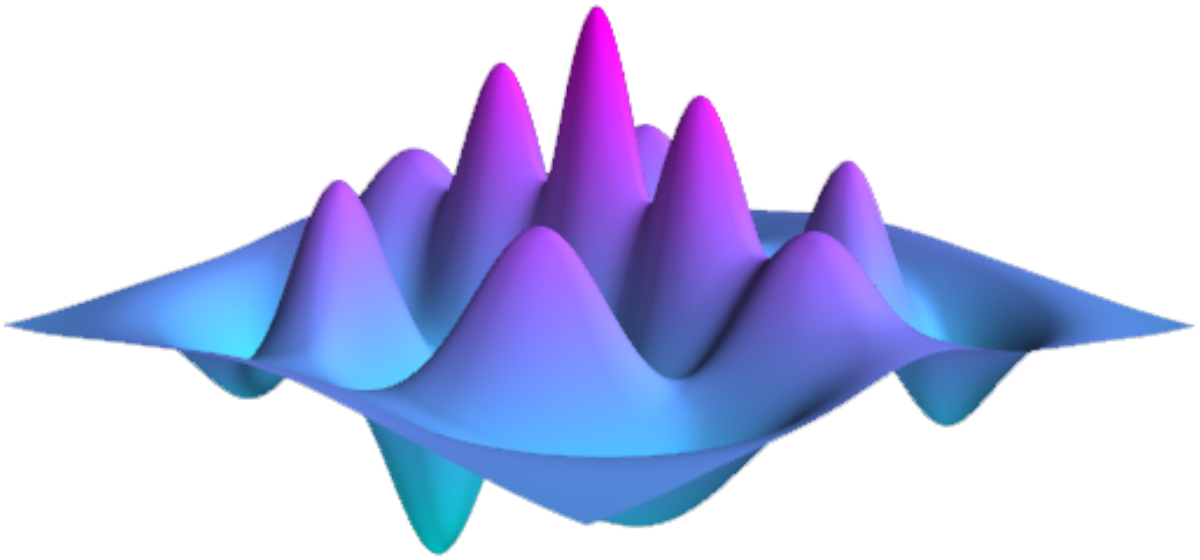


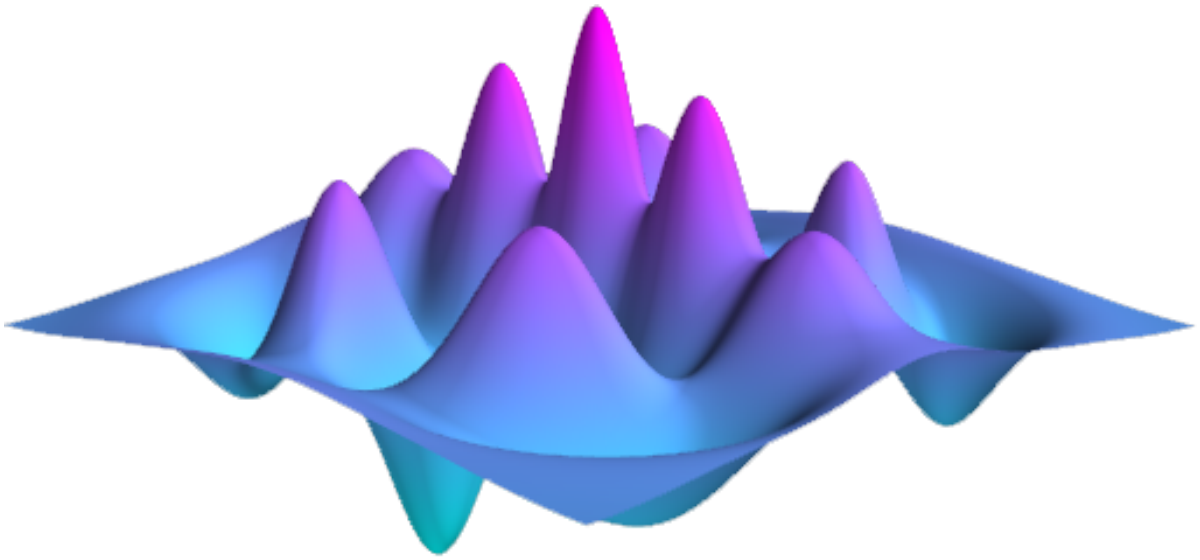
QuTiP: Quantum Toolbox in Python

Release 4.7.1

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

Dec 12, 2022

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 J.R. Johansson

Author P.D. Nation

Author Alexander Pitchford

Author Arne Grimsmo

Author Chris Grenade

Author Nathan Shammah

Author Shahnawaz Ahmed

Author Neill Lambert

Author Eric Giguere

Author Boxi Li

Author Jake Lishman

Author Simon Cross

release 4.7.1

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 downloaded from https://arxiv.org/abs/1211.6518 or https://arxiv.org/abs/1110.0573, respectively.

1.3 Funding

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

NUMFOCUS
OPEN CODE = BETTER SCIENCE

Unitary Fund

QuTiP was partially supported by

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

RIKEN

KOREA UNIVERSITY
고려대학교
KOREA UNIVERSITY

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 QuTiP Plugins

Several libraries depend on QuTiP heavily making QuTiP a super-library

Matsubara Matsubara is a plugin to study the ultrastrong coupling regime with structured baths

QNET QNET is a computer algebra package for quantum mechanics and photonic quantum networks

1.6 Libraries Using QuTiP

Several libraries rely on QuTiP for quantum physics or quantum information processing. Some of them are:

Krotov Krotov focuses on the python implementation of Krotov's method for quantum optimal control

pyEPR pyEPR interfaces classical distributed microwave analysis with that of quantum structures and hamiltonians by providing easy to use analysis function and automation for the design of quantum chips

scQubits scQubits is a Python library which provides a convenient way to simulate superconducting qubits by providing an interface to QuTiP

SimulaQron SimulaQron is a distributed simulation of the end nodes in a quantum internet with the specific goal to explore application development

QInfer QInfer is a library for working with sequential Monte Carlo methods for parameter estimation in quantum information

QPtomographer QPtomographer derive quantum error bars for quantum processes in terms of the diamond norm to a reference quantum channel

QuNetSim QuNetSim is a quantum networking simulation framework to develop and test protocols for quantum networks

qupulse qupulse is a toolkit to facilitate experiments involving pulse driven state manipulation of physical qubits

Pulser Pulser is a framework for composing, simulating and executing pulse sequences for neutral-atom quantum devices.

1.7 Contributing to QuTiP

We welcome anyone who is interested in helping us make QuTiP the best package for simulating quantum systems. There are *detailed instructions on how to contribute code and documentation* in the developers' section of this guide. You can also help out our users by answering questions in the QuTiP discussion mailing list, or by raising issues in the main GitHub repository if you find any bugs. Anyone who contributes code will be duly recognized. Even small contributions are noted. See *Contributors* for a list of people who have helped in one way or another.

Chapter 2

Installation

2.1 Quick Start

From QuTiP version 4.6 onwards, you should be able to get a working version of QuTiP with the standard

```
pip install qutip
```

It is not recommended to install any packages directly into the system Python environment; consider using `pip` or `conda` virtual environments to keep your operating system space clean, and to have more control over Python and other package versions.

You do not need to worry about the details on the rest of this page unless this command did not work, but do also read the next section for the list of optional dependencies. The rest of this page covers *installation directly from conda*, *installation from source*, and *additional considerations when working on Windows*.

2.2 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	3.6+	
NumPy	1.16+	
SciPy	1.0+	Lower versions may have missing features.

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

Package	Version	Details
`matplotlib`	1.2.1+	Needed for all visualisation tasks.
`cython`	0.29.20+	Needed for compiling some time-dependent Hamiltonians.
`cvxpy`	1.0+	Needed to calculate diamond norms.
C++ Compiler	GCC 4.7+, MS VS 2015	Needed for compiling Cython files, made when using string-format time-dependence.
`pytest`, `pytest-rerunfailures`	5.3+	For running the test suite.
LaTeX	TeXLive 2009+	Needed if using LaTeX in matplotlib figures, or for nice circuit drawings in IPython.

In addition, there are several additional packages that are not dependencies, but may give you a better programming experience. IPython provides an improved text-based Python interpreter that is far more full-featured that the

default interpreter, and runs in a terminal. If you prefer a more graphical set-up, Jupyter provides a notebook-style interface to mix code and mathematical notes together. Alternatively, Spyder is a free integrated development environment for Python, with several nice features for debugging code. QuTiP will detect if it is being used within one of these richer environments, and various outputs will have enhanced formatting.

2.3 Installing with conda

QuTiP is designed to work best when using the Anaconda or Intel Python distributions that support the conda package management system. It is still possible to use `pip` to install QuTiP while using conda, but uniformly using conda will make complete dependency management easier.

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

```
conda install qutip
```

This will install the minimum set of dependences, but none of the optional packages.

2.3.1 Adding the conda-forge channel

To install QuTiP from conda, you will need to add the conda-forge channel. The following command adds this channel with lowest priority, so conda will still try and install all other packages normally:

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

If you want to change the order of your channels later, you can edit your `.condarc` (user home folder) file manually, but it is recommended to keep `defaults` as the highest priority.

2.3.2 New conda environments

The default Anaconda environment has all the Python packages needed for running QuTiP installed already, so you will only need to add the `conda-forge` channel and then install the package. If you have only installed Miniconda, or you want a completely clean virtual environment to install QuTiP in, the `conda` package manager provides a convenient way to do this.

To create a conda environment for QuTiP called `qutip-env`:

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

This will automatically install all the necessary packages, and none of the optional packages. You activate the new environment by running

```
conda activate qutip-env
```

You can also install any more optional packages you want with `conda install`, for example `matplotlib`, `ipython` or `jupyter`.

2.4 Installing from Source

Official releases of QuTiP are available from the download section on the project's web pages, and the latest source code is available in our GitHub repository. 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.

You can install from source by using the *Python-recommended PEP 517 procedure*, or if you want more control or to have a development version, you can use the *low-level build procedure with setuptools*.

2.4.1 PEP 517 Source Builds

The easiest way to build QuTiP from source is to use a PEP-517-compatible builder such as the `build` package available on `pip`. These will automatically install all build dependencies for you, and the `pip` installation step afterwards will install the minimum runtime dependencies. You can do this by doing (for example)

```
pip install build
python -m build <path to qutip>
pip install <path to qutip>/dist/qutip-<version>.whl
```

The first command installs the reference PEP-517 build tool, the second effects the build and the third uses `pip` to install the built package. You will need to replace `<path to qutip>` with the actual path to the QuTiP source code. The string `<version>` will depend on the version of QuTiP, the version of Python and your operating system. It will look something like `4.6.0-cp39-cp39-manylinux1_x86_64`, but there should only be one `.whl` file in the `dist/` directory, which will be the correct one.

2.4.2 Direct Setuptools Source Builds

This is the method to have the greatest amount of control over the installation, but it the most error-prone and not recommended unless you know what you are doing. You first need to have all the runtime dependencies installed. The most up-to-date requirements will be listed in `pyproject.toml` file, in the `build-system.requires` key. As of the 4.6.0 release, the build requirements can be installed with

```
pip install setuptools wheel packaging 'cython>=0.29.20' 'numpy>=1.16.6,<1.20'
↪'scipy>=1.0'
```

or similar with `conda` if you prefer. You will also need to have a functional C++ compiler installed on your system. This is likely already done for you if you are on Linux or macOS, but see the *section on Windows installations* if that is your operating system.

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
```

This will attempt to load up OpenMP libraries during the compilation process, which depends on you having suitable C++ compiler and library support. If you are on Linux this is probably already done, but the compiler macOS ships with does not have OpenMP support. You will likely need to refer to external operating-system-specific guides for more detail here, as it may be very non-trivial to correctly configure.

If you wish to contribute to the QuTiP project, then you will want to create your own fork of the QuTiP git repository, clone this to a local folder, and install it into your Python environment using:

```
python setup.py develop
```

When you do `import qutip` in this environment, you will then load the code from your local fork, enabling you to edit the Python files and have the changes immediately available when you restart your Python interpreter, without needing to rebuild the package. Note that if you change any Cython files, you will need to rerun the build command.

You should not need to use `sudo` (or other superuser privileges) to install into a personal virtual environment; if it feels like you need it, there is a good chance that you are installing into the system Python environment instead.

2.5 Installation on Windows

As with other operating systems, the easiest method is to use `pip install qutip`, or use the `conda` procedure described above. If you want to build from source or use runtime compilation with Cython, you will need to have a working C++ compiler.

You can download the Visual Studio IDE from Microsoft, which has a free Community edition containing a sufficient C++ compiler. This is the recommended compiler toolchain on Windows. When installing, be sure to select the following components:

- Windows "X" SDK (where "X" stands for your version: 7/8/8.1/10)
- Visual Studio C++ build tools

You can then follow the *installation from source* section as normal.

Important: In order to prevent issues with the `PATH` environment variable not containing the compiler and associated libraries, it is recommended to use the developer command prompt in the Visual Studio installation folder instead of the built-in command prompt.

The Community edition of Visual Studio takes around 10GB of disk space. If this is prohibitive for you, it is also possible to install only the build tools and necessary SDKs instead, which should save about 2GB of space.

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 also have the `pytest` testing library. After installing QuTiP, leave the installation directory, run Python (or IPython), and call:

```
import qutip.testing
qutip.testing.run()
```

This will take between 10 and 30 minutes, depending on your computer. At the end, the testing report should report a success; it is normal for some tests to be skipped, and for some to be marked "xfail" in yellow. Skips may be tests that do not run on your operating system, or tests of optional components that you have not installed the dependencies for. If any failures or 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 or the GitHub issues page and post a message detailing your particular issue.

2.7 Checking Version Information

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

```
import qutip
qutip.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 https://qutip.org/tutorials.html, this guide should provide a more or less complete overview of QuTip. We also provide the API documentation in *API documentation*.

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 figure below.

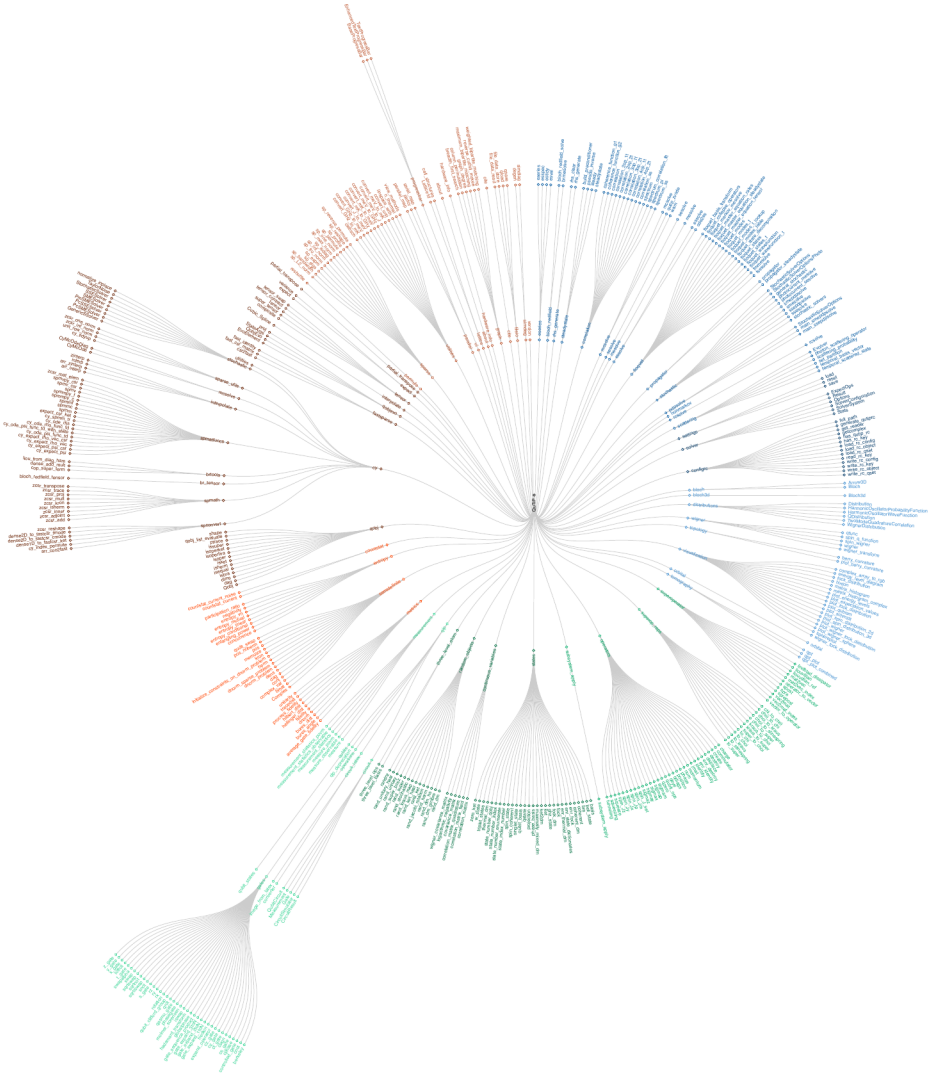


Fig. 1: Tree-diagram of the 468 user accessible functions and classes in QuTiP 4.6. A vector image of the code tree is in `qutip_tree.pdf`.

