tslot_params : dict

>   Parameters for the TimeslotComputer object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

amp_update_mode : string

>   Deprecated. Use tslot_type instead.

init_pulse_type : string

>   type / shape of pulse(s) used to initialise the the control amplitudes. Options (GRAPE) include:
>
>>   RND, LIN, ZERO, SINE, SQUARE, TRIANGLE, SAW DEF is RND
>
>   (see PulseGen classes for details) For the CRAB the this the guess_pulse_type.

init_pulse_params : dict

>   Parameters for the initial / guess pulse generator object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

pulse_scaling : float

>   Linear scale factor for generated initial / guess pulses By default initial pulses are generated with amplitudes in the range (-1.0, 1.0). These will be scaled by this parameter

pulse_offset : float

>   Linear offset for the pulse. That is this value will be added to any initial / guess pulses generated.

ramping_pulse_type : string

>   Type of pulse used to modulate the control pulse. It's intended use for a ramping modulation, which is often required in experimental setups. This is only currently implemented in CRAB. GAUSSIAN_EDGE was added for this purpose.

ramping_pulse_params : dict

>   Parameters for the ramping pulse generator object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

log_level : integer

>   level of messaging output from the logger. Options are attributes of qutip.logging_utils, in decreasing levels of messaging, are: DEBUG_INTENSE, DEBUG_VERBOSE, DEBUG, INFO, WARN, ERROR, CRITICAL Anything WARN or above is effectively 'quiet' execution, assuming everything runs as expected. The default NOTSET implies that the level will be taken from the QuTiP settings file, which by default is WARN

gen_stats : boolean

>   if set to True then statistics for the optimisation run will be generated - accessible through attributes of the stats object

   Returns opt : Optimizer

Instance of an Optimizer, through which the Config, Dynamics, PulseGen, and TerminationConditions objects can be accessed as attributes. The Propagator-Computer, FidelityComputer and TimeslotComputer objects can be accessed as attributes of the Dynamics object, e.g. optimizer.dynamics.fid_computer The optimisation can be run through the optimizer.run_optimization

opt_pulse_crab(*drift*, *ctrls*, *initial*, *target*, *num_tslots=None*, *evo_time=None*, *tau=None*, *amp_lbound=None*, *amp_ubound=None*, *fid_err_targ=1e-05*, *max_iter=500*, *max_wall_time=180*, *alg_params=None*, *num_coeffs=None*, *init_coeff_scaling=1.0*, *optim_params=None*, *optim_method='fmin'*, *method_params=None*, *dyn_type='GEN_MAT'*, *dyn_params=None*, *prop_type='DEF'*, *prop_params=None*, *fid_type='DEF'*, *fid_params=None*, *tslot_type='DEF'*, *tslot_params=None*, *guess_pulse_type=None*, *guess_pulse_params=None*, *guess_pulse_scaling=1.0*, *guess_pulse_offset=0.0*, *guess_pulse_action='MODULATE'*, *ramping_pulse_type=None*, *ramping_pulse_params=None*, *log_level=0*, *out_file_ext=None*, *gen_stats=False*)

Optimise a control pulse to minimise the fidelity error. The dynamics of the system in any given timeslot are governed by the combined dynamics generator, i.e. the sum of the drift+ctrl_amp[j]*ctrls[j] The control pulse is an [n_ts, n_ctrls] array of piecewise amplitudes. The CRAB algorithm uses basis function coefficents as the variables to optimise. It does NOT use any gradient function. A multivariable optimisation algorithm attempts to determines the optimal values for the control pulse to minimise the fidelity error The fidelity error is some measure of distance of the system evolution from the given target evolution in the time allowed for the evolution.

> **Parameters** **drift** : Qobj or list of Qobj
>
> > the underlying dynamics generator of the system can provide list (of length num_tslots) for time dependent drift
> >
> > **ctrls** : List of Qobj or array like [num_tslots, evo_time]
> >
> > a list of control dynamics generators. These are scaled by the amplitudes to alter the overall dynamics Array like imput can be provided for time dependent control generators
> >
> > **initial** : Qobj
> >
> > starting point for the evolution. Typically the identity matrix
> >
> > **target** : Qobj
> >
> > target transformation, e.g. gate or state, for the time evolution
> >
> > **num_tslots** : integer or None
> >
> > number of timeslots. None implies that timeslots will be given in the tau array
> >
> > **evo_time** : float or None
> >
> > total time for the evolution None implies that timeslots will be given in the tau array
> >
> > **tau** : array[num_tslots] of floats or None
> >
> > durations for the timeslots. if this is given then num_tslots and evo_time are derived from it None implies that timeslot durations will be equal and calculated as evo_time/num_tslots
> >
> > **amp_lbound** : float or list of floats
> >
> > lower boundaries for the control amplitudes Can be a scalar value applied to all controls or a list of bounds for each control
> >
> > **amp_ubound** : float or list of floats
> >
> > upper boundaries for the control amplitudes Can be a scalar value applied to all controls or a list of bounds for each control
> >
> > **fid_err_targ** : float

Fidelity error target. Pulse optimisation will terminate when the fidelity error falls below this value

max_iter : integer

Maximum number of iterations of the optimisation algorithm

max_wall_time : float

Maximum allowed elapsed time for the optimisation algorithm

alg_params : Dictionary

options that are specific to the algorithm see above

optim_params : Dictionary

The key value pairs are the attribute name and value used to set attribute values Note: attributes are created if they do not exist already, and are overwritten if they do. Note: method_params are applied afterwards and so may override these

coeff_scaling : float

Linear scale factor for the random basis coefficients By default these range from -1.0 to 1.0 Note this is overridden by alg_params (if given there)

num_coeffs : integer

Number of coefficients used for each basis function Note this is calculated automatically based on the dimension of the dynamics if not given. It is crucial to the performane of the algorithm that it is set as low as possible, while still giving high enough frequencies. Note this is overridden by alg_params (if given there)

optim_method : string

Multi-variable optimisation method The only tested options are 'fmin' and 'Nelder-mead' In theory any non-gradient method implemented in scipy.optimize.mininize could be used.

method_params : dict

Parameters for the optim_method. Note that where there is an attribute of the Optimizer object or the termination_conditions matching the key that attribute. Otherwise, and in some case also, they are assumed to be method_options for the scipy.optimize.minimize method. The commonly used parameter are:

xtol - limit on variable change for convergence ftol - limit on fidelity error change for convergence

dyn_type : string

Dynamics type, i.e. the type of matrix used to describe the dynamics. Options are UNIT, GEN_MAT, SYMPL (see Dynamics classes for details)

dyn_params : dict

Parameters for the Dynamics object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

prop_type : string

Propagator type i.e. the method used to calculate the propagtors and propagtor gradient for each timeslot options are DEF, APPROX, DIAG, FRECHET, AUG_MAT DEF will use the default for the specific dyn_type (see PropagatorComputer classes for details)

prop_params : dict

Parameters for the PropagatorComputer object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

fid_type : string

Fidelity error (and fidelity error gradient) computation method Options are DEF, UNIT, TRACEDIFF, TD_APPROX DEF will use the default for the specific dyn_type (See FidelityComputer classes for details)

fid_params : dict

Parameters for the FidelityComputer object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

tslot_type : string

Method for computing the dynamics generators, propagators and evolution in the timeslots. Options: DEF, UPDATE_ALL, DYNAMIC UPDATE_ALL is the only one that currently works (See TimeslotComputer classes for details)

tslot_params : dict

Parameters for the TimeslotComputer object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

guess_pulse_type : string

type / shape of pulse(s) used modulate the control amplitudes. Options include:

RND, LIN, ZERO, SINE, SQUARE, TRIANGLE, SAW, GAUSSIAN

Default is None

guess_pulse_params : dict

Parameters for the guess pulse generator object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

guess_pulse_action : string

Determines how the guess pulse is applied to the pulse generated by the basis expansion. Options are: MODULATE, ADD Default is MODULATE

pulse_scaling : float

Linear scale factor for generated guess pulses By default initial pulses are generated with amplitudes in the range (-1.0, 1.0). These will be scaled by this parameter

pulse_offset : float

Linear offset for the pulse. That is this value will be added to any guess pulses generated.

ramping_pulse_type : string

Type of pulse used to modulate the control pulse. It's intended use for a ramping modulation, which is often required in experimental setups. This is only currently implemented in CRAB. GAUSSIAN_EDGE was added for this purpose.

ramping_pulse_params : dict

Parameters for the ramping pulse generator object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

log_level : integer

level of messaging output from the logger. Options are attributes of qutip.logging_utils, in decreasing levels of messaging, are: DEBUG_INTENSE, DEBUG_VERBOSE, DEBUG, INFO, WARN, ERROR, CRITICAL Anything WARN or above is effectively 'quiet' execution, assuming everything runs as expected. The default NOTSET implies that the level will be taken from the QuTiP settings file, which by default is WARN

out_file_ext : string or None

---

files containing the initial and final control pulse amplitudes are saved to the current directory. The default name will be postfixed with this extension Setting this to None will suppress the output of files

gen_stats : boolean

if set to True then statistics for the optimisation run will be generated - accessible through attributes of the stats object

Returns opt : OptimResult

Returns instance of OptimResult, which has attributes giving the reason for termination, final fidelity error, final evolution final amplitudes, statistics etc

opt_pulse_crab_unitary(*H_d, H_c, U_0, U_targ, num_tslots=None, evo_time=None, tau=None, amp_lbound=None, amp_ubound=None, fid_err_targ=1e-05, max_iter=500, max_wall_time=180, alg_params=None, num_coeffs=None, init_coeff_scaling=1.0, optim_params=None, optim_method='fmin', method_params=None, phase_option='PSU', dyn_params=None, prop_params=None, fid_params=None, tslot_type='DEF', tslot_params=None, guess_pulse_type=None, guess_pulse_params=None, guess_pulse_scaling=1.0, guess_pulse_offset=0.0, guess_pulse_action='MODULATE', ramping_pulse_type=None, ramping_pulse_params=None, log_level=0, out_file_ext=None, gen_stats=False*)

Optimise a control pulse to minimise the fidelity error, assuming that the dynamics of the system are generated by unitary operators. This function is simply a wrapper for optimize_pulse, where the appropriate options for unitary dynamics are chosen and the parameter names are in the format familiar to unitary dynamics The dynamics of the system in any given timeslot are governed by the combined Hamiltonian, i.e. the sum of the H_d + ctrl_amp[j]*H_c[j] The control pulse is an [n_ts, n_ctrls] array of piecewise amplitudes

The CRAB algorithm uses basis function coeffecents as the variables to optimise. It does NOT use any gradient function. A multivariable optimisation algorithm attempts to determines the optimal values for the control pulse to minimise the fidelity error The fidelity error is some measure of distance of the system evolution from the given target evolution in the time allowed for the evolution.

Parameters H_d : Qobj or list of Qobj

Drift (aka system) the underlying Hamiltonian of the system can provide list (of length num_tslots) for time dependent drift

H_c : List of Qobj or array like [num_tslots, evo_time]

a list of control Hamiltonians. These are scaled by the amplitudes to alter the overall dynamics Array like imput can be provided for time dependent control generators

U_0 : Qobj

starting point for the evolution. Typically the identity matrix

U_targ : Qobj

target transformation, e.g. gate or state, for the time evolution

num_tslots : integer or None

number of timeslots. None implies that timeslots will be given in the tau array

evo_time : float or None

total time for the evolution None implies that timeslots will be given in the tau array

tau : array[num_tslots] of floats or None

durations for the timeslots. if this is given then num_tslots and evo_time are derived from it None implies that timeslot durations will be equal and calculated as evo_time/num_tslots

amp_lbound : float or list of floats

lower boundaries for the control amplitudes Can be a scalar value applied to all controls or a list of bounds for each control

amp_ubound : float or list of floats

upper boundaries for the control amplitudes Can be a scalar value applied to all controls or a list of bounds for each control

fid_err_targ : float

Fidelity error target. Pulse optimisation will terminate when the fidelity error falls below this value

max_iter : integer

Maximum number of iterations of the optimisation algorithm

max_wall_time : float

Maximum allowed elapsed time for the optimisation algorithm

alg_params : Dictionary

options that are specific to the algorithm see above

optim_params : Dictionary

The key value pairs are the attribute name and value used to set attribute values Note: attributes are created if they do not exist already, and are overwritten if they do. Note: method_params are applied afterwards and so may override these

coeff_scaling : float

Linear scale factor for the random basis coefficients By default these range from -1.0 to 1.0 Note this is overridden by alg_params (if given there)

num_coeffs : integer

Number of coefficients used for each basis function Note this is calculated automatically based on the dimension of the dynamics if not given. It is crucial to the performane of the algorithm that it is set as low as possible, while still giving high enough frequencies. Note this is overridden by alg_params (if given there)

optim_method : string

Multi-variable optimisation method The only tested options are 'fmin' and 'Nelder-mead' In theory any non-gradient method implemented in scipy.optimize.mininize could be used.