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, first call the import statement:

```
from qutip import *
```

This will load all of the user available functions. Often, we also need to import the NumPy and Matplotlib libraries with:

```
import numpy as np

import matplotlib.pyplot as plt
```

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 is 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 encapsulates 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`:

```
print(Qobj())
```

Output:

```
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, the names of Python classes, such as `Qobj()`, are capitalized whereas the names of functions are not.

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

```
print(Qobj([[1],[2],[3],[4],[5]]))
```

Output:

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

```
x = np.array([[1, 2, 3, 4, 5]])
print(Qobj(x))
```

Output:

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

```
r = np.random.rand(4, 4)
print(Qobj(r))
```

Output:

```
Quantum object: dims = [[4], [4]], shape = (4, 4), type = oper, isherm = False
Qobj data =
[[0.37454012 0.95071431 0.73199394 0.59865848]
 [0.15601864 0.15599452 0.05808361 0.86617615]
 [0.60111501 0.70807258 0.02058449 0.96990985]
 [0.83244264 0.21233911 0.18182497 0.18340451]]
```

Notice how both the dims and shape change according to the input data. Although dims and shape appear to be the same, dims keep track of the shapes for individual components of a multipartite system, while shape does not. We refer the reader to the section *tensor products and partial traces* for more information.

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 and operators:

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

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:

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

>>> coherent(5,0.5-0.5j)
Quantum object: dims = [[5], [1]], shape = (5, 1), type = ket
Qobj data =
[[ 0.7788017 +0.j        ]
 [ 0.38939142-0.38939142j]
 [ 0.        -0.27545895j]
 [-0.07898617-0.07898617j]
 [-0.04314271+0.j        ]]

>>> 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.        ]
```

(continues on next page)

(continued from previous page)

```
 [0.         0.         0.         1.73205081]
 [0.         0.         0.         0.        ]]

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

>>> 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:

```
>>> q = destroy(4)

>>> q.dims
[[4], [4]]

>>> q.shape
(4, 4)
```

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

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'?

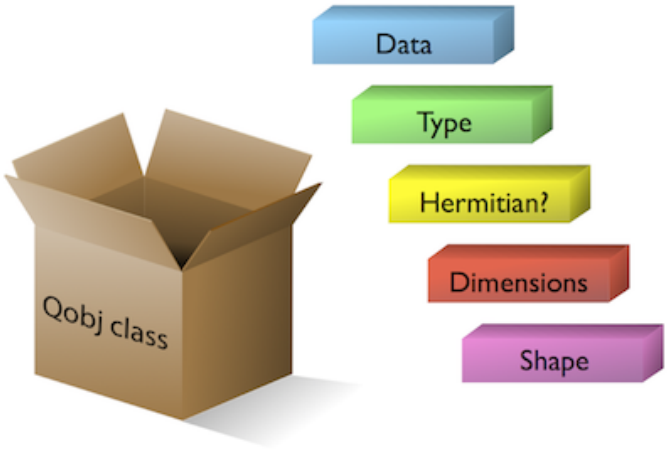


Fig. 2: The `Qobj` Class viewed as a container for the properties needed to characterize a quantum operator or state vector.

For the destruction operator above:

```
>>> q.type
'oper'

>>> q.isherm
False

>>> q.data
<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:

```
>>> q = destroy(4)

>>> x = sigmax()

>>> q + 5
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.        ]]

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

>>> q ** 3
Quantum object: dims = [[4], [4]], shape = (4, 4), type = oper, isherm = False
Qobj data =
```

(continues on next page)

(continued from previous page)

```
[[0.         0.         0.         2.44948974]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]
 [0.         0.         0.         0.        ]]

>>> 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:

```
>>> print(q * x)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-33-0b599f41213e> in <module>
----> 1 print(q * x)

~/Documents/qutip_dev/qutip/qutip/qobj.py in __mul__(self, other)
    553
    554             else:
--> 555                 raise TypeError("Incompatible Qobj shapes")
    556
    557         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().

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

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

>>> 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]
```

(continues on next page)

(continued from previous page)

```
  [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 ]]

>>> coherent_dm(5, 1).diag()
array([0.36791117, 0.36757705, 0.18523331, 0.05810197, 0.0211765 ])

>>> 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,
        0.035077  +0.j],
       [0.08826704+0.j, 0.08822695+0.j, 0.06263061+0.j, 0.035077  +0.j,
        0.0211765 +0.j]])

>>> coherent_dm(5, 1).norm()
1.0000000175063126

>>> coherent_dm(5, 1).sqrtm()
Quantum object: dims = [[5], [5]], shape = (5, 5), type = oper, isherm = False
Qobj data =
[[0.36791117+3.66778589e-09j 0.36774407-2.13388761e-09j
  0.26105441-1.51480558e-09j 0.14620658-8.48384618e-10j
  0.08826704-5.12182118e-10j]
 [0.36774407-2.13388761e-09j 0.36757705+2.41479965e-09j
  0.26093584-1.11446422e-09j 0.14614018+8.98971115e-10j
  0.08822695+6.40705133e-10j]
 [0.26105441-1.51480558e-09j 0.26093584-1.11446422e-09j
  0.18523331+4.02032413e-09j 0.10374209-3.39161017e-10j
  0.06263061-3.71421368e-10j]
 [0.14620658-8.48384618e-10j 0.14614018+8.98971115e-10j
  0.10374209-3.39161017e-10j 0.05810197+3.36300708e-10j
  0.035077  +2.36883273e-10j]
 [0.08826704-5.12182118e-10j 0.08822695+6.40705133e-10j
  0.06263061-3.71421368e-10j 0.035077  +2.36883273e-10j
  0.0211765 +1.71630348e-10j]]

>>> coherent_dm(5, 1).tr()
1.0

>>> (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:

```
vac = basis(5, 0)

print(vac)
```

Output:

```
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:

```
a = destroy(5)

print(a)
```

Output:

```
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`,

```
print(a * vac)
```

Output:

```
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 $\hat{a}^\dagger$:

```
print(a.dag() * vac)
```

Output:

```
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:

```
c = create(5)

print(c * vac)
```

Output:

```
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:

```
print(c * c * vac)
```

Output:

```
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 $\left(\hat{a}^\dagger\right)^2$:

```
print(c ** 2 * vac)
```

Output:

```
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`:

```
print(c * a * vac)
```

Output:

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

or on the $|1\rangle$ state:

```
print(c * a * (c * vac))
```

Output:

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

or the $|2\rangle$ state:

```
print(c * a * (c**2 * vac))
```

Output:

```
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:

```
print(c * a * (c**2 * vac).unit())
```

Output:

```
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:

```
ket = basis(5, 2)

print(ket)
```

Output:

```
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:

```
n = num(5)

print(n)
```

Output:

```
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:

```
print(n * ket)
```

Output:

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

We can also create superpositions of states:

```
ket = (basis(5, 0) + basis(5, 1)).unit()

print(ket)
```

Output:

```
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:

```
print(n * ket)
```

Output:

```
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:

```
vac = basis(5, 0)

d = displace(5, 1j)

s = squeeze(5, np.complex(0.25, 0.25))

print(d * vac)
```

Output:

```
Quantum object: dims = [[5], [1]], shape = (5, 1), type = ket
Qobj data =
[[ 0.60655682+0.j        ]
 [ 0.        +0.60628133j]
 [-0.4303874 +0.j        ]
 [ 0.        -0.24104351j]
 [ 0.14552147+0.j        ]]
```

```
print(d * s * vac)
```

Output:

```
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 no 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:

```
ket = basis(5, 2)

print(ket * ket.dag())
```

Output:

```
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:

```
print(fock_dm(5, 2))
```

Output:

```
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.]]
```

```
print(ket2dm(ket))
```

Output:

```
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:

```
print(0.5 * ket2dm(basis(5, 4)) + 0.5 * ket2dm(basis(5, 2)))
```

Output:

```
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.

```
print(coherent_dm(5, 1.25))
```

Output:

```
Quantum object: dims = [[5], [5]], shape = (5, 5), type = oper, isherm = True
Qobj data =
[[0.20980701 0.26141096 0.23509686 0.15572585 0.13390765]
```

(continues on next page)

(continued from previous page)

```
 [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 ]]
```

```
print(thermal_dm(5, 1.25))
```

Output:

```
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*, Bures angle *qutip.metrics.bures_angle*, and quantum Hellinger distance *qutip.metrics.hellinger_dist*.

```
x = coherent_dm(5, 1.25)

y = coherent_dm(5, np.complex(0, 1.25))  # <-- note the 'j'

z = thermal_dm(5, 0.125)

np.testing.assert_almost_equal(fidelity(x, x), 1)

np.testing.assert_almost_equal(hellinger_dist(x, y), 1.3819080728932833)
```

We also know that for two pure states, the trace distance (T) and the fidelity (F) are related by $T = \sqrt{1 - F^2}$, while the quantum Hellinger distance (QHE) between two pure states $|\psi\rangle$ and $|\phi\rangle$ is given by $QHE = \sqrt{2 - 2\,|\langle\psi|\phi\rangle|^2}$.

```
np.testing.assert_almost_equal(tracedist(y, x), np.sqrt(1 - fidelity(y, x) ** 2))
```

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

```
assert 1 - fidelity(x, z) ** 2 < tracedist(x, z)
```

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:

```
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?

```
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:

```
print(vac)
```

Output:

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

```
c = create(2)

print(c * vac)
```

Output:

```
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:

```
print(spin)
```

Output:

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

```
print(sigmap() * spin)
```

Output:

```
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:

```
print(sigmaz())
```

Output:

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

```
print(sigmaz() * spin)
```

Output:

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

```
spin2 = basis(2, 1)

print(spin2)
```

Output:

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

```
print(sigmaz() * spin2)
```

Output:

```
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:

```
print(sigmaz() * spin)
```

Output:

```
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:

```
vac = basis(5, 0)

one = basis(5, 1)

c = create(5)

N = num(5)

np.testing.assert_almost_equal(expect(N, vac), 0)

np.testing.assert_almost_equal(expect(N, one), 1)

coh = coherent_dm(5, 1.0j)

np.testing.assert_almost_equal(expect(N, coh), 0.9970555745806597)

cat = (basis(5, 4) + 1.0j * basis(5, 3)).unit()

np.testing.assert_almost_equal(expect(c, cat), 0.9999999999999998j)
```

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

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

print(expect(N, states))
```

Output:

```
[0. 1. 2. 3. 4.]
```

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

print(expect(c, cat_list))
```

Output:

```
[ 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, then 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:

```
up = basis(2, 0)

down = basis(2, 1)

np.testing.assert_almost_equal(expect(sigmaz(), up), 1)

np.testing.assert_almost_equal(expect(sigmaz(), down), -1)
```

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

```
spin1 = basis(2, 0)

spin2 = basis(2, 1)

two_spins = tensor(spin1, spin2)

sz1 = tensor(sigmaz(), qeye(2))

sz2 = tensor(qeye(2), sigmaz())

np.testing.assert_almost_equal(expect(sz1, two_spins), 1)

np.testing.assert_almost_equal(expect(sz2, two_spins), -1)
```

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 correspondence between linear operators acting on a Hilbert space, and vectors in two copies of that Hilbert space, $\mathrm{vec} : \mathcal{L}(\mathcal{H}) \to \mathcal{H} \otimes \mathcal{H}$ [Hav03], [Wat13].

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

```
psi = basis(2, 0)

rho = ket2dm(psi)

print(rho)
```

Output:

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

```
vec_rho = operator_to_vector(rho)

print(vec_rho)
```

Output:

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

```
rho2 = vector_to_operator(vec_rho)

np.testing.assert_almost_equal((rho - rho2).norm(), 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 $\mathcal{H} \otimes \mathcal{H}$:

```
A = Qobj(np.arange(4).reshape((2, 2)))

print(A)
```

Output:

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

```
print(operator_to_vector(A))
```

Output:

```
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.

```
X = sigmax()

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.

```
S2 = to_super(X)

np.testing.assert_almost_equal((S - S2).norm(), 0)
```

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

```
print(S)
```

Output:

```
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:

```
print(S.iscp, S.istp, S.iscptp)
```

Output:

```
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 Hamiltonian along with a list of collapse operators, and returns a `type="super"` object that can be exponentiated to find the superoperator for that evolution.

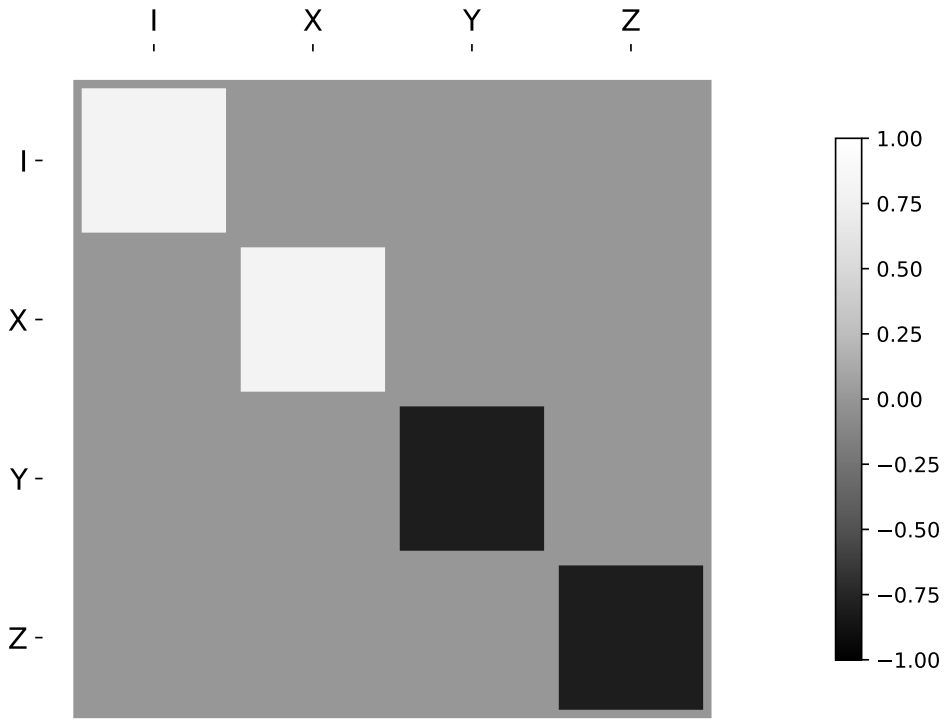
```
H = 10 * sigmaz()

c1 = destroy(2)

L = liouvillian(H, [c1])

print(L)

S = (12 * L).expm()
```

Output:

```
Quantum object: dims = [[[2], [2]], [[2], [2]]], shape = (4, 4), type = super,␣
↪isherm = False
Qobj data =
[[ 0.  +0.j  0.  +0.j  0.  +0.j  1.  +0.j]
 [ 0.  +0.j -0.5+20.j  0.  +0.j  0.  +0.j]
 [ 0.  +0.j  0.  +0.j -0.5-20.j  0.  +0.j]
 [ 0.  +0.j  0.  +0.j  0.  +0.j -1.  +0.j]]
```

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:

```
from qutip import *
settings.colorblind_safe = True

import matplotlib.pyplot as plt
plt.rcParams['savefig.transparent'] = True

X = sigmax()
S = spre(X) * spost(X.dag())

hinton(S)
```



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*.

```
X = sigmax()

S = sprepost(X, X)

J = to_choi(S)

print(J)
```

Output:

```
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.]]
```

```
print(to_choi(spre(qeye(2))))
```

Output:

```
Quantum 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:

```
print(to_choi(J))
```

Output:

```
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:

```
print(to_super(J) - S)
```

Output:

```
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.]]
```

```
print(to_super(S))
```

Output:

```
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.]]
```

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

$$\sum_i (\mathbb{1} \otimes A_i)(\mathbb{1} \otimes A_i)^\dagger |\mathbb{1}\rangle\rangle\langle\langle\mathbb{1}| = \sum_i |A_i\rangle\rangle\langle\langle A_i| = J(\Lambda).$$

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

```
del sum # np.sum overwrote sum and caused a bug.
```

```
I, X, Y, Z = qeye(2), sigmax(), sigmay(), sigmaz()
```

```
S = sum([sprepost(P, P) for P in (I, X, Y, Z)]) / 4
print(S)
```

Output:

```
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]]
```

```
J = to_choi(S)
print(J)
```

Output:

```
Quantum 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]]
```

```
print(J.eigenstates()[1])
```

Output:

```
[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 =
```

(continues on next page)

(continued from previous page)

```
[[0.]
 [0.]
 [0.]
 [1.]]]
```

```
K = to_kraus(S)
print(K)
```

Output:

```
[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]]]
```

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`.

```
KJ = to_kraus(J)
print(KJ)
```

Output:

```
[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]]]
```

```
for A, AJ in zip(K, KJ):
  print(A - AJ)
```

Output:

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

(continues on next page)

(continued from previous page)

```
 [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 $\mathcal{H}$,

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

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.

```
a = create(2).dag()

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)
```

Output:

```
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]]
```

```
A, B = to_stinespring(S)
print(A)
```

Output:

```
Quantum object: dims = [[2, 3], [2]], shape = (6, 2), type = oper, isherm = False
Qobj data =
[[-0.98845443  0.        ]
 [ 0.          0.31622777]
 [ 0.15151842  0.        ]
 [ 0.         -0.93506452]
 [ 0.          0.        ]
 [ 0.         -0.16016975]]
```

```
print(B)
```

Output:

```
Quantum object: dims = [[2, 3], [2]], shape = (6, 2), type = oper, isherm = False
Qobj data =
[[-0.98845443  0.        ]
```

(continues on next page)

(continued from previous page)

```
 [ 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).

```
to_kraus(S)
print(to_choi(S).eigenenergies())
```

Output:

```
[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.

```
chi = to_chi(S)
print(chi)
```

Output:

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

```
np.testing.assert_almost_equal(average_gate_fidelity(S), 0.9499999999999998)

print(chi[0, 0] / 4)
```

Output:

```
(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.

```
rho = ket2dm(bell_state())
rho_out = partial_transpose(rho, [0, 1])
print(rho_out.eigenenergies())
```

Output:

```
[-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.