method_params : dict

Parameters for the optim_method. Note that where there is an attribute of the Optimizer object or the termination_conditions matching the key that attribute. Otherwise, and in some case also, they are assumed to be method_options for the scipy.optimize.minimize method. The commonly used parameter are:

xtol - limit on variable change for convergence ftol - limit on fidelity error change for convergence

phase_option : string

determines how global phase is treated in fidelity calculations (fid_type='UNIT' only). Options:

PSU - global phase ignored SU - global phase included

dyn_params : dict

Parameters for the Dynamics object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

prop_params : dict

Parameters for the PropagatorComputer object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

fid_params : dict

Parameters for the FidelityComputer object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

tslot_type : string

Method for computing the dynamics generators, propagators and evolution in the timeslots. Options: DEF, UPDATE_ALL, DYNAMIC UPDATE_ALL is the only one that currently works (See TimeslotComputer classes for details)

tslot_params : dict

Parameters for the TimeslotComputer object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

guess_pulse_type : string

type / shape of pulse(s) used modulate the control amplitudes. Options include:

RND, LIN, ZERO, SINE, SQUARE, TRIANGLE, SAW, GAUSSIAN

Default is None

guess_pulse_params : dict

Parameters for the guess pulse generator object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

guess_pulse_action : string

Determines how the guess pulse is applied to the pulse generated by the basis expansion. Options are: MODULATE, ADD Default is MODULATE

pulse_scaling : float

Linear scale factor for generated guess pulses By default initial pulses are generated with amplitudes in the range (-1.0, 1.0). These will be scaled by this parameter

pulse_offset : float

Linear offset for the pulse. That is this value will be added to any guess pulses generated.

ramping_pulse_type : string

Type of pulse used to modulate the control pulse. It's intended use for a ramping modulation, which is often required in experimental setups. This is only currently implemented in CRAB. GAUSSIAN_EDGE was added for this purpose.

ramping_pulse_params : dict

Parameters for the ramping pulse generator object The key value pairs are assumed to be attribute name value pairs They applied after the object is created

log_level : integer

level of messaging output from the logger. Options are attributes of qutip.logging_utils, in decreasing levels of messaging, are: DEBUG_INTENSE, DEBUG_VERBOSE, DEBUG, INFO, WARN, ERROR, CRITICAL Anything WARN or above is effectively 'quiet' execution, assuming everything runs as expected. The default NOTSET implies that the level will be taken from the QuTiP settings file, which by default is WARN

out_file_ext : string or None

files containing the initial and final control pulse amplitudes are saved to the current directory. The default name will be postfixed with this extension Setting this to None will suppress the output of files

gen_stats : boolean

if set to True then statistics for the optimisation run will be generated - accessible through attributes of the stats object

Returns opt : OptimResult

Returns instance of OptimResult, which has attributes giving the reason for termination, final fidelity error, final evolution final amplitudes, statistics etc

Pulse generator - Generate pulses for the timeslots Each class defines a gen_pulse function that produces a float array of size num_tslots. Each class produces a differ type of pulse. See the class and gen_pulse function descriptions for details

create_pulse_gen (*pulse_type='RND'*, *dyn=None*, *pulse_params=None*)

Create and return a pulse generator object matching the given type. The pulse generators each produce a different type of pulse, see the gen_pulse function description for details. These are the random pulse options:

RND - Independent random value in each timeslot RNDFOURIER - Fourier series with random coefficients RNDWAVES - Summation of random waves RNDWALK1 - Random change in amplitude each timeslot RNDWALK2 - Random change in amp gradient each timeslot

These are the other non-periodic options:

LIN - Linear, i.e. contant gradient over the time ZERO - special case of the LIN pulse, where the gradient is 0

These are the periodic options

SINE - Sine wave SQUARE - Square wave SAW - Saw tooth wave TRIANGLE - Triangular wave

If a Dynamics object is passed in then this is used in instantiate the PulseGen, meaning that some timeslot and amplitude properties are copied over.

4.2.8 Utilitiy Functions

Graph Theory Routines

This module contains a collection of graph theory routines used mainly to reorder matrices for iterative steady state solvers.

breadth_first_search (*A*, *start*)

Breadth-First-Search (BFS) of a graph in CSR or CSC matrix format starting from a given node (row). Takes Qobjs and CSR or CSC matrices as inputs.

This function requires a matrix with symmetric structure. Use A+trans(A) if original matrix is not symmetric or not sure.

Parameters A : csc_matrix, csr_matrix

Input graph in CSC or CSR matrix format

start : int

Staring node for BFS traversal.

Returns order : array

Order in which nodes are traversed from starting node.

levels : array

Level of the nodes in the order that they are traversed.

graph_degree(*A*)

Returns the degree for the nodes (rows) of a symmetric graph in sparse CSR or CSC format, or a qobj.

> **Parameters A** : qobj, csr_matrix, csc_matrix
>
> > Input quantum object or csr_matrix.
>
> **Returns degree** : array
>
> > Array of integers giving the degree for each node (row).

reverse_cuthill_mckee(*A*, *sym=False*)

Returns the permutation array that orders a sparse CSR or CSC matrix in Reverse-Cuthill McKee ordering. Since the input matrix must be symmetric, this routine works on the matrix A+Trans(A) if the sym flag is set to False (Default).

It is assumed by default (*sym=False*) that the input matrix is not symmetric. This is because it is faster to do A+Trans(A) than it is to check for symmetry for a generic matrix. If you are guaranteed that the matrix is symmetric in structure (values of matrix element do not matter) then set *sym=True*

> **Parameters A** : csc_matrix, csr_matrix
>
> > Input sparse CSC or CSR sparse matrix format.
>
> **sym** : bool {False, True}
>
> > Flag to set whether input matrix is symmetric.
>
> **Returns perm** : array
>
> > Array of permuted row and column indices.

Notes

This routine is used primarily for internal reordering of Lindblad superoperators for use in iterative solver routines.

References

E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices", ACM '69 Proceedings of the 1969 24th national conference, (1969).

maximum_bipartite_matching(*A*, *perm_type='row'*)

Returns an array of row or column permutations that removes nonzero elements from the diagonal of a nonsingular square CSC sparse matrix. Such a permutation is always possible provided that the matrix is nonsingular. This function looks at the structure of the matrix only.

The input matrix will be converted to CSC matrix format if necessary.