```
J = rho_out
J.dims = [[[2], [2]], [[2], [2]]]
J.superrep = 'choi'
print(J.iscp)
```

Output:

```
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 $(\rho^{\mathrm{T}})^\dagger = \rho^*$, the complex conjugate of ρ. By assumption, $\rho = \rho^\dagger = (\rho^*)^{\mathrm{T}}$, though, such that $\Lambda(\rho) = \Lambda(\rho^\dagger) = \rho^*$. We can confirm this by checking the `ishp` attribute:

```
print(J.ishp)
```

Output:

```
True
```

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

```
print(J.istp)
```

Output:

```
False
```

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.

```
>>> print(J.iscptp)
False

>>> 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 arbitrary 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:

```
print(tensor(basis(2, 0), basis(2, 0)))
```

Output:

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

or equivalently using the `list` format:

```
print(tensor([basis(2, 0), basis(2, 0)]))
```

Output:

```
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:

```
print(tensor((basis(2, 0) + basis(2, 1)).unit(), (basis(2, 0) + basis(2, 1)).
↪unit(), basis(2, 0)))
```

Output:

```
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:

```
print(tensor(sigmax(), sigmax()))
```

Output:

```
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 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:

```
print(tensor(sigmaz(), identity(2)))
```

Output:

```
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 the qubits have 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:

```
H = tensor(sigmaz(), identity(2)) + tensor(identity(2), sigmaz()) + 0.05 *␣
↪tensor(sigmax(), sigmax())

print(H)
```

Output:

```
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:

```
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()))

print(H)
```

Output:

```
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-Cummings Hamiltonian can be constructed as:

```
N = 10

omega_a = 1.0

omega_c = 1.25

g = 0.05

a = tensor(identity(2), destroy(N))

sm = tensor(destroy(2), identity(N))

sz = tensor(sigmaz(), identity(N))

H = 0.5 * omega_a * sz + omega_c * a.dag() * a + g * (a.dag() * sm + a * sm.dag())

print(H)
```

Output:

```
Quantum object: dims = [[2, 10], [2, 10]], shape = (20, 20), type = oper, isherm =␣
↪True
Qobj data =
[[ 0.5         0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.          0.          0.          0.
   0.          0.        ]
 [ 0.          1.75        0.          0.          0.          0.
   0.          0.          0.          0.          0.05        0.
   0.          0.          0.          0.          0.          0.
   0.          0.        ]
 [ 0.          0.          3.          0.          0.          0.
   0.          0.          0.          0.          0.          0.07071068
   0.          0.          0.          0.          0.          0.
   0.          0.        ]
 [ 0.          0.          0.          4.25        0.          0.
   0.          0.          0.          0.          0.          0.
   0.08660254  0.          0.          0.          0.          0.
   0.          0.        ]
 [ 0.          0.          0.          0.          5.5         0.
   0.          0.          0.          0.          0.          0.
   0.          0.1         0.          0.          0.          0.
   0.          0.        ]
 [ 0.          0.          0.          0.          0.          6.75
   0.          0.          0.          0.          0.          0.
   0.          0.          0.1118034   0.          0.          0.
   0.          0.        ]
 [ 0.          0.          0.          0.          0.          0.
   8.          0.          0.          0.          0.          0.
   0.          0.          0.          0.12247449  0.          0.
   0.          0.        ]
 [ 0.          0.          0.          0.          0.          0.
   0.          9.25        0.          0.          0.          0.
   0.          0.          0.          0.          0.13228757  0.
   0.          0.        ]
```

(continues on next page)

(continued from previous page)

```
[ 0.          0.          0.          0.          0.          0.
  0.          0.         10.5         0.          0.          0.
  0.          0.          0.          0.          0.          0.14142136
  0.          0.        ]
[ 0.          0.          0.          0.          0.          0.
  0.          0.          0.         11.75        0.          0.
  0.          0.          0.          0.          0.          0.
  0.15        0.        ]
[ 0.          0.05        0.          0.          0.          0.
  0.          0.          0.          0.         -0.5         0.
  0.          0.          0.          0.          0.          0.
  0.          0.        ]
[ 0.          0.          0.07071068  0.          0.          0.
  0.          0.          0.          0.          0.          0.75
  0.          0.          0.          0.          0.          0.
  0.          0.        ]
[ 0.          0.          0.          0.08660254  0.          0.
  0.          0.          0.          0.          0.          0.
  2.          0.          0.          0.          0.          0.
  0.          0.        ]
[ 0.          0.          0.          0.          0.1         0.
  0.          0.          0.          0.          0.          0.
  0.          3.25        0.          0.          0.          0.
  0.          0.        ]
[ 0.          0.          0.          0.          0.          0.1118034
  0.          0.          0.          0.          0.          0.
  0.          0.          4.5         0.          0.          0.
  0.          0.        ]
[ 0.          0.          0.          0.          0.          0.
  0.12247449  0.          0.          0.          0.          0.
  0.          0.          0.          5.75        0.          0.
  0.          0.        ]
[ 0.          0.          0.          0.          0.          0.
  0.          0.13228757  0.          0.          0.          0.
  0.          0.          0.          0.          7.          0.
  0.          0.        ]
[ 0.          0.          0.          0.          0.          0.
  0.          0.          0.14142136  0.          0.          0.
  0.          0.          0.          0.          0.          8.25
  0.          0.        ]
[ 0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.15        0.          0.
  0.          0.          0.          0.          0.          0.
  9.5         0.        ]
[ 0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.          0.          0.
  0.         10.75      ]]
```

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:

```
>>> psi = tensor(basis(2, 0), basis(2, 1))

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

>>> 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):

```
>>> psi = tensor((basis(2, 0) + basis(2, 1)).unit(), basis(2, 0))

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

>>> 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]]

>>> rho = tensor(ket2dm((basis(2, 0) + basis(2, 1)).unit()), fock_dm(2, 0))

>>> rho
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. ]]

>>> 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.superop_reps.to_super*:

```
>>> A = qeye([2])

>>> B = qeye([3])

>>> to_super(tensor(A, B)).dims
[[[2, 3], [2, 3]], [[2, 3], [2, 3]]]

>>> tensor(to_super(A), to_super(B)).dims
[[[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.tensor.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:

```
>>> super_tensor(to_super(A), to_super(B)).dims
[[[2, 3], [2, 3]], [[2, 3], [2, 3]]]
```

The *qutip.tensor.composite* function automatically switches between *qutip.tensor* and *qutip.tensor.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.

```
>>> composite(A, B).dims
[[2, 3], [2, 3]]

>>> composite(to_super(A), to_super(B)).dims
[[[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:

```
>>> tensor_contract(composite(to_super(A), to_super(B)), (1, 3), (4, 6)).dims
[[[2], [2]], [[3], [3]]]
```

3.5 Time Evolution and Quantum System Dynamics

3.5.1 Introduction

Although in some cases, we want to find the stationary states of a quantum system, often we are interested in the dynamics: how the state of a system or an ensemble of systems evolves with time. QuTiP provides many ways to model dynamics.

Broadly speaking, there are two categories of dynamical models: unitary and non-unitary. In unitary evolution, the state of the system remains normalized. In non-unitary, or dissipative, systems, it does not.

There are two kinds of quantum systems: open systems that interact with a larger environment and closed systems that do not. In a closed system, the state can be described by a state vector, although when there is entanglement a density matrix may be needed instead. When we are modeling an open system, or an ensemble of systems, the use of the density matrix is mandatory.

Collapse operators are used to model the collapse of the state vector that can occur when a measurement is performed.

The following tables lists some of the solvers QuTiP provides for dynamic quantum systems and indicates the type of object returned by the solver:

Table 1: QuTiP Solvers

Solver	Returns	Remarks
sesolve()	*qutip.solver.Result*	Unitary evolution, single system
mesolve()	*qutip.solver.Result*	Lindblad master eqn. or Von Neuman eqn. Density matrix.
mcsolve()	*qutip.solver.Result*	Monte Carlo with collapse operators
essolve()	Array of expectation values	Exponential series with collapse operators
bloch_redfield_solve()	`qutip.solver`	
floquet_markov_solve()	*qutip.solver.Result*	Floquet-Markov master equation
fmmesolve()	`qutip.solver`	Floquet-Markov master equation
smesolve()	*qutip.solver.Result*	Stochastic master equation
ssesolve()	*qutip.solver.Result*	Stochastic Schrödinger equation

3.5.2 Dynamics Simulation Results

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.

3.5.3 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

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

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.sesolve*.

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.

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

```
>>> H = 2*np.pi * 0.1 * sigmax()
>>> psi0 = basis(2, 0)
>>> times = np.linspace(0.0, 10.0, 20)
>>> result = sesolve(H, psi0, times, [sigmaz()])
```

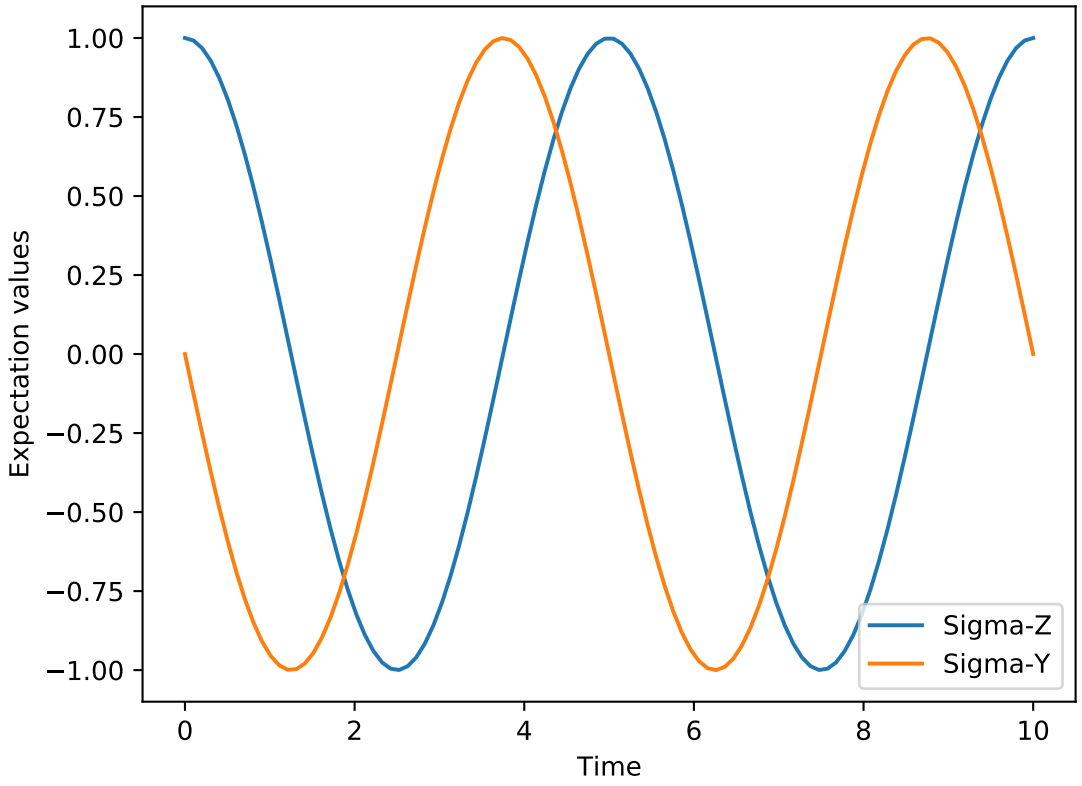
See the next section for examples on how dissipation is included by defining a list of collapse operators and using *qutip.mesolve* instead.

The function *qutip.sesolve* 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 fourth 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).

```
>>> result = sesolve(H, psi0, times, [sigmaz(), sigmay()])
>>> result.expect
[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:

```
>>> H = 2*np.pi * 0.1 * sigmax()
>>> psi0 = basis(2, 0)
>>> times = np.linspace(0.0, 10.0, 100)
>>> result = sesolve(H, psi0, times, [sigmaz(), sigmay()])
>>> fig, ax = plt.subplots()
>>> ax.plot(result.times, result.expect[0])
>>> ax.plot(result.times, result.expect[1])
>>> ax.set_xlabel('Time')
>>> ax.set_ylabel('Expectation values')
>>> ax.legend(("Sigma-Z", "Sigma-Y"))
>>> plt.show()
```



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

```
>>> times = [0.0, 1.0]
>>> result = sesolve(H, psi0, times, [])
>>> result.states
[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.        -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

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

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

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

includes the original system Hamiltonian $H_{\rm sys}$, the Hamiltonian for the environment $H_{\rm env}$, and a term representing the interaction between the system and its environment $H_{\rm int}$. Since we are only interested in the dynamics of the system, we can at this point perform a partial trace over the environmental degrees of freedom in Eq. (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 $\rho = {\rm Tr}_{\rm env}[\rho_{\rm tot}]$

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

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 $H_{\rm int}$, 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 $\rho^I_{\rm tot}(0) = \rho^I(0) \otimes \rho^I_{\rm env}(0)$.
- **Born approximation:** Requires: (1) that the state of the environment does not significantly change as a result of the interaction with the system; (2) The system and the environment remain separable throughout the evolution. These assumptions are justified if the interaction is weak, and if the environment is much larger than the system. In summary, $\rho_{\rm tot}(t) \approx \rho(t) \otimes \rho_{\rm env}$.
- **Markov approximation** The time-scale of decay for the environment $\tau_{\rm env}$ is much shorter than the smallest time-scale of the system dynamics $\tau_{\rm sys} \gg \tau_{\rm env}$. This approximation is often deemed a "short-memory environment" as it requires that environmental correlation functions decay on a time-scale fast compared to those of the system.
- **Secular approximation** Stipulates that elements in the master equation corresponding to transition frequencies satisfy $|\omega_{ab} - \omega_{cd}| \ll 1/\tau_{\rm sys}$, i.e., all fast rotating terms in the interaction picture can be neglected. It also ignores terms that lead to a small renormalization of the system energy levels. This approximation is not strictly necessary for all master-equation formalisms (e.g., the Block-Redfield master equation), but it is required for arriving at the Lindblad form (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.

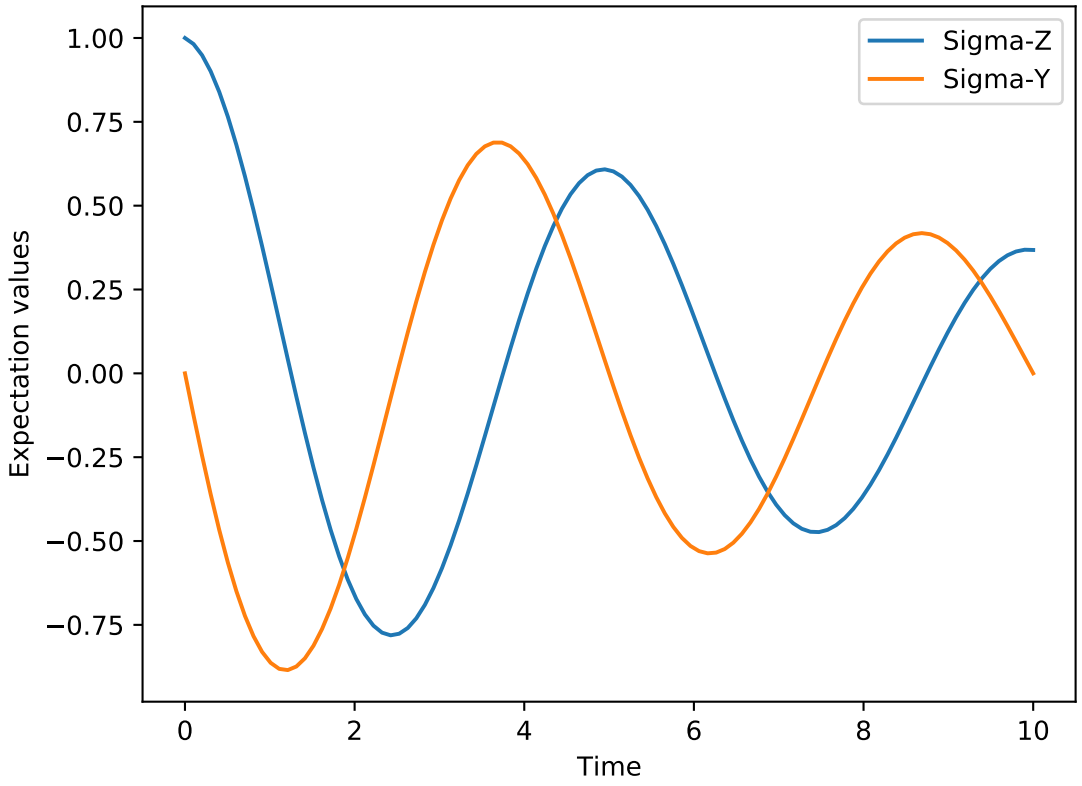
For non-unitary evolution of a quantum systems, i.e., evolution that includes incoherent processes such as relaxation and dephasing, it is common to use master equations. In QuTiP, the function *qutip.mesolve* is used for both: the evolution according to the Schrödinger equation and to the master equation, 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()` in the fourth parameter to the *qutip.mesolve* function and moving the expectation operators `[sigmaz(), sigmay()]` to the fifth argument.

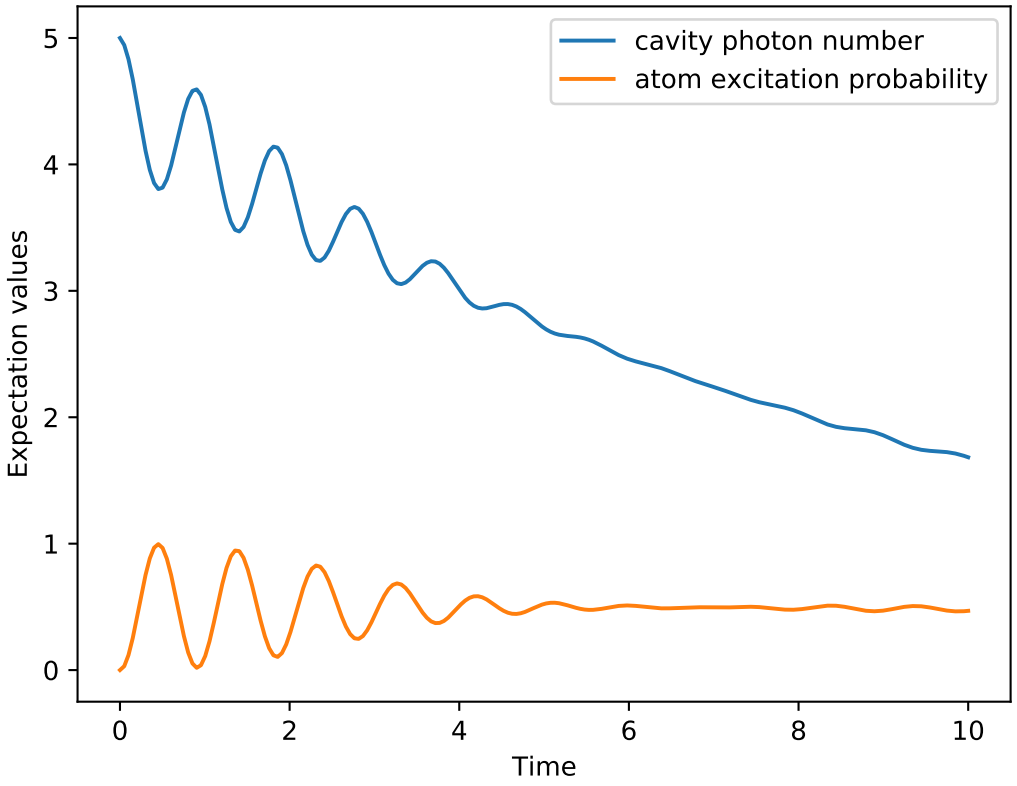
```
>>> times = np.linspace(0.0, 10.0, 100)
>>> result = mesolve(H, psi0, times, [np.sqrt(0.05) * sigmax()], [sigmaz(),␣
↪sigmay()])
>>> fig, ax = plt.subplots()
>>> ax.plot(times, result.expect[0])
>>> ax.plot(times, result.expect[1])
>>> ax.set_xlabel('Time')
>>> ax.set_ylabel('Expectation values')
>>> ax.legend(("Sigma-Z", "Sigma-Y"))
>>> plt.show()
```



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

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