> **Parameters A** : sparse matrix
>
> > Input matrix
>
> **perm_type** : str {'row', 'column'}
>
> > Type of permutation to generate.
>
> **Returns perm** : array
>
> > Array of row or column permutations.

Notes

This function relies on a maximum cardinality bipartite matching algorithm based on a breadth-first search (BFS) of the underlying graph[1]_.

References

I. S. Duff, K. Kaya, and B. Ucar, "Design, Implementation, and Analysis of Maximum Transversal Algorithms", ACM Trans. Math. Softw. 38, no. 2, (2011).

weighted_bipartite_matching(*A*, *perm_type='row'*)

Returns an array of row permutations that attempts to maximize the product of the ABS values of the diagonal elements in a nonsingular square CSC sparse matrix. Such a permutation is always possible provided that the matrix is nonsingular.

This function looks at both the structure and ABS values of the underlying matrix.

> **Parameters** **A** : csc_matrix
>
>> Input matrix
>
> **perm_type** : str {'row', 'column'}
>
>> Type of permutation to generate.
>
> **Returns** **perm** : array
>
>> Array of row or column permutations.

Notes

This function uses a weighted maximum cardinality bipartite matching algorithm based on breadth-first search (BFS). The columns are weighted according to the element of max ABS value in the associated rows and are traversed in descending order by weight. When performing the BFS traversal, the row associated to a given column is the one with maximum weight. Unlike other techniques[1]_, this algorithm does not guarantee the product of the diagonal is maximized. However, this limitation is offset by the substantially faster runtime of this method.

References

I. S. Duff and J. Koster, "The design and use of algorithms for permuting large entries to the diagonal of sparse matrices", SIAM J. Matrix Anal. and Applics. 20, no. 4, 889 (1997).

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 $\omega_{\text{th}} = k_B T / \hbar$.

> **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.

convert_unit (*value*, *orig='meV'*, *to='GHz'*)

Convert an energy from unit *orig* to unit *to*.

> **Parameters value** : float / array
>
> > The energy in the old unit.
>
> **orig** : string
>
> > The name of the original unit ("J", "eV", "meV", "GHz", "mK")
>
> **to** : string
>
> > The name of the new unit ("J", "eV", "meV", "GHz", "mK")

Returns value_new_unit : float / array

> The energy in the new unit.

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.

Parallelization

This function provides functions for parallel execution of loops and function mappings, using the builtin Python module multiprocessing.

parfor(*func*, **args*, ***kwargs*)

Executes a multi-variable function in parallel on the local machine.

Parallel execution of a for-loop over function *func* for multiple input arguments and keyword arguments.

Note: From QuTiP 3.1, we recommend to use `qutip.parallel_map` instead of this function.

Parameters func : function_type

A function to run in parallel on the local machine. The function 'func' accepts a series of arguments that are passed to the function as variables. In general, the function can have multiple input variables, and these arguments must be passed in the same order as they are defined in the function definition. In addition, the user can pass multiple keyword arguments to the function.

The following keyword argument is reserved:

num_cpus : int

Number of CPU's to use. Default uses maximum number of CPU's. Performance degrades if num_cpus is larger than the physical CPU count of your machine.

Returns result : list

A `list` with length equal to number of input parameters containing the output from *func*.

parallel_map(*task*, *values*, *task_args=()*, *task_kwargs={}*, ***kwargs*)

Parallel execution of a mapping of *values* to the function *task*. This is functionally equivalent to:

```
result = [task(value, *task_args, **task_kwargs) for value in values]
```

Parameters task : a Python function

The function that is to be called for each value in `task_vec`.

values : array / list

The list or array of values for which the `task` function is to be evaluated.

task_args : list / dictionary

The optional additional argument to the `task` function.

task_kwargs : list / dictionary

The optional additional keyword argument to the `task` function.

progress_bar : ProgressBar

Progress bar class instance for showing progress.

Returns result : list

The result list contains the value of `task(value, *task_args, **task_kwargs)` for each value in `values`.

serial_map(*task*, *values*, *task_args=()*, *task_kwargs={}*, ***kwargs*)

Serial mapping function with the same call signature as parallel_map, for easy switching between serial and parallel execution. This is functionally equivalent to:

```
result = [task(value, *task_args, **task_kwargs) for value in values]
```

This function work as a drop-in replacement of `qutip.parallel_map`.

> **Parameters** **task** : a Python function
>
> > The function that is to be called for each value in `task_vec`.
>
> **values** : array / list
>
> > The list or array of values for which the `task` function is to be evaluated.
>
> **task_args** : list / dictionary
>
> > The optional additional argument to the `task` function.
>
> **task_kwargs** : list / dictionary
>
> > The optional additional keyword argument to the `task` function.
>
> **progress_bar** : ProgressBar
>
> > Progress bar class instance for showing progress.
>
> **Returns** **result** : list
>
> > The result list contains the value of `task(value, *task_args, **task_kwargs)` for each value in `values`.

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]`.

parallel_map(*task*, *values*, *task_args=None*, *task_kwargs=None*, *client=None*, *view=None*, *progress_bar=None*, *show_scheduling=False*, ***kwargs*)

Call the function `task` for each value in `values` using a cluster of IPython engines. The function `task` should have the signature `task(value, *args, **kwargs)`.

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`.

values: array / list

> The list or array of values for which the `task` function is to be evaluated.

task_args: list / dictionary

> The optional additional argument to the `task` function.

task_kwargs: list / dictionary

> The optional additional keyword argument to the `task` function.

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 during the execution of the parfor loop.

Returns **result** : list

> The result list contains the value of `task(value, task_args, task_kwargs)` for each value in `values`.

version_table(*verbose=False*)

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, Cython, and MatPlotLib.

simdiag(*ops*, *evals=True*)

> Simulateous diagonalization of communting Hermitian matrices..

> > **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 5

Change Log

5.1 Version 4.3.0 ()

5.1.1 Improvements

- Added fast projector method, `Q.proj()`.
- Computing matrix elements, `Q.matrix_element` is now ~10x faster.
- Computing expectation values for ket vectors using `expect` is now ~10x faster.
- `Q.tr()` is now faster for small Hilbert space dimensions.

5.1.2 Bug Fixes

- Fixed bug in trace-norm for non-Hermitian operators.

5.2 Version 4.2.0 (July 28, 2017)

5.2.1 Improvements

- **MAJOR FEATURE**: Initial implementation of time-dependent Bloch-Redfield Solver.
- Qobj tidyup is now an order of magnitude faster.
- Time-dependent codegen now generates output NumPy arrays faster.
- Improved calculation for analytic coefficients in coherent states (Sebastian Kramer).
- Input array to correlation FFT method now checked for validity.
- Function-based time-dependent mesolve and sesolve routines now faster.
- Codegen now makes sure that division is done in C, as opposed to Python.
- Can now set different controls for a each timeslot in quantum optimization.