```
>>> times = np.linspace(0.0, 10.0, 200)
>>> psi0 = tensor(fock(2,0), fock(10, 5))
>>> a  = tensor(qeye(2), destroy(10))
>>> sm = tensor(destroy(2), qeye(10))
>>> H = 2 * np.pi * a.dag() * a + 2 * np.pi * sm.dag() * sm + 2 * np.pi * 0.25 *␣
↪(sm * a.dag() + sm.dag() * a)
>>> result = mesolve(H, psi0, times, [np.sqrt(0.1)*a], [a.dag()*a, sm.dag()*sm])
>>> plt.figure()
>>> plt.plot(times, result.expect[0])
>>> plt.plot(times, result.expect[1])
>>> plt.xlabel('Time')
>>> plt.ylabel('Expectation values')
>>> plt.legend(("cavity photon number", "atom excitation probability"))
>>> plt.show()
```



3.5.4 Monte Carlo Solver

Introduction

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

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

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

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

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

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

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

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

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 $|\psi(0)\rangle$:

- **Ia:** Choose a random number r_1 between zero and one, representing the probability that a quantum jump occurs.
- **Ib:** Choose a random number r_2 between zero and one, used to select which collapse operator was responsible for the jump.
- **II:** Integrate the Schrödinger equation, using the effective Hamiltonian (3.4) until a time τ such that the norm of the wave function satisfies $\langle\psi(\tau)\ |\psi(\tau)\rangle = r_1$, 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:

$$\sum_{i=1}^{n} P_n(\tau) \geq r_2 \tag{3.8}$$

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.

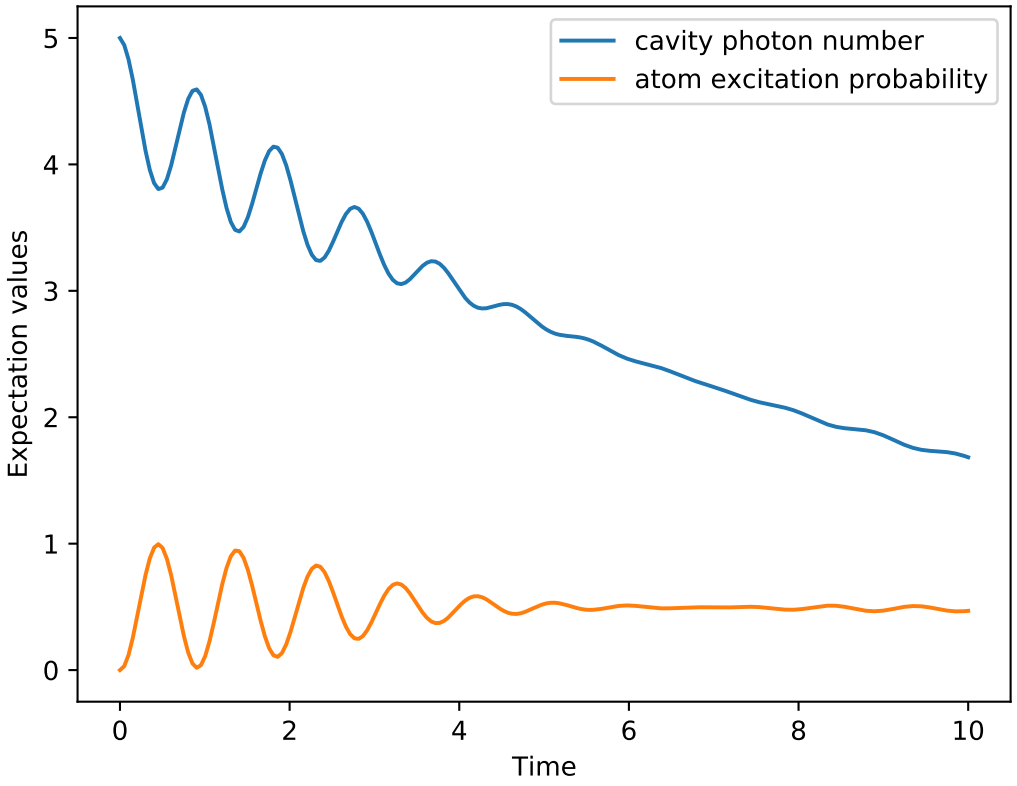
- **IV:** Using the renormalized state from step III as the new initial condition at time τ, draw a new random number, and repeat the above procedure until the final simulation time is reached.

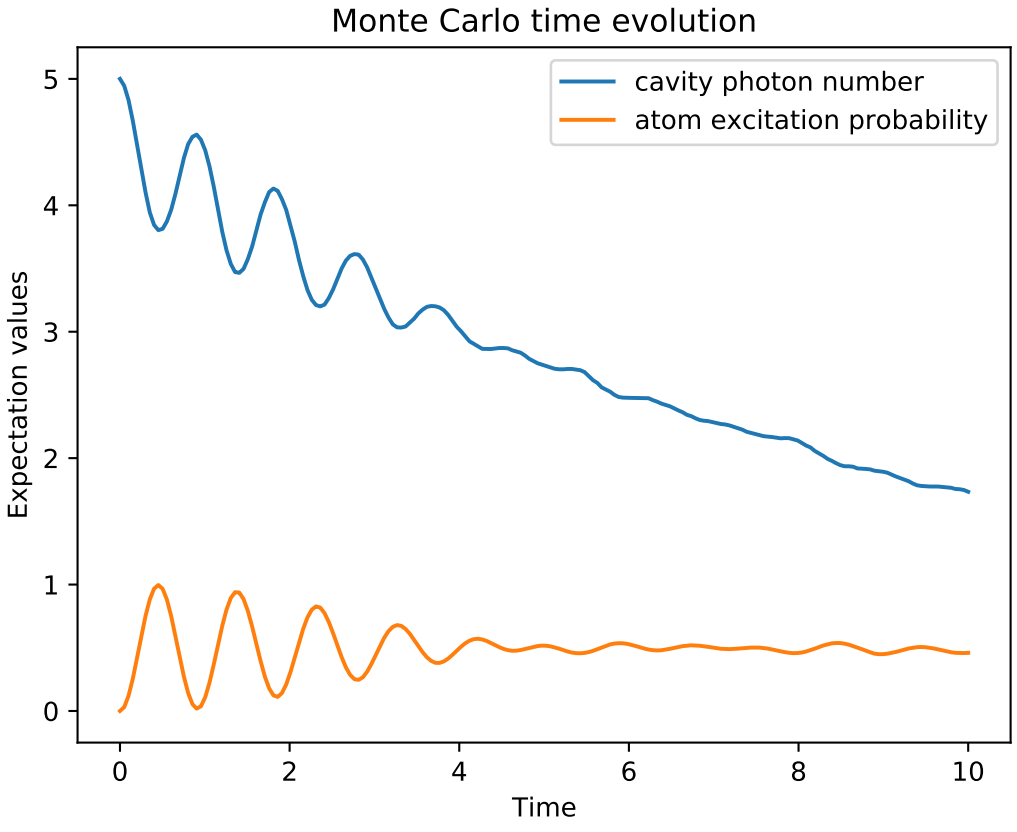
Monte Carlo in QuTiP

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*

```
times = np.linspace(0.0, 10.0, 200)
psi0 = tensor(fock(2, 0), fock(10, 5))
a  = tensor(qeye(2), destroy(10))
sm = tensor(destroy(2), qeye(10))
H = 2*np.pi*a.dag()*a + 2*np.pi*sm.dag()*sm + 2*np.pi*0.25*(sm*a.dag() + sm.
↪dag()*a)
data = mcsolve(H, psi0, times, [np.sqrt(0.1) * a], [a.dag() * a, sm.dag() * sm])

plt.figure()
plt.plot(times, data.expect[0], times, data.expect[1])
plt.title('Monte Carlo time evolution')
plt.xlabel('Time')
plt.ylabel('Expectation values')
plt.legend(("cavity photon number", "atom excitation probability"))
plt.show()
```


cavity photon number
atom excitation probability
Expectation values
Time
0
1
2
3
4
5
0
2
4
6
8
10


Monte Carlo time evolution
cavity photon number
atom excitation probability
Expectation values
Time
0
1
2
3
4
5
0
2
4
6
8
10

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:

```
data = mcsolve(H, psi0, times, [np.sqrt(0.1) * a], [a.dag() * a, sm.dag() * sm],␣
↪ntraj=1000)
```

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

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

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

```
data = mcsolve(H, psi0, times, [np.sqrt(0.1) * a], [a.dag() * a, sm.dag() * sm],␣
↪ntraj=[1, 10, 100, 1000])
```

we can extract the relevant expectation values using:

```
expt1 = data.expect[0]
expt10 = data.expect[1]
expt100 = data.expect[2]
expt1000 = data.expect[3]
```

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.

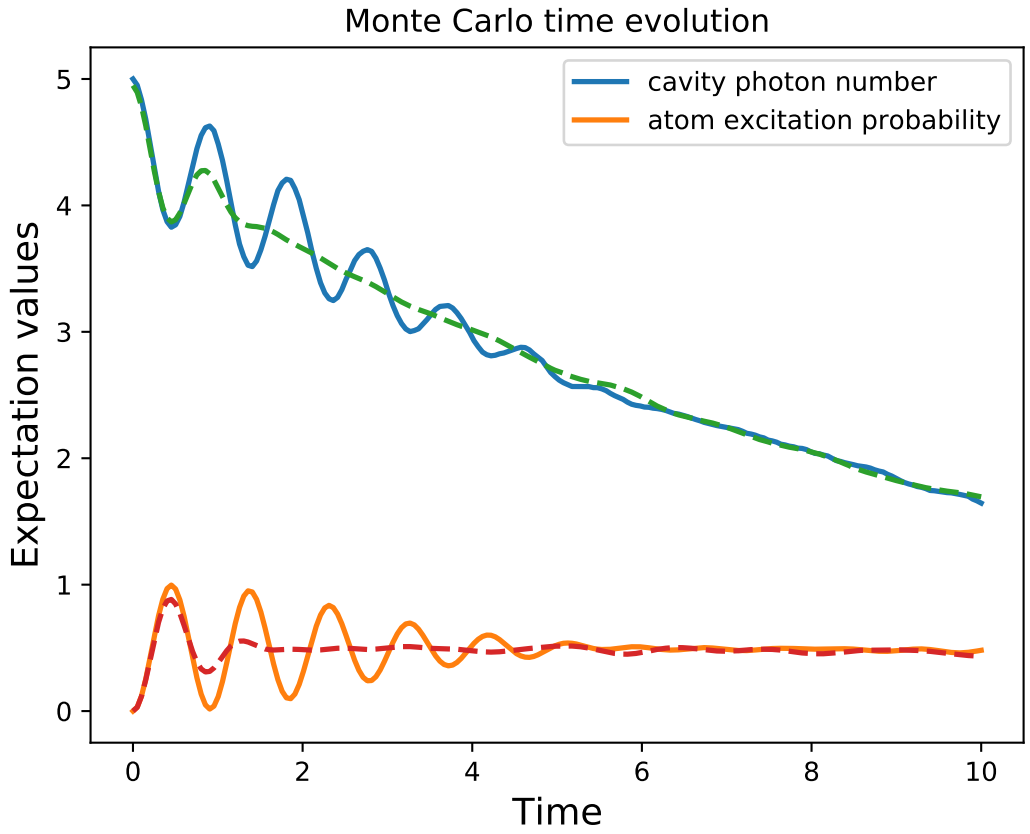
In order to solve a given simulation as fast as possible, the solvers in QuTiP take the given input operators and break them down into simpler components before passing them on to the ODE solvers. Although these operations are reasonably fast, the time spent organizing data can become appreciable when 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*:

```
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

```
times = np.linspace(0.0, 10.0, 200)
psi0 = tensor(fock(2, 0), fock(10, 5))
a  = tensor(qeye(2), destroy(10))
sm = tensor(destroy(2), qeye(10))

H = 2*np.pi*a.dag()*a + 2*np.pi*sm.dag()*sm + 2*np.pi*0.25*(sm*a.dag() + sm.
↪dag()*a)
data1 = mcsolve(H, psi0, times, [np.sqrt(0.1) * a], [a.dag() * a, sm.dag() * sm])
psi1 = tensor(fock(2, 0), coherent(10, 2 - 1j))
opts = Options(rhs_reuse=True) # Run a second time, reusing RHS
data2 = mcsolve(H, psi1, times, [np.sqrt(0.1) * a], [a.dag() * a, sm.dag() * sm],␣
↪options=opts)

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

3.5.5 Krylov Solver

Introduction

The Krylov-subspace method is a standard method to approximate quantum dynamics. Let $|\psi\rangle$ be a state in a D-dimensional complex Hilbert space that evolves under a time-independent Hamiltonian H. Then, the N-dimensional Krylov subspace associated with that state and Hamiltonian is given by

$$\mathcal{K}_N = \text{span}\left\{|\psi\rangle, H|\psi\rangle, \ldots, H^{N-1}|\psi\rangle\right\}, \tag{3.9}$$

where the dimension $N < D$ is a parameter of choice. To construct an orthonormal basis B_N for $\mathcal{K}_N$, the simplest algorithm is the well-known Lanczos algorithm, which provides a sort of Gram-Schmidt procedure that harnesses the fact that orthonormalization needs to be imposed only for the last two vectors in the basis. Written in this basis the time-evolved state can be approximated as

$$|\psi(t)\rangle = e^{-iHt}|\psi\rangle \approx \mathbb{P}_N e^{-iHt}\mathbb{P}_N|\psi\rangle = \mathbb{V}_N^\dagger e^{-iT_N t}\mathbb{V}_N|\psi\rangle \equiv |\psi_N(t)\rangle\,, \tag{3.10}$$

where $T_N = \mathbb{V}_N H \mathbb{V}_N^\dagger$ is the Hamiltonian reduced to the Krylov subspace (which takes a tridiagonal matrix form), and $\mathbb{V}_N^\dagger$ is the matrix containing the vectors of the Krylov basis as columns.

With the above approximation, the time-evolution is calculated only with a smaller square matrix of the desired size. Therefore, the Krylov method provides huge speed-ups in computation of short-time evolutions when the dimension of the Hamiltonian is very large, a point at which exact calculations on the complete subspace are practically impossible.

One of the biggest problems with this type of method is the control of the error. After a short time, the error starts to grow exponentially. However, this can be easily corrected by restarting the subspace when the error reaches a certain threshold. Therefore, a series of M Krylov-subspace time evolutions provides accurate solutions for the complete time evolution. Within this scheme, the magic of Krylov resides not only in its ability to capture complex time evolutions from very large Hilbert spaces with very small dimenions M, but also in the computing speed-up it presents.

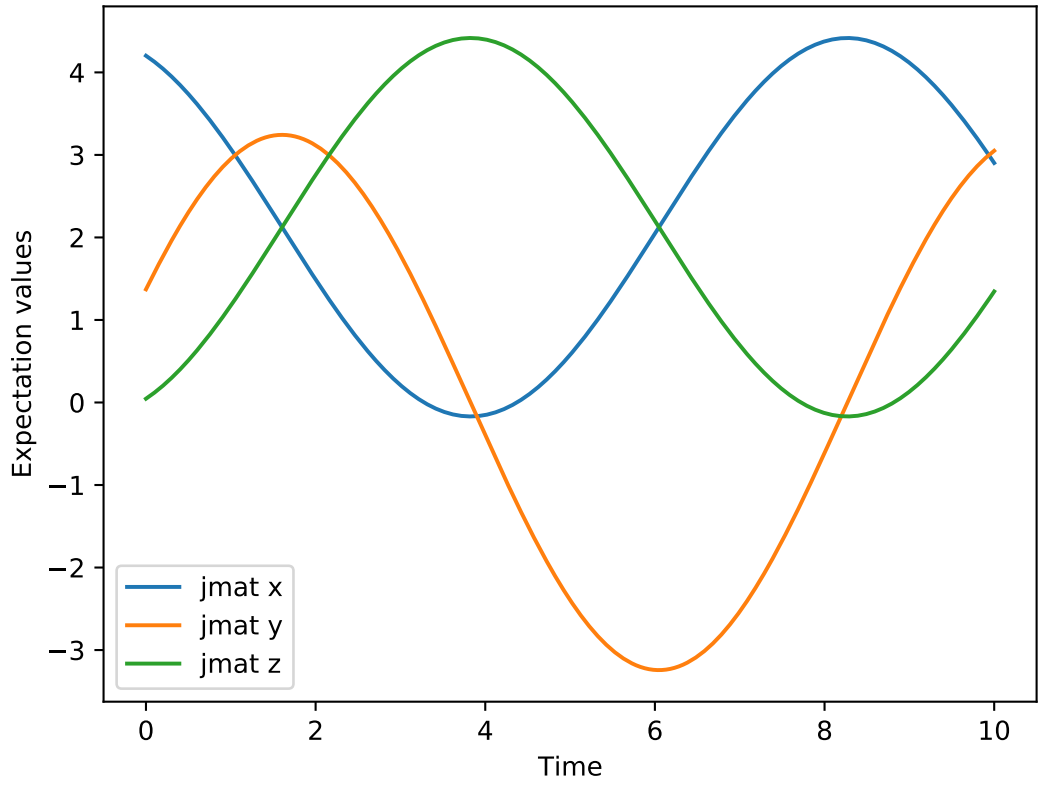
For exceptional cases, the Lanczos algorithm might arrive at the exact evolution of the initial state at a dimension $M_{hb} < M$. This is called a happy breakdown. For example, if a Hamiltonian has a symmetry subspace $D_{\text{sim}} < M$, then the algorithm will optimize using the value math:*M_{hb}<M*:, at which the evolution is not only exact but also cheap.

Krylov Solver in QuTiP

In QuTiP, Krylov-subspace evolution is implemented as the function *qutip.krylovsolve*. Arguments are nearly the same as *qutip.mesolve* function for master-equation evolution, except that the initial state must be a ket vector, as opposed to a density matrix, and the additional parameter `krylov_dim` that defines the maximum allowed Krylov-subspace dimension. The maximum number of allowed Lanczos partitions can also be determined using the `qutip.solver.options.nsteps` parameter, which defaults to '10000'.

Let's solve a simple example using the algorithm in QuTiP to get familiar with the method.