This allows time-varying controls to be used in pulse optimisation.

5.2.2 Bug Fixes

- rcsolve importing old Odeoptions Class rather than Options.
- Non-int issue in spin Q and Wigner functions.

- Qobj's should tidyup before determining isherm.

- Fixed time-dependent RHS function loading on Win.

- Fixed several issues with compiling with Cython 0.26.

- Liouvillian superoperators were hard setting isherm=True by default.

- Fixed an issue with the solver safety checks when inputing a list

with Python functions as time-dependence.

- Fixed non-int issue in Wigner_cmap.

- MKL solver error handling not working properly.

5.3 Version 4.1.0 (March 10, 2017)

5.3.1 Improvements

Core libraries

- **MAJOR FEATURE**: QuTiP now works for Python 3.5+ on Windows using Visual Studio 2015.
- **MAJOR FEATURE**: Cython and other low level code switched to C++ for MS Windows compatibility.
- **MAJOR FEATURE**: Can now use interpolating cubic splines as time-dependent coefficients.
- **MAJOR FEATURE**: Sparse matrix - vector multiplication now parallel using OPENMP.
- Automatic tuning of OPENMP threading threshold.
- Partial trace function is now up to 100x+ faster.
- Hermitian verification now up to 100x+ faster.
- Internal Qobj objects now created up to 60x faster.
- Inplace conversion from COO -> CSR sparse formats (e.g. Memory efficiency improvement.)
- Faster reverse Cuthill-Mckee and sparse one and inf norms.

5.3.2 Bug Fixes

- Cleanup of temp. Cython files now more robust and working under Windows.

5.4 Version 4.0.2 (January 5, 2017)

5.4.1 Bug Fixes

- td files no longer left behind by correlation tests
- Various fast sparse fixes

5.5 Version 4.0.0 (December 22, 2016)

5.5.1 Improvements

Core libraries

- **MAJOR FEATURE**: Fast sparse: New subclass of csr_matrix added that overrides commonly used methods to avoid certain checks that incurr execution cost. All Qobj.data now fast_csr_matrix

- HEOM performance enhancements

- spmv now faster

- mcsolve codegen further optimised

Control modules

- Time dependent drift (through list of pwc dynamics generators)

- memory optimisation options provided for control.dynamics

5.5.2 Bug Fixes

- recompilation of pyx files on first import removed

- tau array in control.pulseoptim funcs now works

5.6 Version 3.2.0 (Never officially released)

5.6.1 New Features

Core libraries

- **MAJOR FEATURE**: Non-Markovian solvers: Hierarchy (**Added by Neill Lambert**), Memory-Cascade, and Transfer-Tensor methods.

- **MAJOR FEATURE**: Default steady state solver now up to 100x faster using the Intel Pardiso library under the Anaconda and Intel Python distributions.

- The default Wigner function now uses a Clenshaw summation algorithm to evaluate a polynomial series that is applicable for any number of exciations (previous limitation was ~50 quanta), and is ~3x faster than before. (**Added by Denis Vasilyev**)

- Can now define a given eigen spectrum for random Hermitian and density operators.

- The Qobj `expm` method now uses the equivilent SciPy routine, and performs a much faster `exp` operation if the matrix is diagonal.

- One can now build zero operators using the `qzero` function.

Control modules

- **MAJOR FEATURE**: CRAB algorithm added This is an alternative to the GRAPE algorithm, which allows for analytical control functions, which means that experimental constraints can more easily be added into optimisation. See tutorial notebook for full information.

5.6.2 Improvements

Core libraries

- Two-time correlation functions can now be calculated for fully time-dependent Hamiltonians and collapse operators. (**Added by Kevin Fischer**)

- The code for the inverse-power method for the steady state solver has been simplified.

- Bloch-Redfield tensor creation is now up to an order of magnitude faster. (**Added by Johannes Feist**)

- Q.transform now works properly for arrays directly from sp_eigs (or eig).

- Q.groundstate now checks for degeneracy.

- Added `sinm` and `cosm` methods to the Qobj class.

- Added `charge` and `tunneling` operators.

- Time-dependent Cython code is now easier to read and debug.

Control modules

- The internal state / quantum operator data type can now be either Qobj or ndarray Previous only ndarray was possible. This now opens up possibility of using Qobj methods in fidelity calculations The attributes and functions that return these operators are now preceded by an underscore, to indicate that the data type could change depending on the configuration options. In most cases these functions were for internal processing only anyway, and should have been 'private'. Accessors to the properties that could be useful outside of the library have been added. These always return Qobj. If the internal operator data type is not Qobj, then there could be signicant overhead in the conversion, and so this should be avoided during pulse optimisation. If custom sub-classes are developed that use Qobj properties and methods (e.g. partial trace), then it is very likely that it will be more efficient to set the internal data type to Qobj. The internal operator data will be chosen automatically based on the size and sparsity of the dynamics generator. It can be forced by setting `dynamics.oper_dtype = <type>` Note this can be done by passing `dyn_params={'oper_dtype':<type>}` in any of the pulseoptim functions.

  Some other properties and methods were renamed at the same time. A full list is given here.

    - All modules - function: `set_log_level` -> property: `log_level`
    - dynamics functions

        * `_init_lists` now `_init_evo`
        * `get_num_ctrls` now property: `num_ctrls`
        * `get_owd_evo_target` now property: `onto_evo_target`
        * `combine_dyn_gen` now `_combine_dyn_gen` (no longer returns a value)
        * `get_dyn_gen` now `_get_phased_dyn_gen`
        * `get_ctrl_den_gen` now `_get_phased_ctrl_dyn_gen`
        * `ensure_decomp_curr` now `_ensure_decomp_curr`
        * `spectral_decomp` now `_spectral_decomp`

    - dynamics properties

        * `evo_init2t` now `_fwd_evo` (`fwd_evo` as Qobj)
        * `evo_t2end` now `_onwd_evo` (`onwd_evo` as Qobj)
        * `evo_t2targ` now `_onto_evo` (`onto_evo` as Qobj)

    - fidcomp properties

        * `uses_evo_t2end` now `uses_onwd_evo`
        * `uses_evo_t2targ` now `uses_onto_evo`
        * `set_phase_option` function now property `phase_option`

    - propcomp properties

        * `grad_exact` (now read only)

    - propcomp functions

        * `compute_propagator` now `_compute_propagator`
        * `compute_diff_prop` now `_compute_diff_prop`
        * `compute_prop_grad` now `_compute_prop_grad`

    - tslotcomp functions

        * `get_timeslot_for_fidelity_calc` now `_get_timeslot_for_fidelity_calc`

Miscellaneous

- QuTiP Travis CI tests now use the Anaconda distribution.
- The `about` box and ipynb `version_table` now display addition system information.
- Updated Cython cleanup to remove depreciation warning in sysconfig.
- Updated ipynb_parallel to look for `ipyparallel` module in V4 of the notebooks.