```
>>> from qutip import jmat, rand_ket, krylovsolve
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> dim = 100
>>> e_ops = [jmat((dim - 1) / 2.0, "x"), jmat((dim - 1) / 2.0, "y"), jmat((dim -␣
↪1) / 2.0, "z")]
>>> H = .5*jmat((dim - 1) / 2.0, "z") + .5*jmat((dim - 1) / 2.0, "x")
>>> psi0 = rand_ket(dim)
>>> tlist = np.linspace(0.0, 10.0, 200)
>>> results = krylovsolve(H, psi0, tlist, krylov_dim=20, e_ops=e_ops)
>>> plt.figure()
>>> for expect in results.expect:
>>>    plt.plot(tlist, expect)
>>> plt.legend(('jmat x', 'jmat y', 'jmat z'))
>>> plt.xlabel('Time')
>>> plt.ylabel('Expectation values')
>>> plt.show()
```



Sparse and Dense Hamiltonians

If the Hamiltonian of interest is known to be sparse, *qutip.krylovsolve* also comes equipped with the possibility to store its internal data in a sparse optimized format using scipy. This allows for significant speed-ups, let's showcase it:

```
>>> from qutip import krylovsolve, rand_herm, rand_ket
>>> import numpy as np
>>> from time import time
>>> def time_krylov(psi0, H, tlist, sparse):
>>>     start = time()
>>>     krylovsolve(H, psi0, tlist, krylov_dim=30, sparse=sparse)
>>>     end = time()
>>>     return end - start
>>> dim = 2000
>>> tlist = np.linspace(0, 1, 10)
>>> psi0 = rand_ket(dim, seed=0)
>>> H_sparse = rand_herm(dim, density=0.1, seed=0)
>>> H_dense = rand_herm(dim, density=0.9, seed=0)
>>> # first index for type of H and second index for sparse = True or False (dense)
>>> t_ss = time_krylov(psi0, H_sparse, tlist, sparse=True)
>>> t_sd = time_krylov(psi0, H_sparse, tlist, sparse=False)
>>> t_ds = time_krylov(psi0, H_dense, tlist, sparse=True)
>>> t_dd = time_krylov(psi0, H_dense, tlist, sparse=False)
>>> print(f"Average time of solution for a sparse H is {round((t_sd)/t_ss, 2)}␣
↪faster for sparse=True in comparison to sparse=False")
>>> print(f"Average time of solution for a dense H is {round((t_dd)/t_ds, 2)}␣
↪slower for sparse=True in comparison to sparse=False")
Average time of solution for a sparse H is 2.46 faster for sparse=True in␣
↪comparison to sparse=False
```

(continues on next page)

(continued from previous page)

```
Average time of solution for a dense H is 0.45 slower for sparse=True in␣
↪comparison to sparse=False
```

3.5.6 Stochastic Solver - Photocurrent

Photocurrent method, like monte-carlo method, allows for simulating an individual realization of the system evolution under continuous measurement.

Closed system

Photocurrent evolution have the state evolve deterministically between quantum jumps. During the deterministic part, the system evolve by schrodinger equation with a non-hermitian, norm conserving effective Hamiltonian.

$$H_{\rm eff} = H_{\rm sys} + \frac{i\hbar}{2}\left(-\sum_n C_n^+ C_n + |C_n\psi|^2\right). \tag{3.11}$$

With C_n, the collapse operators. This effective Hamiltonian is equivalent to the monte-carlo effective Hamiltonian with an extra term to keep the state normalized. At each time step of δt, the wave function has a probability

$$\delta p_n = \left\langle \psi(t)|C_n^+ C_n|\psi(t)\right\rangle \delta t \tag{3.12}$$

of making a quantum jump. δt must be chosen small enough to keep that probability small $\delta p << 1$. *If multiple jumps happen at the same time step, the state become unphysical.* Each jump result in a sharp variation of the state by,

$$\delta\psi = \left(\frac{C_n\psi}{|C_n\psi|} - \psi\right) \tag{3.13}$$

The basic photocurrent method directly integrates these equations to the first-order. Starting from a state $|\psi(0)\rangle$, it evolves the state according to

$$\delta\psi(t) = -iH_{\rm sys}\psi(t)\delta t + \sum_n \left(-\frac{C_n^+ C_n}{2}\psi(t)\delta t + \frac{|C_n\psi|^2}{2}\delta t + \delta N_n\left(\frac{C_n\psi}{|C_n\psi|} - \psi\right)\right), \tag{3.14}$$

for each time-step. Here $\delta N = 1$ with a probability of $\delta\omega$ and $\delta N_n = 0$ with a probability of $1 - \delta\omega$.

Trajectories obtained with this algorithm are equivalent to those obtained with monte-carlo evolution (up to $O(\delta t^2)$). In most cases, `qutip.mcsolve` is more efficient than `qutip.stochastic.photocurrent_sesolve`.

Open system

Photocurrent approach allows to obtain trajectories for a system with both measured and dissipative interaction with the bath. The system evolves according to the master equation between jumps with a modified liouvillian

$$L_{\rm eff}(\rho(t)) = L_{\rm sys}(\rho(t)) + \sum_n \left(\mathrm{tr}\left(\mathrm{C_n^+ C_n \rho C_n^+ C_n}\right) - \mathrm{C_n^+ C_n \rho C_n^+ C_n}\right), \tag{3.15}$$

with the probability of jumps in a time step δt given by

$$\delta p = \mathrm{tr}\left(\mathrm{C\rho C^+}\right)\delta \mathrm{t}. \tag{3.16}$$

After a jump, the density matrix become

$$\rho' = \frac{C\rho C^+}{\mathrm{tr}\left(\mathrm{C\rho C^+}\right)}.$$

The evolution of the system at each time step if thus given by

$$\rho(t+\delta t) = \rho(t) + L_{\rm eff}(\rho)\delta t + \delta N\left(\frac{C\rho C^+}{\mathrm{tr}\left(\mathrm{C\rho C^+}\right)} - \rho\right). \tag{3.17}$$

3.5.7 Stochastic Solver

When a quantum system is subjected to continuous measurement, through homodyne detection for example, it is possible to simulate the conditional quantum state using stochastic Schrodinger and master equations. The solution of these stochastic equations are quantum trajectories, which represent the conditioned evolution of the system given a specific measurement record.

In general, the stochastic evolution of a quantum state is calculated in QuTiP by solving the general equation

$$d\rho(t) = d_1\rho dt + \sum_n d_{2,n}\rho dW_n, \tag{3.18}$$

where dW_n is a Wiener increment, which has the expectation values $E[dW] = 0$ and $E[dW^2] = dt$. Stochastic evolution is implemented with the `qutip.stochastic.general_stochastic` function.

Stochastic Schrodinger Equation

The stochastic Schrodinger equation is given by (see section 4.4, [Wis09])

$$d\psi(t) = -iH\psi(t)dt - \sum_n \left(\frac{S_n^\dagger S_n}{2} - \frac{e_n}{2}S_n + \frac{e_n^2}{8}\right)\psi(t)dt + \sum_n \left(S_n - \frac{e_n}{2}\right)\psi(t)dW_n, \tag{3.19}$$

where H is the Hamiltonian, S_n are the stochastic collapse operators, and e_n is

$$e_n = \left\langle \psi(t)|S_n + S_n^\dagger|\psi(t)\right\rangle \tag{3.20}$$

In QuTiP, this equation can be solved using the function `qutip.stochastic.ssesolve`, which is implemented by defining d_1 and $d_{2,n}$ from Equation (3.18) as

$$d_1 = -iH - \frac{1}{2}\sum_n \left(S_n^\dagger S_n - e_n S_n + \frac{e_i^2}{4}\right), \tag{3.21}$$

and

$$d_{2,n} = S_n - \frac{e_n}{2}. \tag{3.22}$$

The solver `qutip.stochastic.ssesolve` will construct the operators d_1 and $d_{2,n}$ once the user passes the Hamiltonian (`H`) and the stochastic operator list (`sc_ops`). As with the `qutip.mcsolve`, the number of trajectories and the seed for the noise realisation can be fixed using the arguments: `ntraj` and `noise`, respectively. If the user also requires the measurement output, the argument `store_measurement=True` should be included.

Additionally, homodyne and heterodyne detections can be easily simulated by passing the arguments `method='homodyne'` or `method='heterodyne'` to `qutip.stochastic.ssesolve`.

Examples of how to solve the stochastic Schrodinger equation using QuTiP can be found in this development notebook.

Stochastic Master Equation

When the initial state of the system is a density matrix ρ, the stochastic master equation solver `qutip.stochastic.smesolve` must be used. The stochastic master equation is given by (see section 4.4, [Wis09])

$$d\rho(t) = -i[H, \rho(t)]dt + D[A]\rho(t)dt + \mathcal{H}[A]\rho dW(t) \tag{3.23}$$

where

$$D[A]\rho = \frac{1}{2}\left[2A\rho A^\dagger - \rho A^\dagger A - A^\dagger A\rho\right], \tag{3.24}$$

and

$$\mathcal{H}[A]\rho = A\rho(t) + \rho(t)A^\dagger - \mathrm{tr}\left[A\rho(t) + \rho(t)A^\dagger\right]. \tag{3.25}$$

In QuTiP, solutions for the stochastic master equation are obtained using the solver *qutip.stochastic.smesolve*. The implementation takes into account 2 types of collapse operators. C_i (`c_ops`) represent the dissipation in the environment, while S_n (`sc_ops`) are monitored operators. The deterministic part of the evolution, described by the d_1 in Equation (3.18), takes into account all operators C_i and S_n:

$$d_1 = -i[H(t), \rho(t)] + \sum_i D[C_i]\rho + \sum_n D[S_n]\rho, \tag{3.26}$$

The stochastic part, $d_{2,n}$, is given solely by the operators S_n

$$d_{2,n} = S_n\rho(t) + \rho(t)S_n^\dagger - \mathrm{tr}\left(S_n\rho(t) + \rho(t)S_n^\dagger\right)\rho(t). \tag{3.27}$$

As in the stochastic Schrodinger equation, the detection method can be specified using the `method` argument.

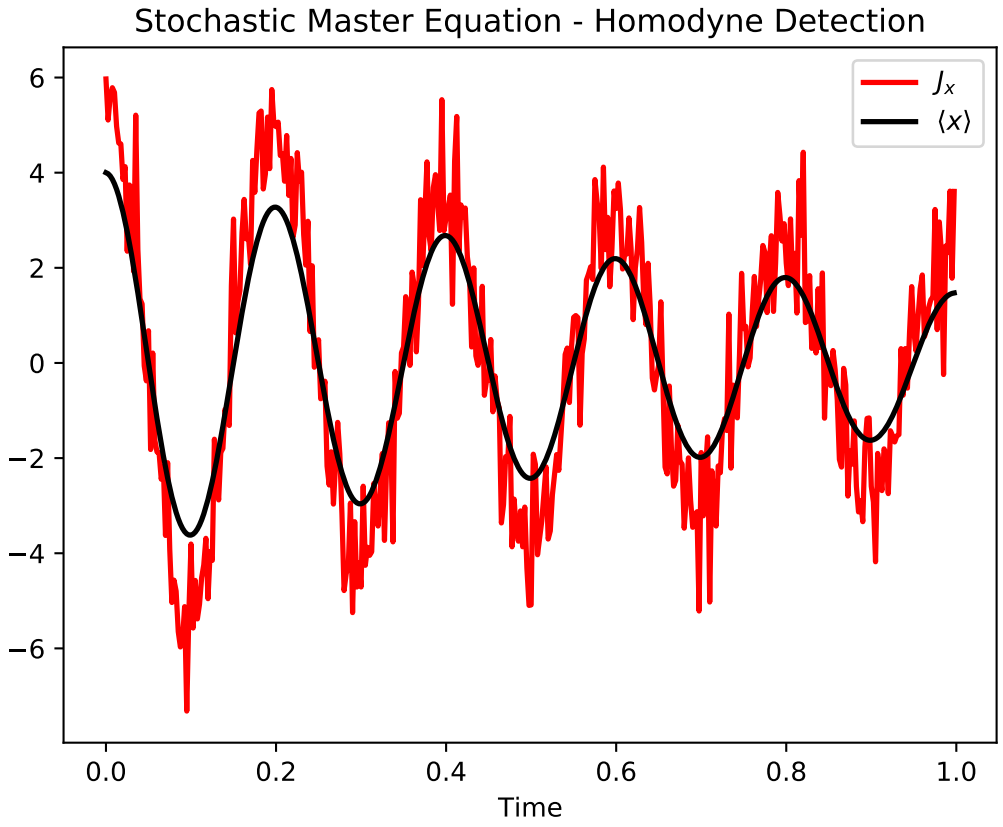
Example

Below, we solve the dynamics for an optical cavity at 0K whose output is monitored using homodyne detection. The cavity decay rate is given by κ and the Δ is the cavity detuning with respect to the driving field. The measurement operators can be passed using the option `m_ops`. The homodyne current J_x is calculated using

$$J_x = \langle x \rangle + dW, \tag{3.28}$$

where x is the operator passed using `m_ops`. The results are available in `result.measurements`.

```
import numpy as np
import matplotlib.pyplot as plt
import qutip as qt

# parameters
DIM = 20              # Hilbert space dimension
DELTA = 5*2*np.pi     # cavity detuning
KAPPA = 2             # cavity decay rate
INTENSITY = 4         # intensity of initial state
NUMBER_OF_TRAJECTORIES = 500

# operators
a = qt.destroy(DIM)
x = a + a.dag()
H = DELTA*a.dag()* a

rho_0 = qt.coherent(DIM, np.sqrt(INTENSITY))
times = np.arange(0, 1, 0.0025)

stoc_solution = qt.smesolve(H, rho_0, times,
                            c_ops=[],
                            sc_ops=[np.sqrt(KAPPA) * a],
                            e_ops=[x],
                            ntraj=NUMBER_OF_TRAJECTORIES,
                            nsubsteps=2,
                            store_measurement=True,
                            dW_factors=[1],
                            method='homodyne')

fig, ax = plt.subplots()
ax.set_title('Stochastic Master Equation - Homodyne Detection')
ax.plot(times, np.array(stoc_solution.measurement).mean(axis=0)[:].real,
        'r', lw=2, label=r'$J_x$')
ax.plot(times, stoc_solution.expect[0], 'k', lw=2,
        label=r'$\langle x \rangle$')
ax.set_xlabel('Time')
ax.legend()
```



For other examples on `qutip.stochastic.smesolve`, see the following notebook, as well as these notebooks available at QuTiP Tutorials page: heterodyne detection, inneficient detection, and feedback control.

3.5.8 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`, `qutip.bloch_redfield.brmesolve` `qutip.stochastic.ssesolve`, `qutip.stochastic.photocurrent_sesolve`, `qutip.stochastic.smesolve`, and `qutip.stochastic.photocurrent_mesolve` 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. **Array Based**: The Hamiltonian and/or collapse operators are expressed as a list of [qobj, np.array] pairs. The arrays are 1 dimensional and dtype are complex or float. They must contain one value for each time in the tlist given to the solver. Cubic spline interpolation will be used between the given times.
4. **Hamiltonian function (outdated)**: The Hamiltonian is itself a Python function with time-dependence. Collapse operators must be time independent using this input format.

Given 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.bloch_redfield.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', 'zerf', 'imag', 'log', 'log10', 'norm', 'pi',
 '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, 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)'` and `scipy.special` imported as `spe`. This includes a wide range of functionality, but comes with a small overhead created by going from C++->Python->C++.

Finally option #4, expressing the Hamiltonian as a Python function, is the original method for time dependence in QuTiP 1.x. This method is somewhat less efficient then the previously mentioned ones. However, in contrast to the other options 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` and `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 following code sets up the problem

```
ustate = basis(3, 0)
excited = basis(3, 1)
ground = basis(3, 2)

N = 2 # Set where to truncate Fock state for cavity
sigma_ge = tensor(qeye(N), ground * excited.dag())  # |g><e|
sigma_ue = tensor(qeye(N), ustate * excited.dag())  # |u><e|
a = tensor(destroy(N), qeye(3))
ada = tensor(num(N), qeye(3))

c_ops = []  # Build collapse operators
kappa = 1.5 # Cavity decay rate
c_ops.append(np.sqrt(kappa) * a)
gamma = 6  # Atomic decay rate
c_ops.append(np.sqrt(5*gamma/9) * sigma_ue) # Use Rb branching ratio of 5/9 e->u
c_ops.append(np.sqrt(4*gamma/9) * sigma_ge) # 4/9 e->g

t = np.linspace(-15, 15, 100) # Define time vector
psi0 = tensor(basis(N, 0), ustate) # Define initial state

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

g = 5  # coupling strength
H0 = -g * (sigma_ge.dag() * a + a.dag() * sigma_ge)  # time-independent term
H1 = (sigma_ue.dag() + sigma_ue)  # time-dependent term
```

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

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

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

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

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

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

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

```
kappa = 0.5

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

N = 10  # number of basis states
a = destroy(N)
H = a.dag() * a  # simple HO
psi0 = basis(N, 9)  # initial state
c_ops = [[a, col_coeff]]  # time-dependent collapse term
times = np.linspace(0, 10, 100)
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 use of the `args` variable

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

or equivalently,

```
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:

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

or to keep things looking pretty

```
args = {'A': 9, 'sigma': 5}
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:

```
ustate = basis(3, 0)
excited = basis(3, 1)
ground = basis(3, 2)

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

sigma_ge = tensor(qeye(N), ground * excited.dag())  # |g><e|
sigma_ue = tensor(qeye(N), ustate * excited.dag())  # |u><e|
```

(continues on next page)

(continued from previous page)

```
a = tensor(destroy(N), qeye(3))
ada = tensor(num(N), qeye(3))

c_ops = []  # Build collapse operators
kappa = 1.5 # Cavity decay rate
c_ops.append(np.sqrt(kappa) * a)
gamma = 6  # Atomic decay rate
c_ops.append(np.sqrt(5*gamma/9) * sigma_ue) # Use Rb branching ratio of 5/9 e->u
c_ops.append(np.sqrt(4*gamma/9) * sigma_ge) # 4/9 e->g

t = np.linspace(-15, 15, 100) # Define time vector
psi0 = tensor(basis(N, 0), ustate) # Define initial state
state_GG = tensor(basis(N, 1), ground) # Define states onto which to project
sigma_GG = state_GG * state_GG.dag()
state_UU = tensor(basis(N, 0), ustate)
sigma_UU = state_UU * state_UU.dag()

g = 5  # coupling strength
H0 = -g * (sigma_ge.dag() * a + a.dag() * sigma_ge)  # time-independent term
H1 = (sigma_ue.dag() + sigma_ue)  # time-dependent term
```

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

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

```
output = mesolve(H, psi0, 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)'`

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

Important: Naming your `args` variables `exp`, `sin`, `pi` etc. 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

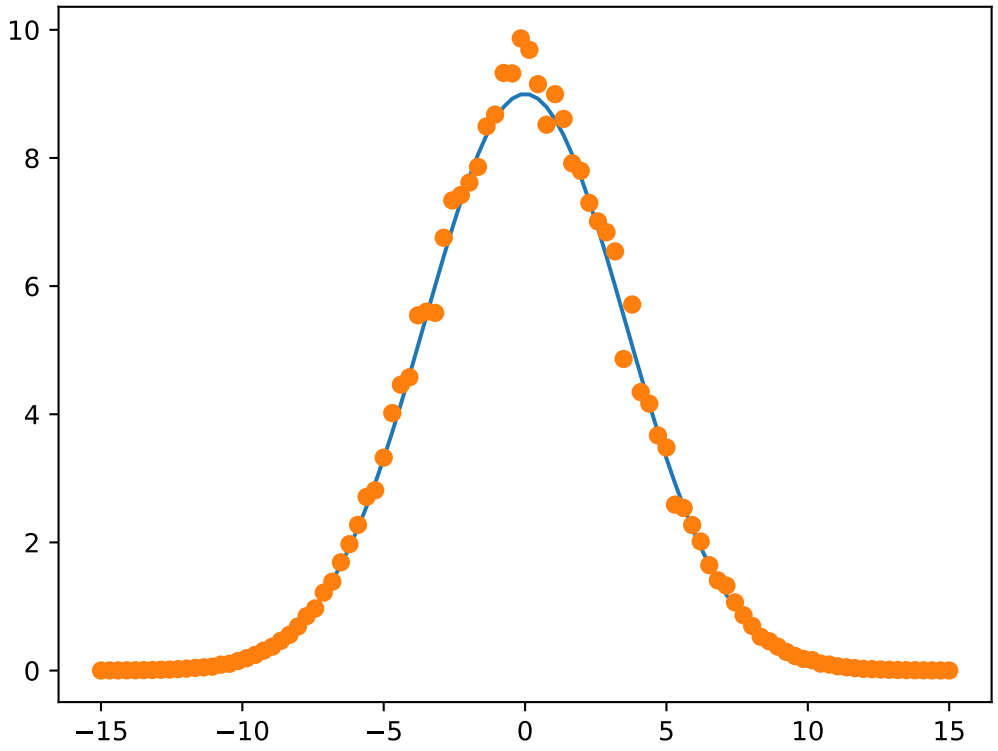
Sometimes it is necessary to model a system where the time-dependent parameters are non-analytic functions, or are derived from experimental data (i.e. a collection of data points). In these situations, one can use interpolating functions as an approximate functional form for input into a time-dependent solver. QuTiP includes its 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:

```
t = np.linspace(-15, 15, 100)
func = lambda t: 9*np.exp(-(t / 5)** 2)
noisy_func = lambda t: func(t)+(0.05*func(t))*np.random.randn(t.shape[0])
noisy_data = noisy_func(t)

plt.figure()
plt.plot(t, func(t))
```

(continues on next page)

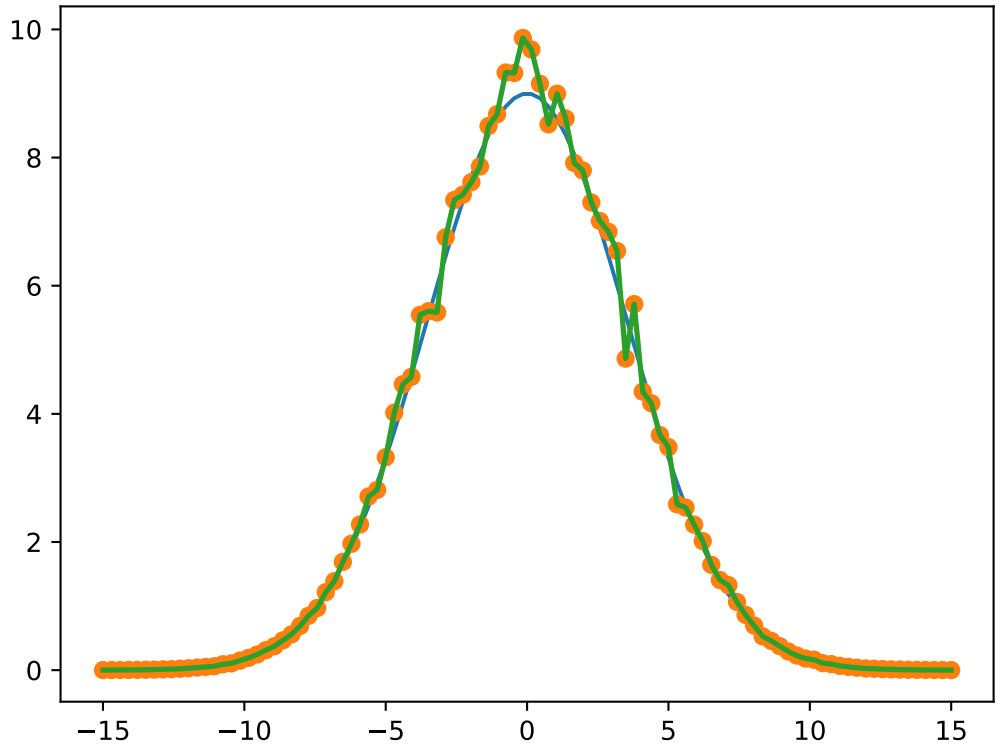
(continued from previous page)

```
plt.plot(t, noisy_data, 'o')
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:

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

plt.figure()
plt.plot(t, func(t))
plt.plot(t, noisy_data, 'o')
plt.plot(t, S(t), lw=2)
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.

Accesing the state from solver

New in QuTiP 4.4

The state of the system, the ket vector or the density matrix, is available to time-dependent Hamiltonian and collapse operators in `args`. Some keys of the argument dictionary are understood by the solver to be values to be updated with the evolution of the system. The state can be obtained in 3 forms: `Qobj`, vector (1d `np.array`), matrix (2d `np.array`), expectation values and collapse can also be obtained.

	Preparation	usage	Notes
state as Qobj	`name+"=Qobj(psi0)"`	`psi0=args[name]`	The ket or density matrix as a Qobj with `psi0`'s dimensions
state as matrix	`name+"=mat(psi0)"`	`psi0=args[name]`	The state as a matrix, equivalent to `state.full()`
state as vector	`name+"=vec(psi0)"`	`psi0=args[name]`	The state as a vector, equivalent to `state.full().ravel('F')`
expectation value	`name+"=expect(O,psi0)"`	`O=args[name]`	Expectation value of the operator `O`, either $\langle\psi(t)\|O\|\psi(t)\rangle$ or $\mathrm{tr}\left(O\rho(t)\right)$
collpases	`name+"=collapse(psi0)"`	`psi0=args[name]`	List of collapse, each collapse is a tuple of the pair `(time, which)` `which` being the indice of the collapse operator. `mcsolve` only.

Here `psi0` is the initial value used for tests before the evolution begins. *qutip.bloch_redfield.brmesolve* does not support these arguments.

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.solver.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

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

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.parallel.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.solver.Options* class. Because the *qutip.parallel.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.solver.Options* 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:

```
delta = 0.1  * 2 * np.pi  # qubit sigma_x coefficient
w = 2.0  * 2 * np.pi      # driving frequency
T = 2 * np.pi / w         # driving period
gamma1 = 0.00001          # relaxation rate
gamma2 = 0.005            # dephasing  rate

eps_list = np.linspace(-10.0, 10.0, 51) * 2 * np.pi  # epsilon
A_list = np.linspace(0.0, 20.0, 51) * 2 * np.pi      # Amplitude

sx = sigmax(); sz = sigmaz(); sm = destroy(2); sn = num(2)

c_ops = [np.sqrt(gamma1) * sm, np.sqrt(gamma2) * sz]  # relaxation and dephasing
H0 = -delta / 2.0 * sx
H1 = [sz, '-eps / 2.0 + A / 2.0 * sin(w * t)']
H_td = [H0, H1]
Hargs = {'w': w, 'eps': eps_list[0], 'A': A_list[0]}
```

where the last code block sets up the problem using a string-based Hamiltonian, and `Hargs` is a dictionary of arguments to be passed into the Hamiltonian. In this example, we are going to use the *qutip.propagator* and *qutip.propagator.propagator_steadystate* to find expectation values for different values of ϵ and A in the Hamiltonian $H = -\frac{1}{2}\Delta\sigma_x - \frac{1}{2}\epsilon\sigma_z - \frac{1}{2}A\sin(\omega t)$.

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

```
opts = Options(rhs_reuse=True)
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]`:

```
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 tells the *qutip.mesolve* function used in the propagator to call the pre-generated file `lz_func`. If this was missing then the routine would fail.

3.5.9 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): $H = H_{\rm S} + H_{\rm B} + H_{\rm I}$, where H is the total system+bath Hamiltonian, $H_{\rm S}$ and $H_{\rm B}$ are the system and bath Hamiltonians, respectively, and $H_{\rm I}$ is the interaction Hamiltonian.

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

$$\frac{d}{dt}\rho_S(t) = -\hbar^{-2}\int_0^t d\tau\ {\rm Tr}_B[H_I(t), [H_I(\tau), \rho_S(\tau)\otimes\rho_B]], \tag{3.29}$$

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 (3.29) 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 $\rho_S(\tau)$ is replaced by $\rho_S(t)$ in Eq. (3.29). The result is the Redfield equation

$$\frac{d}{dt}\rho_S(t) = -\hbar^{-2}\int_0^t d\tau\ {\rm Tr}_B[H_I(t), [H_I(\tau), \rho_S(t)\otimes\rho_B]], \tag{3.30}$$

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

$$\frac{d}{dt}\rho_S(t) = -\hbar^{-2}\int_0^\infty d\tau\ {\rm Tr}_B[H_I(t), [H_I(t-\tau), \rho_S(t)\otimes\rho_B]]. \tag{3.31}$$

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 (3.31) 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

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

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 $S(\omega) = \int_{-\infty}^{\infty} d\tau e^{i\omega\tau}g(\tau)$:

$$\int_0^\infty d\tau\ g_{\alpha\beta}(\tau)e^{i\omega\tau} = \frac{1}{2}S_{\alpha\beta}(\omega) + i\lambda_{\alpha\beta}(\omega), \tag{3.32}$$

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

$$\frac{d}{dt}\rho_{ab}(t) = -i\omega_{ab}\rho_{ab}(t) + \sum_{c,d}^{\mathrm{sec}}R_{abcd}\rho_{cd}(t), \tag{3.33}$$

where

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

is the Bloch-Redfield tensor.

The Bloch-Redfield master equation in the form Eq. (3.33) is suitable for numerical implementation. The input parameters are the system Hamiltonian H, the system operators through which the environment couples to the system A_α, and the noise-power spectrum $S_{\alpha\beta}(\omega)$ associated with each system-environment interaction term.

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

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

Bloch-Redfield master equation in QuTiP

In QuTiP, the Bloch-Redfield tensor Eq. (3.34) 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

$$H = -\frac{1}{2}\Delta\sigma_x - \frac{1}{2}\epsilon_0\sigma_z \tag{3.34}$$

```
delta = 0.2 * 2*np.pi
eps0 = 1.0 * 2*np.pi
gamma1 = 0.5

H = - delta/2.0 * sigmax() - eps0/2.0 * sigmaz()

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


R, H_ekets = bloch_redfield_tensor(H, [[sigmax(), ohmic_spectrum]])

print(R)
```

Output:

```
Quantum object: dims = [[[2], [2]], [[2], [2]]], shape = (4, 4), type = super,␣
↪isherm = False
Qobj data =
[[ 0.         +0.j          0.         +0.j          0.         +0.j
   0.24514517+0.j        ]
 [ 0.         +0.j         -0.16103412-6.4076169j  0.         +0.j
   0.         +0.j        ]
 [ 0.         +0.j          0.         +0.j        -0.16103412+6.4076169j
   0.         +0.j        ]
 [ 0.         +0.j          0.         +0.j          0.         +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 the eigenstates of *H* (*H_ekets*), since they are calculated in the process of calculating the Bloch-Redfield tensor *R*, and the *H_ekets* are usually needed again later when transforming operators between the computational basis and the eigenbasis.

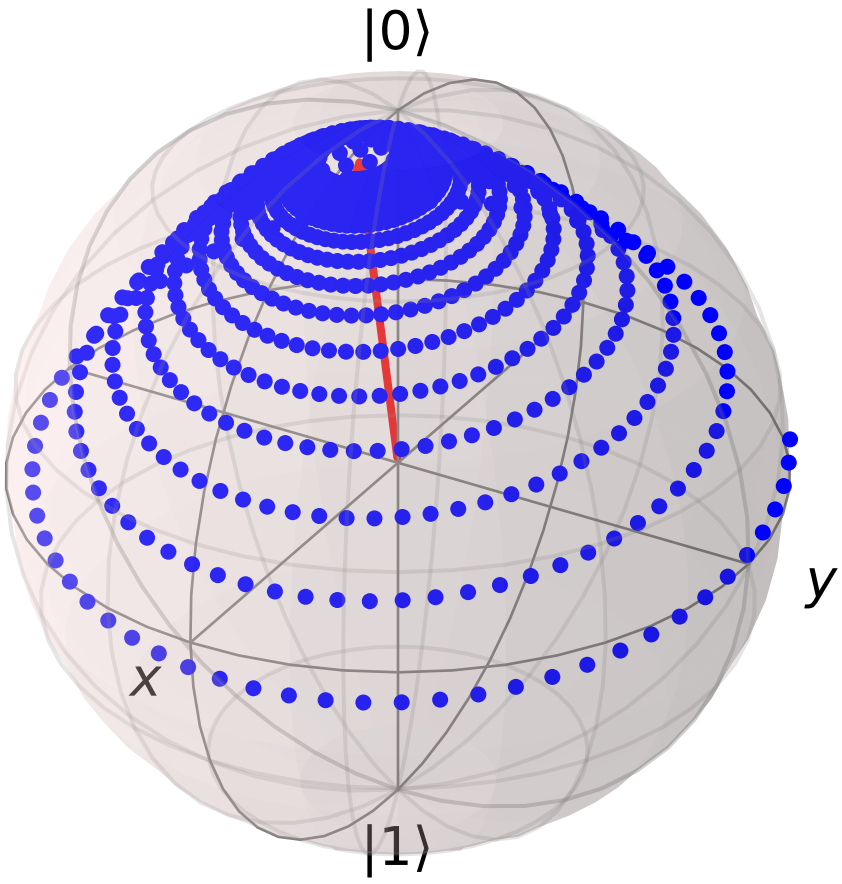
The evolution of a wavefunction or density matrix, according to the Bloch-Redfield master equation (3.33), can be calculated using the QuTiP function *qutip.bloch_redfield.bloch_redfield_solve*. It takes five mandatory arguments: the Bloch-Redfield tensor `R`, the list of eigenkets `H_ekets` of the hamiltonian, the initial state `psi0` (as a ket or density matrix), a list of times `tlist` for which to evaluate the expectation values, and a list of operators `e_ops` for which to evaluate the expectation values at each time step defined by *tlist*. For example, to evaluate the expectation values of the σ_x, σ_y, and σ_z operators for the example above, we can use the following code:

```
tlist = np.linspace(0, 15.0, 1000)
```

(continues on next page)

(continued from previous page)

```
psi0 = rand_ket(2)

e_ops = [sigmax(), sigmay(), sigmaz()]

expt_list = bloch_redfield_solve(R, H_ekets, psi0, tlist, e_ops)

sphere = Bloch()

sphere.add_points([expt_list[0], expt_list[1], expt_list[2]])

sphere.vector_color = ['r']

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

sphere.make_sphere()
```



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`.

```
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

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:

```
ohmic = "{gamma1} / 2.0 * (w / (2 * pi)) * (w > 0.0)".format(gamma1=gamma1)

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.

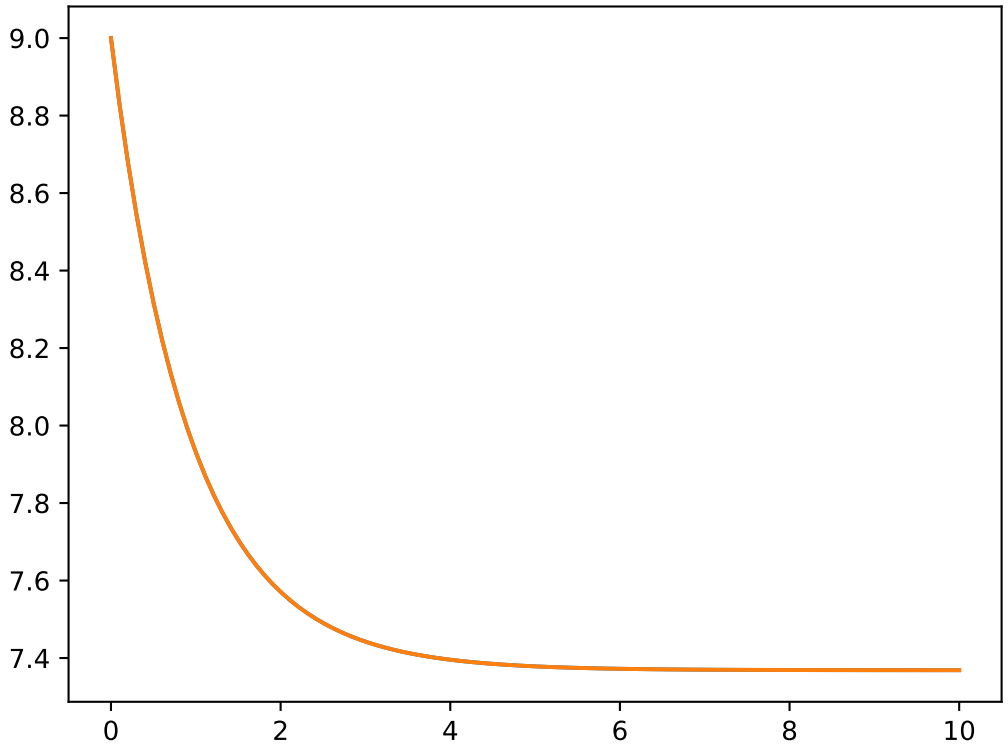
For actual time-dependent Hamiltonians, the Hamiltonian itself can be passed into the solver like any other string-based Hamiltonian, as thus we will not discuss this topic further. Instead, here the focus is on time-dependent bath coupling terms. To this end, suppose that we have a dissipative harmonic 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:

```
N = 10  # number of basis states to consider

a = destroy(N)