5.6.3 Bug Fixes

- Fixes for countstat and psuedo-inverse functions
- Fixed Qobj division tests on 32-bit systems.
- Removed extra call to Python in time-dependent Cython code.
- Fixed issue with repeated Bloch sphere saving.
- Fixed T_0 triplet state not normalized properly. (**Fixed by Eric Hontz**)
- Simplified compiler flags (support for ARM systems).
- Fixed a decoding error in `qload`.
- Fixed issue using complex.h math and np.kind_t variables.
- Corrected output states mismatch for `ntraj=1` in the mcf90 solver.
- Qobj data is now copied by default to avoid a bug in multiplication. (**Fixed by Richard Brierley**)
- Fixed bug overwriting `hardware_info` in `__init__`. (**Fixed by Johannes Feist**)
- Restored ability to explicity set Q.isherm, Q.type, and Q.superrep.
- Fixed integer depreciation warnings from NumPy.
- Qobj * (dense vec) would result in a recursive loop.
- Fixed args=None -> args={} in correlation functions to be compatible with mesolve.
- Fixed depreciation warnings in mcsolve.
- Fixed neagtive only real parts in `rand_ket`.
- Fixed a complicated list-cast-map-list antipattern in super operator reps. (**Fixed by Stefan Krastanov**)
- Fixed incorrect `isherm` for `sigmam` spin operator.
- Fixed the dims when using `final_state_output` in `mesolve` and `sesolve`.

5.7 Version 3.1.0 (January 1, 2015):

5.7.1 New Features

- **MAJOR FEATURE**: New module for quantum control (qutip.control).
- **NAMESPACE CHANGE**: QuTiP no longer exports symbols from NumPy and matplotlib, so those modules must now be explicitly imported when required.
- New module for counting statistics.
- Stochastic solvers now run trajectories in parallel.
- New superoperator and tensor manipulation functions (super_tensor, composite, tensor_contract).
- New logging module for debugging (qutip.logging).
- New user-available API for parallelization (parallel_map).

- New enhanced (optional) text-based progressbar (qutip.ui.EnhancedTextProgressBar)

- Faster Python based monte carlo solver (mcsolve).

- Support for progress bars in propagator function.

- Time-dependent Cython code now calls complex cmath functions.

- Random numbers seeds can now be reused for successive calls to mcsolve.

- The Bloch-Redfield master equation solver now supports optional Lindblad type collapse operators.

- Improved handling of ODE integration errors in mesolve.

- Improved correlation function module (for example, improved support for time-dependent problems).

- Improved parallelization of mcsolve (can now be interrupted easily, support for IPython.parallel, etc.)

- Many performance improvements, and much internal code restructuring.

5.7.2 Bug Fixes

- Cython build files for time-dependent string format now removed automatically.

- Fixed incorrect solution time from inverse-power method steady state solver.

- mcsolve now supports *Options(store_states=True)*

- Fixed bug in *hadamard* gate function.

- Fixed compatibility issues with NumPy 1.9.0.

- Progressbar in mcsolve can now be suppressed.

- Fixed bug in *gate_expand_3toN*.

- Fixed bug for time-dependent problem (list string format) with multiple terms in coefficient to an operator.

5.8 Version 3.0.1 (Aug 5, 2014):

5.8.1 Bug Fixes

- Fix bug in create(), which returned a Qobj with CSC data instead of CSR.

- Fix several bugs in mcsolve: Incorrect storing of collapse times and collapse operator records. Incorrect averaging of expectation values for different trajectories when using only 1 CPU.

- Fix bug in parsing of time-dependent Hamiltonian/collapse operator arguments that occurred when the args argument is not a dictionary.

- Fix bug in internal _version2int function that cause a failure when parsingthe version number of the Cython package.

-

5.9 Version 3.0.0 (July 17, 2014):

5.9.1 New Features

- New module *qutip.stochastic* with stochastic master equation and stochastic Schrödinger equation solvers.

- Expanded steady state solvers. The function `steady` has been deprecated in favor of `steadystate`. The steadystate solver no longer use umfpack by default. New pre-processing methods for reordering and balancing the linear equation system used in direct solution of the steady state.

- New module *qutip.qip* with utilities for quantum information processing, including pre-defined quantum gates along with functions for expanding arbitrary 1, 2, and 3 qubit gates to N qubit registers, circuit representations, library of quantum algorithms, and basic physical models for some common QIP architectures.

- New module *qutip.distributions* with unified API for working with distribution functions.

- New format for defining time-dependent Hamiltonians and collapse operators, using a pre-calculated numpy array that specifies the values of the Qobj-coefficients for each time step.

- New functions for working with different superoperator representations, including Kraus and Chi representation.

- New functions for visualizing quantum states using Qubism and Schimdt plots: `plot_qubism` and `plot_schmidt`.

- Dynamics solver now support taking argument `e_ops` (expectation value operators) in dictionary form.

- Public plotting functions from the `qutip.visualization` module are now prefixed with `plot_` (e.g., `plot_fock_distribution`). The `plot_wigner` and `plot_wigner_fock_distribution` now supports 3D views in addition to contour views.

- New API and new functions for working with spin operators and states, including for example `spin_Jx`, `spin_Jy`, `spin_Jz` and `spin_state`, `spin_coherent`.

- The `expect` function now supports a list of operators, in addition to the previously supported list of states.

- Simplified creation of qubit states using `ket` function.

- The module `qutip.cyQ` has been renamed to `qutip.cy` and the sparse matrix-vector functions `spmv` and `spmv1d` has been combined into one function `spmv`. New functions for operating directly on the underlaying sparse CSR data have been added (e.g., `spmv_csr`). Performance improvements. New and improved Cython functions for calculating expectation values for state vectors, density matrices in matrix and vector form.

- The `concurrence` function now supports both pure and mixed states. Added function for calculating the entangling power of a two-qubit gate.