H = a.dag() * a

psi0 = basis(N, 9)  # initial state

kappa = 0.2  # coupling to oscillator

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

tlist = np.linspace(0, 10, 100)

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

actual_answer = 9.0 * np.exp(-kappa * (1.0 - np.exp(-tlist)))

plt.figure()
```

(continues on next page)

(continued from previous page)

```
plt.plot(tlist, out.expect[0])

plt.plot(tlist, actual_answer)

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:

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

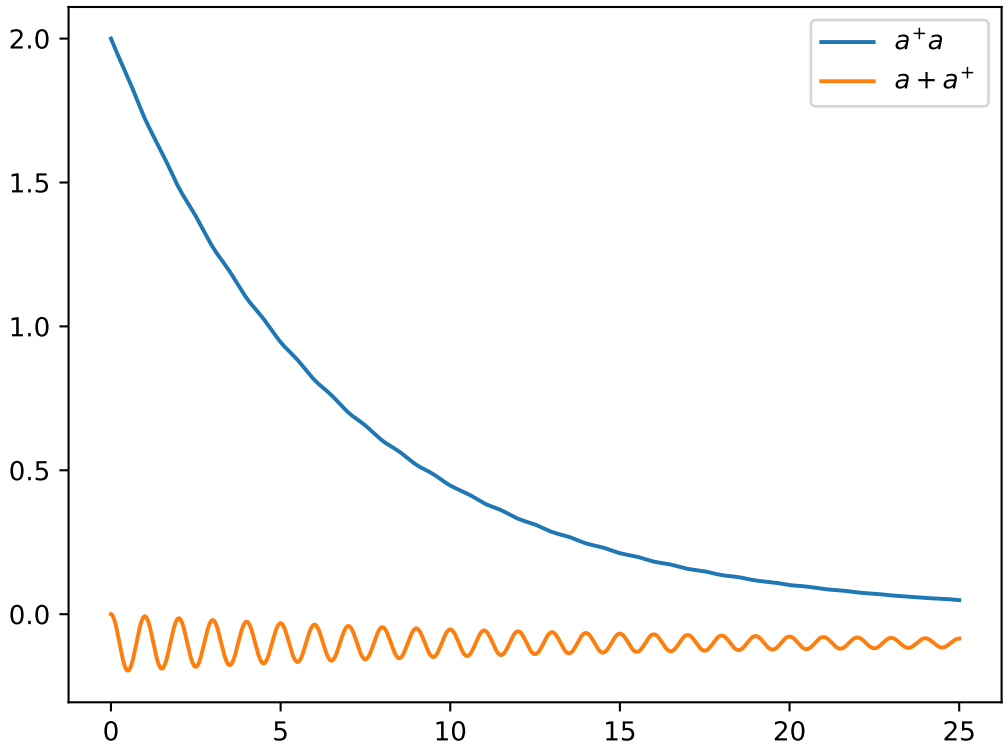
where the first tuple element `(a, a.dag())` tells the solver which operators make up the full Hermitian coupling operator. The second tuple `('{0} * (w >= 0)'.format(kappa), 'exp(1j*t)', 'exp(-1j*t)')`, gives the noise power spectrum, and time-dependence of each operator. Note that the noise spectrum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

```
N = 10

w0 = 1.0 * 2 * np.pi

g = 0.05 * w0

kappa = 0.15
```

(continues on next page)

(continued from previous page)

```
times = np.linspace(0, 25, 1000)

a = destroy(N)

H = w0 * a.dag() * a + g * (a + a.dag())

psi0 = ket2dm((basis(N, 4) + basis(N, 2) + basis(N, 0)).unit())

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

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

res_brme = brmesolve(H, psi0, times, a_ops, e_ops)

plt.figure()

plt.plot(times,res_brme.expect[0], label=r'$a^{+}a$')

plt.plot(times,res_brme.expect[1], label=r'$a+a^{+}$')

plt.legend()

plt.show()
```



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

3.5.10 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

$$H(t)\Psi(t) = i\hbar\frac{\partial}{\partial t}\Psi(t), \tag{3.35}$$

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 (3.35) of the form

$$\Psi_\alpha(t) = \exp(-i\epsilon_\alpha t/\hbar)\Phi_\alpha(t), \tag{3.36}$$

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

$$\Psi(t) = \sum_\alpha c_\alpha \Psi_\alpha(t) = \sum_\alpha c_\alpha \exp(-i\epsilon_\alpha t/\hbar)\Phi_\alpha(t), \tag{3.37}$$

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 (3.35). By substituting (3.36) into the Schrödinger equation (3.35) we obtain an eigenvalue equation for the Floquet modes and quasienergies

$$\mathcal{H}(t)\Phi_\alpha(t) = \epsilon_\alpha \Phi_\alpha(t), \tag{3.38}$$

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 (3.35), which by definition satisfies

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

Inserting the Floquet states from (3.36) 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

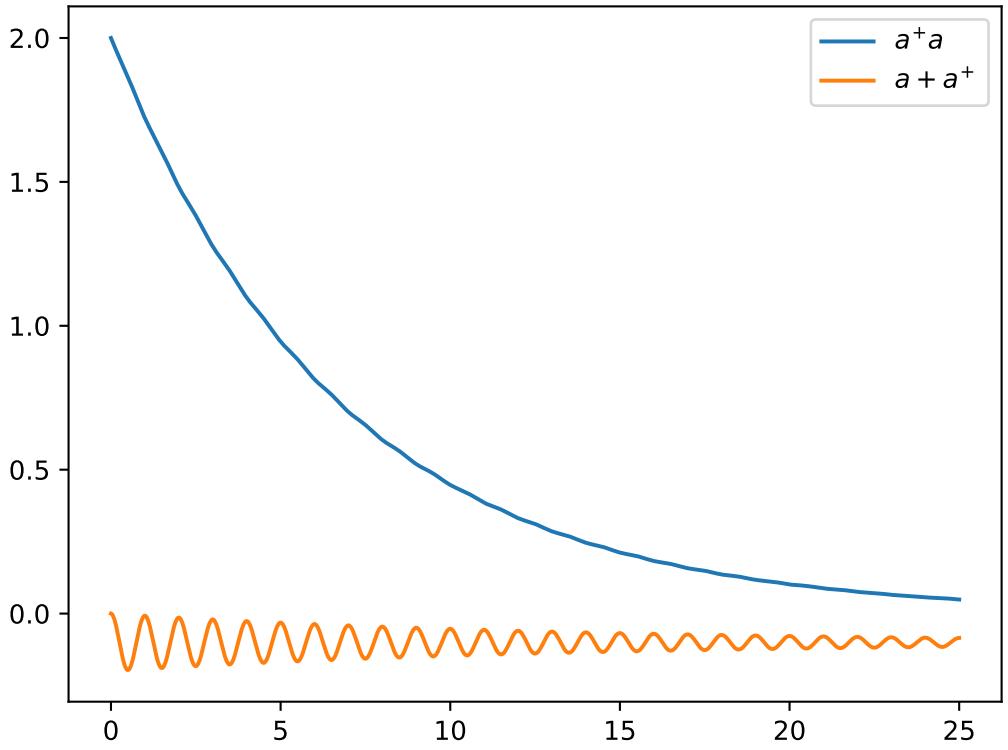
QuTiP provides a family of functions to calculate the Floquet modes and quasi energies, Floquet state decomposition, etc., given a time-dependent Hamiltonian on the *callback format*, *list-string format* and *list-callback format* (see, e.g., `qutip.mesolve` for details).

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

$$H(t) = -\frac{1}{2}\Delta\sigma_x - \frac{1}{2}\epsilon_0\sigma_z + \frac{1}{2}A\sin(\omega t)\sigma_z. \tag{3.39}$$

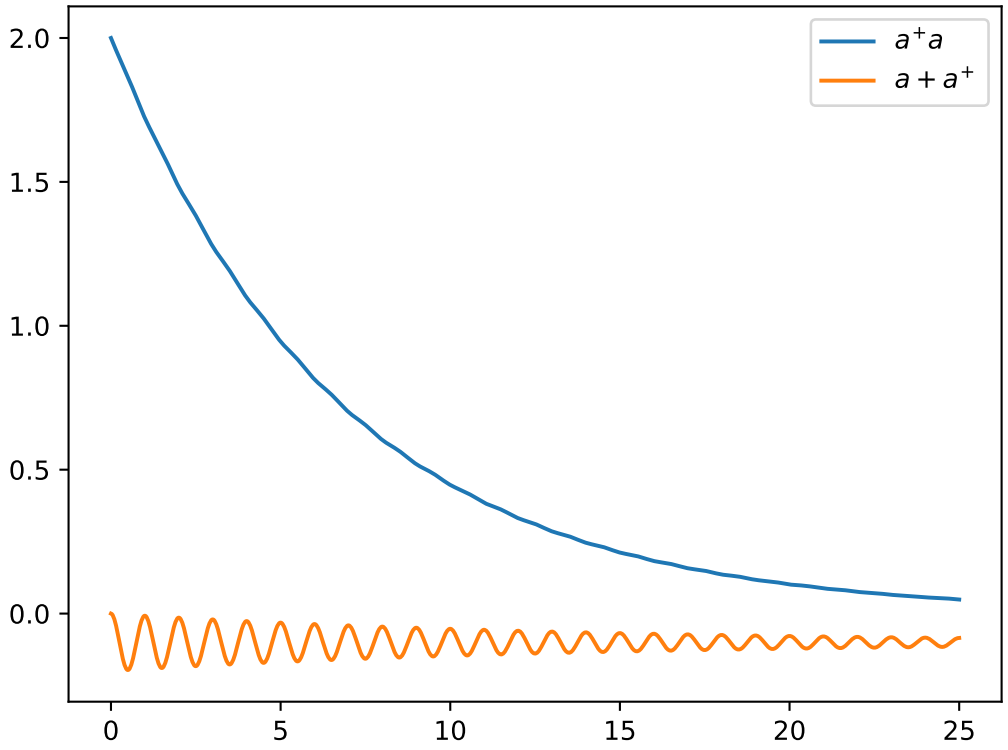
In QuTiP we can define this Hamiltonian as follows:

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



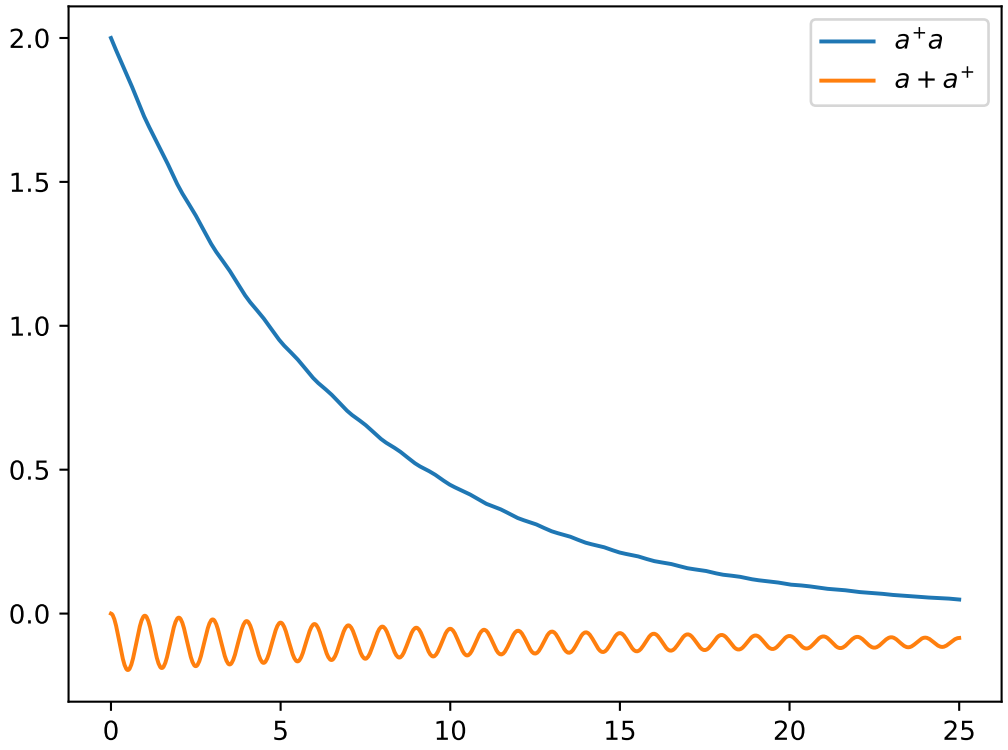
The $t = 0$ Floquet modes corresponding to the Hamiltonian (3.39) can then be calculated using the *qutip.floquet.floquet_modes* function, which returns lists containing the Floquet modes and the quasienergies

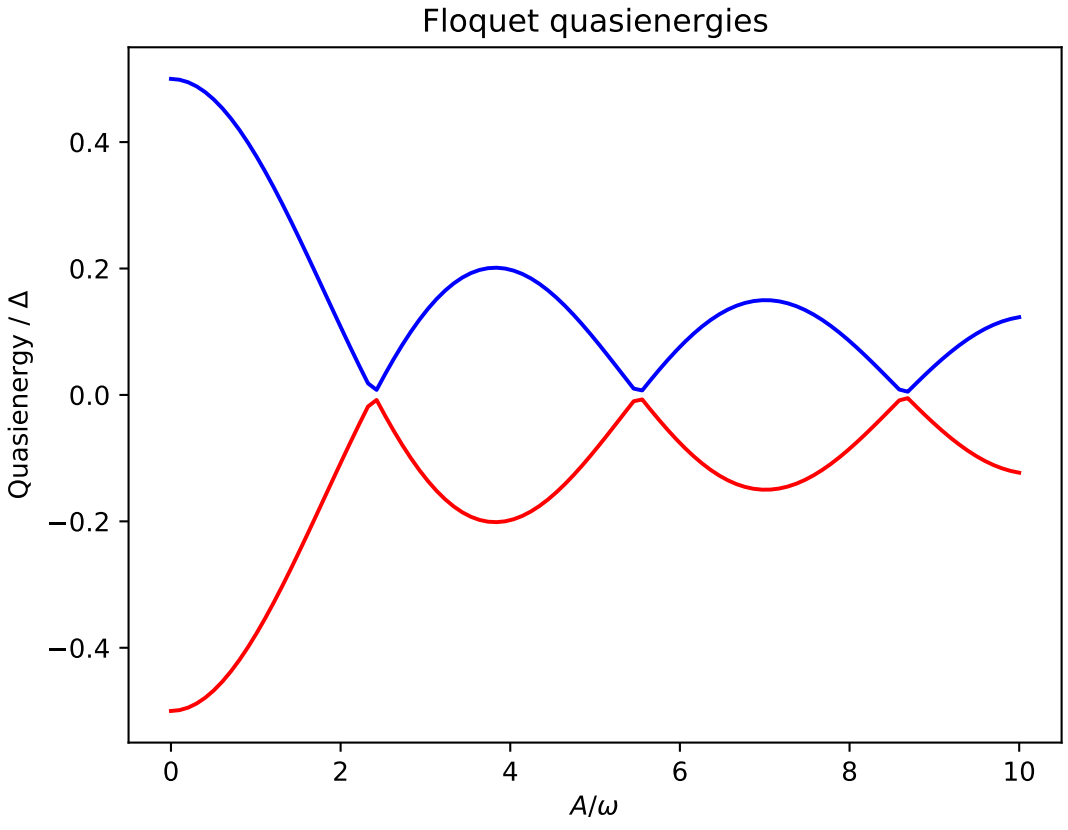
```
>>> T = 2*np.pi / omega
>>> f_modes_0, f_energies = floquet_modes(H, T, args)
>>> f_energies
array([-2.83131212,  2.83131212])
>>> f_modes_0
[Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[ 0.72964231+0.j      ]
 [-0.39993746+0.554682j]],
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[0.39993746+0.554682j]
 [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 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).

```
>>> delta = 0.2 * 2*np.pi
>>> eps0  = 0.0 * 2*np.pi
>>> omega = 1.0 * 2*np.pi
>>> A_vec = np.linspace(0, 10, 100) * omega
>>> T = (2*np.pi)/omega
>>> tlist = np.linspace(0.0, 10 * T, 101)
>>> spsi0 = basis(2,0)
>>> q_energies = np.zeros((len(A_vec), 2))
>>> H0 = delta/2.0 * sigmaz() - eps0/2.0 * sigmax()
>>> args = {'w': omega}
>>> for idx, A in enumerate(A_vec):
>>>   H1 = A/2.0 * sigmax()
>>>   H = [H0, [H1, lambda t, args: np.sin(args['w']*t)]]
>>>   f_modes, f_energies = floquet_modes(H, T, args, True)
>>>   q_energies[idx,:] = f_energies
>>> plt.figure()
>>> plt.plot(A_vec/omega, q_energies[:,0] / delta, 'b', A_vec/omega, q_energies[:,
↪1] / delta, 'r')
>>> plt.xlabel(r'$A/\omega$')
>>> plt.ylabel(r'Quasienergy / $\Delta$')
>>> plt.title(r'Floquet quasienergies')
>>> plt.show()
```


a^+a
$a + a^+$
2.0
1.5
1.0
0.5
0.0
0
5
10
15
20
25


Floquet quasienergies
Quasienergy / Δ
0.4
0.2
0.0
−0.2
−0.4
0
2
4
6
8
10
A/ω

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

```
>>> f_modes_t = floquet_modes_t(f_modes_0, f_energies, 2.5, H, T, args)
>>> f_modes_t
[Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[-0.89630512-0.23191946j]
 [ 0.37793106-0.00431336j]],
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[-0.37793106-0.00431336j]
 [-0.89630512+0.23191946j]]]
```

The purpose of calculating the Floquet modes is to find the wavefunction solution to the original problem (3.39) given some initial state $|\psi_0\rangle$. To do that, we first need to decompose the initial state in the Floquet states, using the function *qutip.floquet.floquet_state_decomposition*

```
>>> psi0 = rand_ket(2)
>>> f_coeff = floquet_state_decomposition(f_modes_0, f_energies, psi0)
>>> f_coeff
[(-0.645265993068382+0.7304552549315746j),
(0.15517002114250228-0.16121161022382 58j)]
```

and given this decomposition of the initial state in the Floquet states we can easily evaluate the wavefunction that is the solution to (3.39) at an arbitrary time t using the function *qutip.floquet.floquet_wavefunction_t*

```
>>> t = 10 * np.random.rand()
>>> psi_t = floquet_wavefunction_t(f_modes_0, f_energies, f_coeff, t, H, T, args)
```

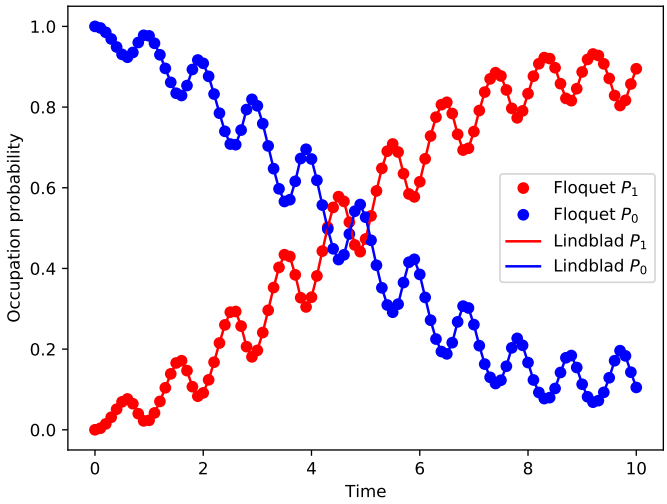
The following example illustrates how to use the functions introduced above to calculate and plot the time-evolution of (3.39).