- Added function for generating (generalized) Lindblad dissipator superoperators.

- New functions for generating Bell states, and singlet and triplet states.

- QuTiP no longer contains the demos GUI. The examples are now available on the QuTiP web site. The `qutip.gui` module has been renamed to `qutip.ui` and does no longer contain graphical UI elements. New text-based and HTML-based progressbar classes.

- Support for harmonic oscillator operators/states in a Fock state basis that does not start from zero (e.g., in the range [M,N+1]). Support for eliminating and extracting states from Qobj instances (e.g., removing one state from a two-qubit system to obtain a three-level system).

- Support for time-dependent Hamiltonian and Liouvillian callback functions that depend on the instantaneous state, which for example can be used for solving master equations with mean field terms.

5.9.2 Improvements

- Restructured and optimized implementation of Qobj, which now has significantly lower memory footprint due to avoiding excessive copying of internal matrix data.

- The classes `OdeData`, `Odeoptions`, `Odeconfig` are now called `Result`, `Options`, and `Config`, respectively, and are available in the module *qutip.solver*.

- The `squeez` function has been renamed to `squeeze`.

- Better support for sparse matrices when calculating propagators using the `propagator` function.

- Improved Bloch sphere.

- Restructured and improved the module `qutip.sparse`, which now only operates directly on sparse matrices (not on Qobj instances).

- Improved and simplified implement of the `tensor` function.

- Improved performance, major code cleanup (including namespace changes), and numerous bug fixes.

- Benchmark scripts improved and restructured.

- QuTiP is now using continuous integration tests (TravisCI).

5.10 Version 2.2.0 (March 01, 2013):

5.10.1 New Features

- **Added Support for Windows**

- New Bloch3d class for plotting 3D Bloch spheres using Mayavi.

- Bloch sphere vectors now look like arrows.

- Partial transpose function.

- Continuos 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.

- New Wigner colormap for highlighting negative values.

- More graph styles to the visualization module.

5.10.2 Bug Fixes:

- Function based time-dependent Hamiltonians now keep the correct phase.

- mcsolve no longer prints to the command line if ntraj=1.

5.11 Version 2.1.0 (October 05, 2012):

5.11.1 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.

5.11.2 Bug Fixes:

- mesolver now correctly uses the user defined rhs_filename in Odeoptions().

- rhs_generate() now handles user defined filenames properly.

- Density matrix returned by propagator_steadystate is now Hermitian.

- eseries_value returns real list if all imag parts are zero.

- mcsolver now gives correct results for strong damping rates.

- Odeoptions now prints mc_avg correctly.

- Do not check for PyObj in mcsolve when gui=False.

- Eseries now correctly handles purely complex rates.

- thermal_dm() function now uses truncated operator method.

- Cython based time-dependence now Python 3 compatible.

- Removed call to NSAutoPool on mac systems.

- Progress bar now displays the correct number of CPU's used.

- Qobj.diag() returns reals if operator is Hermitian.

- Text for progress bar on Linux systems is no longer cutoff.

5.12 Version 2.0.0 (June 01, 2012):

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:

5.12.1 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.

Important: This breaks compatibility with QuTiP version 1.x.

- 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.

5.13 Version 1.1.4 (May 28, 2012):

5.13.1 Bug Fixes:

- Fixed bug pointed out by Brendan Abolins.

- Qobj.tr() returns zero-dim ndarray instead of float or complex.

- Updated factorial import for scipy version 0.10+

5.14 Version 1.1.3 (November 21, 2011):

5.14.1 New Functions:

- Allow custom naming of Bloch sphere.

5.14.2 Bug Fixes:

- Fixed text alignment issues in AboutBox.

- Added fix for SciPy V>0.10 where factorial was moved to scipy.misc module.

- Added tidyup function to tensor function output.

- Removed openmp flags from setup.py as new Mac Xcode compiler does not recognize them.

- Qobj diag method now returns real array if all imaginary parts are zero.

- Examples GUI now links to new documentation.

- Fixed zero-dimensional array output from metrics module.

5.15 Version 1.1.2 (October 27, 2011)

5.15.1 Bug Fixes

- Fixed issue where Monte Carlo states were not output properly.

5.16 Version 1.1.1 (October 25, 2011)

THIS POINT-RELEASE INCLUDES VASTLY IMPROVED TIME-INDEPENDENT MCSOLVE AND ODESOLVE PERFORMANCE

5.16.1 New Functions

- Added linear entropy function.
- Number of CPU's can now be changed.

5.16.2 Bug Fixes

- Metrics no longer use dense matrices.
- Fixed Bloch sphere grid issue with matplotlib 1.1.
- Qobj trace operation uses only sparse matrices.
- Fixed issue where GUI windows do not raise to front.

5.17 Version 1.1.0 (October 04, 2011)

THIS RELEASE NOW REQUIRES THE GCC COMPILER TO BE INSTALLED

5.17.1 New Functions

- tidyup function to remove small elements from a Qobj.
- Added concurrence function.
- Added simdiag for simultaneous diagonalization of operators.
- Added eigenstates method returning eigenstates and eigenvalues to Qobj class.
- Added fileio for saving and loading data sets and/or Qobj's.
- Added hinton function for visualizing density matrices.

5.17.2 Bug Fixes

- Switched Examples to new Signals method used in PySide 1.0.6+.
- Switched ProgressBar to new Signals method.
- Fixed memory issue in expm functions.
- Fixed memory bug in isherm.
- Made all Qobj data complex by default.
- Reduced ODE tolerance levels in Odeoptions.
- Fixed bug in ptrace where dense matrix was used instead of sparse.
- Fixed issue where PyQt4 version would not be displayed in about box.
- Fixed issue in Wigner where xvec was used twice (in place of yvec).

5.18 Version 1.0.0 (July 29, 2011)

- **Initial release.**

Chapter 6

Developers

6.1 Lead Developers

Robert Johansson (RIKEN)

Paul Nation (Korea University)

6.2 Contributors

Note: Anyone is welcome to contribute to QuTiP. If you are interested in helping, please let us know!

alexbrc (github user) - Code contributor

Alexander Pitchford (Aberystwyth University) - Code contributor

Amit Jamadagni - Bug fix

Anders Lund (Technical University of Denmark) - Bug hunting for the Monte-Carlo solver

Andre Carvalho - Bug hunter

André Xuereb (University of Hannover) - Bug hunter

Anubhav Vardhan (IIT, Kanpur) - Bug hunter, Code contributor, Documentation

Arne Grimsmo (University of Auckland) - Bug hunter, Code contributor

Ben Criger (Waterloo IQC) - Code contributor

Bredan Abolins (Berkeley) - Bug hunter

Chris Granade - Code contributor

Claudia Degrandi (Yale University) - Documentation

Dawid Crivelli - Bug hunter

Denis Vasilyev (St. Petersburg State University) - Code contributor

Dong Zhou (Yale University) - Bug hunter

Florian Ong (Institute for Quantum Computation) - Bug hunter

Frank Schima - Macports packaging

Henri Nielsen (Technical University of Denmark) - Bug hunter

Hwajung Kang (Systems Biology Institute, Tokyo) - Suggestions for improving Bloch class

James Clemens (Miami University - Ohio) - Bug hunter

Johannes Feist - Code contributor

Jonas Hörsch - Code contributor

Jonas Neergaard-Nielsen (Technical University of Denmark) - Code contributor, Windows support

JP Hadden (University of Bristol) - Code contributor, improved Bloch sphere visualization

Kevin Fischer (Stanford) - Code contributor

Laurence Stant - Documentation

Markus Baden (Centre for Quantum Technologies, Singapore) - Code contributor, Documentation

Myung-Joong Hwang (Pohang University of Science and Technology) - Bug hunter

Neill Lambert (RIKEN) - Code contributor, Windows support

Nikolas Tezak (Stanford) - Code contributor

Per Nielsen (Technical University of Denmark) - Bug hunter, Code contributor

Piotr Migdał (ICFO) - Code contributor

Reinier Heeres (Yale University) - Code contributor

Robert Jördens (NIST) - Linux packaging

Simon Whalen - Code contributor

W.M. Witzel - Bug hunter

Chapter 7

Bibliography

Chapter 8

Indices and tables

- genindex
- modindex
- search

Bibliography

[R1] Shore, B. W., "The Theory of Coherent Atomic Excitation", Wiley, 1990.

[R2] http://en.wikipedia.org/wiki/Concurrence_(quantum_computing)

[BCSZ08] 23. Bruzda, V. Cappellini, H.-J. Sommers, K. Życzkowski, *Random Quantum Operations*, Phys. Lett. A **373**, 320-324 (2009). doi:10.1016/j.physleta.2008.11.043.

[Hav03] Havel, T. *Robust procedures for converting among Lindblad, Kraus and matrix representations of quantum dynamical semigroups*. Journal of Mathematical Physics **44** 2, 534 (2003). doi:10.1063/1.1518555.

[Wat13] Watrous, J. *Theory of Quantum Information*, lecture notes.

[Mez07] 6. Mezzadri, *How to generate random matrices from the classical compact groups*, Notices of the AMS **54** 592-604 (2007). arXiv:math-ph/0609050.

[Mis12] 10. (a) Miszczak, *Generating and using truly random quantum states in Mathematica*, Computer Physics Communications **183** 1, 118-124 (2012). doi:10.1016/j.cpc.2011.08.002.

[Moh08] 13. Mohseni, A. T. Rezakhani, D. A. Lidar, *Quantum-process tomography: Resource analysis of different strategies*, Phys. Rev. A **77**, 032322 (2008). doi:10.1103/PhysRevA.77.032322.

[Gri98] 13. Grifoni, P. Hänggi, *Driven quantum tunneling*, Physics Reports **304**, 299 (1998). doi:10.1016/S0370-1573(98)00022-2.

[Cre03] 3. (a) Creffield, *Location of crossings in the Floquet spectrum of a driven two-level system*, Phys. Rev. B **67**, 165301 (2003). doi:10.1103/PhysRevB.67.165301.

[Gar03] Gardineer and Zoller, *Quantum Noise* (Springer, 2004).

[Bre02] H.-P. Breuer and F. Petruccione, *The Theory of Open Quantum Systems* (Oxford, 2002).

[Coh92] 3. Cohen-Tannoudji, J. Dupont-Roc, G. Grynberg, *Atom-Photon Interactions: Basic Processes and Applications*, (Wiley, 1992).

[WBC11] C. Wood, J. Biamonte, D. G. Cory, *Tensor networks and graphical calculus for open quantum systems*. arXiv:1111.6950

[dAless08] 4. d'Alessandro, *Introduction to Quantum Control and Dynamics*, (Chapman & Hall/CRC, 2008)

[Kha05] 14. Khaneja, T. Reiss, C. Kehlet, T. Schulte-Herbruggen, and S. J. Glaser, *Optimal control of coupled spin dynamics: Design of NMR pulse sequences by gradient ascent algorithms*, J. Magn. Reson. **172**, 296 (2005). doi:10.1016/j.jmr.2004.11.004

[Byrd95] 18. (a) Byrd, P. Lu, J. Nocedal, and C. Zhu, *A Limited Memory Algorithm for Bound Constrained Optimization*, SIAM J. Sci. Comput. **16**, 1190 (1995). doi:10.1137/0916069

[Flo12] 6. (a) Floether, P. de Fouquieres, and S. G. Schirmer, *Robust quantum gates for open systems via optimal control: Markovian versus non-Markovian dynamics*, New J. Phys. **14**, 073023 (2012). doi:10.1088/1367-2630/14/7/073023

[Lloyd14] 19. Lloyd and S. Montangero, *Information theoretical analysis of quantum optimal control*, Phys. Rev. Lett. **113**, 010502 (2014). doi:10.1103/PhysRevLett.113.010502

[Doria11] 16. Doria, T. Calarco & S. Montangero, *Optimal Control Technique for Many-Body Quantum Dynamics*, Phys. Rev. Lett. **106**, 190501 (2011). doi:10.1103/PhysRevLett.106.190501

[Caneva11]   20. Caneva, T. Calarco, & S. Montangero, *Chopped random-basis quantum optimization*, Phys. Rev. A **84**, 022326 (2011). doi:10.1103/PhysRevA.84.022326

[Rach15]   14. Rach, M. M. Müller, T. Calarco, and S. Montangero, *Dressing the chopped-random-basis optimization: A bandwidth-limited access to the trap-free landscape*, Phys. Rev. A. **92**, 062343 (2015). doi:10.1103/PhysRevA.92.062343

[DYNAMO]   19. Machnes, U. Sander, S. J. Glaser, P. De Fouquieres, A. Gruslys, S. Schirmer, and T. Schulte-Herbrueggen, *Comparing, Optimising and Benchmarking Quantum Control Algorithms in a Unifying Programming Framework*, Phys. Rev. A. **84**, 022305 (2010). arXiv:1011.4874

Python Module Index

q

Index