```
import numpy as np
from matplotlib import pyplot

import qutip

delta = 0.2 * 2*np.pi
eps0  = 1.0 * 2*np.pi
A     = 0.5 * 2*np.pi
omega = 1.0 * 2*np.pi
T     = (2*np.pi)/omega
tlist  = np.linspace(0.0, 10 * T, 101)
psi0   = qutip.basis(2, 0)

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

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

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

# calculate the wavefunctions using the from the floquet modes
p_ex = np.zeros(len(tlist))
for n, t in enumerate(tlist):
    psi_t = qutip.floquet_wavefunction_t(f_modes_0, f_energies, f_coeff, t, H, T,␣
↪args)
```

(continues on next page)

(continued from previous page)

```
    p_ex[n] = qutip.expect(qutip.num(2), psi_t)

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

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



Pre-computing the Floquet modes for one period

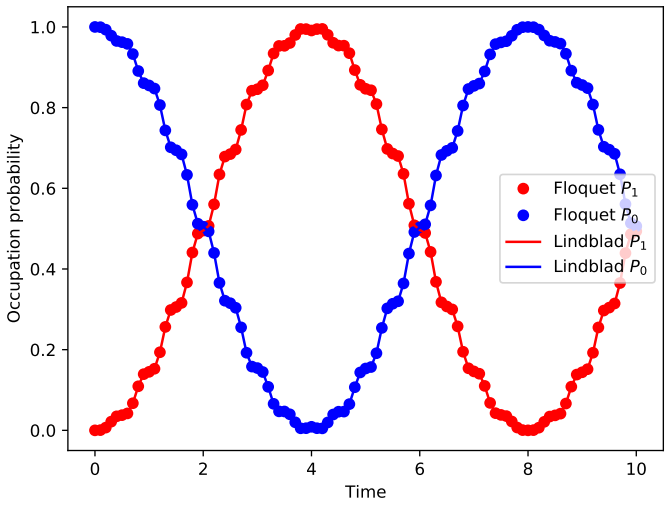
When evaluating the Floquet states or the wavefunction at many points in time it is useful to pre-compute the Floquet modes for the first period of the driving with the required resolution. In QuTiP the function *qutip.floquet.floquet_modes_table* calculates a table of Floquet modes which later can be 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.

```
import numpy as np
from matplotlib import pyplot
import qutip

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

H0 = - delta/2.0 * qutip.sigmax() - eps0/2.0 * qutip.sigmaz()
H1 = A/2.0 * qutip.sigmax()
args = {'w': omega}
```

(continues on next page)

(continued from previous page)

```
H = [H0, [H1, lambda t, args: np.sin(args['w'] * t)]]

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

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

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

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

# plot the results
pyplot.plot(tlist, np.real(p_ex),     'ro', tlist, 1-np.real(p_ex),     'bo')
pyplot.plot(tlist, np.real(p_ex_ref), 'r',  tlist, 1-np.real(p_ex_ref), 'b')
pyplot.xlabel('Time')
pyplot.ylabel('Occupation probability')
pyplot.legend(("Floquet $P_1$", "Floquet $P_0$", "Lindblad $P_1$", "Lindblad $P_0$
↪"))
pyplot.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=psi0, tlist=tlist, e_ops=[qutip.num(2)], args=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.

For a brief summary of the derivation, the important contents for the implementation in QuTiP are listed below.

The floquet mode $|\phi_\alpha(t)\rangle$ refers to a full class of quasienergies defined by $\epsilon_\alpha + k\Omega$ for arbitrary k. Hence, the quasienenergy difference between two floquet modes is given by

$$\Delta_{\alpha\beta k} = \frac{\epsilon_\alpha - \epsilon_\beta}{\hbar} + k\Omega$$

For any coupling operator q (given by the user) the matrix elements in the floquet basis are calculated as:

$$X_{\alpha\beta k} = \frac{1}{T}\int_0^T dt\; e^{-ik\Omega t}\,\langle\phi_\alpha(t)|\,q\,|\phi_\beta(t)\rangle$$

From the matrix elements and the spectral density $J(\omega)$, the decay rate $\gamma_{\alpha\beta k}$ is defined:

$$\gamma_{\alpha\beta k} = 2\pi\Theta(\Delta_{\alpha\beta k})J(\Delta_{\alpha\beta k})|X_{\alpha\beta k}|^2$$

where Θ is the Heaviside function. The master equation is further simplified by the RWA, which makes the following matrix useful:

$$A_{\alpha\beta} = \sum_{k=-\infty}^{\infty}[\gamma_{\alpha\beta k} + n_{th}(|\Delta_{\alpha\beta k}|)(\gamma_{\alpha\beta k} + \gamma_{\alpha\beta -k})$$

The density matrix of the system then evolves according to:

$$\dot\rho_{\alpha\alpha}(t) = \sum_\nu (A_{\alpha\nu}\rho_{\nu\nu}(t) - A_{\nu\alpha}\rho_{\alpha\alpha}(t))$$

$$\dot\rho_{\alpha\beta}(t) = -\frac{1}{2}\sum_\nu (A_{\nu\alpha} + A_{\nu\beta})\rho_{\alpha\beta}(t) \qquad \alpha \neq \beta$$

The Floquet-Markov master equation in QuTiP

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

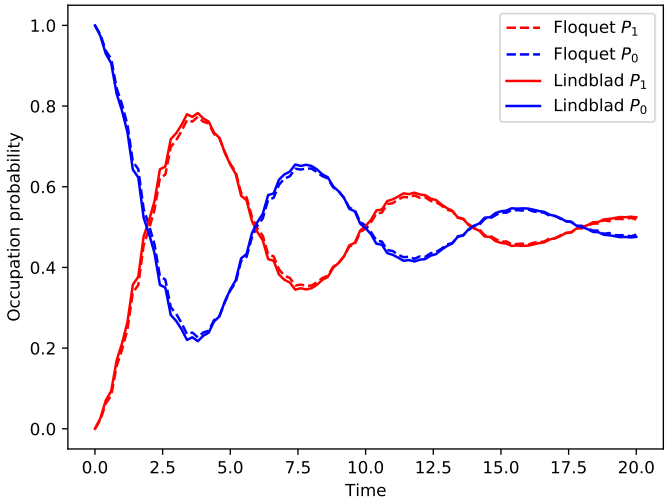
Note: Currently the *qutip.floquet.fmmesolve* can only accept a single environment coupling operator and spectral-density function.

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

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

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

```
import numpy as np
from matplotlib import pyplot
import qutip

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

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

# noise power spectrum
gamma1 = 0.1
def noise_spectrum(omega):
    return 0.5 * gamma1 * omega/(2*np.pi)

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

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

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

# calculate expectation values in the computational basis
p_ex = np.zeros(tlist.shape, dtype=np.complex128)
for idx, t in enumerate(tlist):
    f_modes_t = qutip.floquet_modes_t_lookup(f_modes_table_t, t, T)
    f_states_t = qutip.floquet_states(f_modes_t, f_energies, t)
    p_ex[idx] = qutip.expect(qutip.num(2), output.states[idx].transform(f_states_t,
↪ True))

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

# plot the results
pyplot.plot(tlist, np.real(p_ex), 'r--', tlist, 1-np.real(p_ex), 'b--')
pyplot.plot(tlist, np.real(p_ex_ref), 'r', tlist, 1-np.real(p_ex_ref), 'b')
pyplot.xlabel('Time')
pyplot.ylabel('Occupation probability')
pyplot.legend(("Floquet $P_1$", "Floquet $P_0$", "Lindblad $P_1$", "Lindblad $P_0$
↪"))
pyplot.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, by using:

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

3.5.11 Permutational Invariance

Permutational Invariant Quantum Solver (PIQS)

The *Permutational Invariant Quantum Solver (PIQS)* is a QuTiP module that allows to study the dynamics of an open quantum system consisting of an ensemble of identical qubits that can dissipate through local and collective baths according to a Lindblad master equation.

The Liouvillian of an ensemble of N qubits, or two-level systems (TLSs), $\mathcal{D}_{TLS}(\rho)$, can be built using only polynomial – instead of exponential – resources. This has many applications for the study of realistic quantum optics models of many TLSs and in general as a tool in cavity QED.

Consider a system evolving according to the equation

$$\dot{\rho} = \mathcal{D}_{\text{TLS}}(\rho) = -\frac{i}{\hbar}[H,\rho] + \frac{\gamma_{\text{CE}}}{2}\mathcal{L}_{J_-}[\rho] + \frac{\gamma_{\text{CD}}}{2}\mathcal{L}_{J_z}[\rho] + \frac{\gamma_{\text{CP}}}{2}\mathcal{L}_{J_+}[\rho] + \sum_{n=1}^{N}\left(\frac{\gamma_{\text{E}}}{2}\mathcal{L}_{J_{-,n}}[\rho] + \frac{\gamma_{\text{D}}}{2}\mathcal{L}_{J_{z,n}}[\rho] + \frac{\gamma_{\text{P}}}{2}\mathcal{L}_{J_{+,n}}[\rho]\right)$$

where $J_{\alpha,n} = \frac{1}{2}\sigma_{\alpha,n}$ are SU(2) Pauli spin operators, with $\alpha = x, y, z$ and $J_{\pm,n} = \sigma_{\pm,n}$. The collective spin operators are $J_\alpha = \sum_n J_{\alpha,n}$. The Lindblad super-operators are $\mathcal{L}_A = 2A\rho A^\dagger - A^\dagger A\rho - \rho A^\dagger A$.

The inclusion of local processes in the dynamics lead to using a Liouvillian space of dimension 4^N. By exploiting the permutational invariance of identical particles [2-8], the Liouvillian $\mathcal{D}_{\text{TLS}}(\rho)$ can be built as a block-diagonal matrix in the basis of Dicke states $|j,m\rangle$.

The system under study is defined by creating an object of the `Dicke` class, e.g. simply named `system`, whose first attribute is

- `system.N`, the number of TLSs of the system N.

The rates for collective and local processes are simply defined as

- `collective_emission` defines γ_{CE}, collective (superradiant) emission
- `collective_dephasing` defines γ_{CD}, collective dephasing

- `collective_pumping` defines γ_{CP}, collective pumping.
- `emission` defines γ_{E}, incoherent emission (losses)
- `dephasing` defines γ_{D}, local dephasing
- `pumping` defines γ_{P}, incoherent pumping.

Then the `system.lindbladian()` creates the total TLS Lindbladian superoperator matrix. Similarly, `system.hamiltonian` defines the TLS hamiltonian of the system H_{TLS}.

The system's Liouvillian can be built using `system.liouvillian()`. The properties of a Piqs object can be visualized by simply calling `system`. We give two basic examples on the use of *PIQS*. In the first example the incoherent emission of N driven TLSs is considered.

```
from piqs import Dicke
from qutip import steadystate
N = 10
system = Dicke(N, emission = 1, pumping = 2)
L = system.liouvillian()
steady = steadystate(L)
```

For more example of use, see the "Permutational Invariant Lindblad Dynamics" section in the tutorials section of the website, https://qutip.org/tutorials.html.

Table 2: Useful PIQS functions.

Operators	Command	Description
Collective spin algebra $J_x,\ J_y,\ J_z$	`jspin(N)`	The collective spin algebra $J_x,\ J_y,\ J_z$ for N TLSs
Collective spin J_x	`jspin(N, "x")`	The collective spin operator Jx. Requires N number of TLSs
Collective spin J_y	`jspin(N, "y")`	The collective spin operator J_y. Requires N number of TLSs
Collective spin J_z	`jspin(N, "z")`	The collective spin operator J_z. Requires N number of TLSs
Collective spin J_+	`jspin(N, "+")`	The collective spin operator J_+.
Collective spin J_-	`jspin(N, "-")`	The collective spin operator J_-.
Collective spin J_z in uncoupled basis	`jspin(N, "z", basis='uncoupled')`	The collective spin operator J_z in the uncoupled basis of dimension 2^N.
Dicke state $\|j,m\rangle$ density matrix	`dicke(N, j, m)`	The density matrix for the Dicke state given by $\|j,m\rangle$
Excited-state density matrix in Dicke basis	`excited(N)`	The excited state in the Dicke basis
Excited-state density matrix in uncoupled basis	`excited(N, basis="uncoupled")`	The excited state in the uncoupled basis
Ground-state density matrix in Dicke basis	`ground(N)`	The ground state in the Dicke basis
GHZ-state density matrix in the Dicke basis	`ghz(N)`	The GHZ-state density matrix in the Dicke (default) basis for N number of TLS
Collapse operators of the ensemble	`Dicke.c_ops()`	The collapse operators for the ensemble can be called by the *c_ops* method of the Dicke class.

Note that the mathematical object representing the density matrix of the full system that is manipulated (or obtained from *steadystate*) in the Dicke-basis formalism used here is a *representative of the density matrix*. This *representative object* is of linear size N^2, whereas the full density matrix is defined over a 2^N Hilbert space. In order to calculate nonlinear functions of such density matrix, such as the Von Neumann entropy or the purity, it is necessary to take into account the degeneracy of each block of such block-diagonal density matrix. Note that as long as one calculates expected values of operators, being Tr[A*rho] a *linear* function of *rho*, the *representative density matrix* give straightforwardly the correct result. When a *nonlinear* function of the density matrix needs to be calculated, one needs to weigh each degenerate block correctly; this is taken care by the *dicke_function_trace*

in *qutip.piqs*, and the user can use it to define general nonlinear functions that can be described as the trace of a Taylor expandable function. Two nonlinear functions that use *dicke_function_trace* and are already implemented are *purity_dicke*, to calculate the purity of a density matrix in the Dicke basis, and *entropy_vn_dicke*, which can be used to calculate the Von Neumann entropy.

More functions relative to the *qutip.piqs* module can be found at *API documentation*. Attributes to the `qutip.piqs.Dicke` and `qutip.piqs.Pim` class can also be found there.

3.5.12 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`.

```
options = Options()
```

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

```
print(options)
```

Output:

```
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:           2
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.store_final_state	False	Whether or not to store the final state of the evolution.
options.store_states	False	Whether or not to store the state vectors or density matrices.
options.rhs_filename	None	RHS filename when using compiled time-dependent Hamiltonians.
options.rhs_reuse	False	Reuse compiled RHS function. Useful for repetitive tasks.
options.rhs_with_state	False	Whether or not to include the state in the Hamiltonian function callback signature.
options.num_cpus	installed num of processors	Integer number of cpus used by mcsolve.
options.seeds	None	Array containing random number seeds for mcsolver.
options.norm_tol	1e-6	Tolerance used when finding wavefunction norm in mcsolve.
options.norm_steps	5	Max. number of steps used to find wavefunction's norm to within norm_tol in mcsolve.
options.steady_state_average	False	Include an estimation of the steady state in mcsolve.
options.ntraj	500	Number of trajectories in stochastic solvers.
options.average_expect	True	Average expectation values over trajectories.
options.average_states	False	Average of the states over trajectories.
options.openmp_threads	installed num of processors	Number of OPENMP threads to use.
options.use_openmp	None	Use OPENMP for sparse matrix vector multiplication.

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,

```
options = Options()
options.num_cpus = 3
options.atol = 1e-10
```

or one can use an inline method,

```
options = Options(num_cpus=4, atol=1e-10)
```

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

```
print(options)
```

Output:

```
Options:
-----------
atol:               1e-10
```

(continues on next page)

(continued from previous page)

```
rtol:               1e-06
method:             adams
order:              12
nsteps:             1000
first_step:         0
min_step:           0
max_step:           0
tidy:               True
num_cpus:           4
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 Hierarchical Equations of Motion

3.6.1 Introduction

The Hierarchical Equations of Motion (HEOM) method was originally developed by Tanimura and Kubo [TK89] in the context of physical chemistry to ''exactly" solve a quantum system in contact with a bosonic environment, encapsulated in the Hamiltonian:

$$H = H_s + \sum_k \omega_k a_k^\dagger a_k + \hat{Q} \sum_k g_k \left(a_k + a_k^\dagger\right).$$

As in other solutions to this problem, the properties of the bath are encapsulated by its temperature and its spectral density,

$$J(\omega) = \pi \sum_k g_k^2 \delta(\omega - \omega_k).$$

In the HEOM, for bosonic baths, one typically chooses a Drude-Lorentz spectral density:

$$J_D = \frac{2\lambda\gamma\omega}{(\gamma^2 + \omega^2)},$$

or an under-damped Brownian motion spectral density:

$$J_U = \frac{\alpha^2\Gamma\omega}{[(\omega_c^2 - \omega^2)^2 + \Gamma^2\omega^2]}.$$

Given the spectral density, the HEOM requires a decomposition of the bath correlation functions in terms of exponentials. In *Bosonic Environments* we describe how this is done with code examples, and how these expansions are passed to the solver.

In addition to support for bosonic environments, QuTiP also provides support for feriomic environments which is described in *Fermionic Environments*.

Both bosonic and fermionic environments are supported via a single solver, `HEOMSolver`, that supports solving for both dynamics and steady-states.

3.6.2 Bosonic Environments

In this section we consider a simple two-level system coupled to a Drude-Lorentz bosonic bath. The system Hamiltonian, H_{sys}, and the bath spectral density, J_D, are

$$H_{sys} = \frac{\epsilon\sigma_z}{2} + \frac{\Delta\sigma_x}{2}$$
$$J_D = \frac{2\lambda\gamma\omega}{(\gamma^2 + \omega^2)},$$

We will demonstrate how to describe the bath using two different expansions of the spectral density correlation function (Matsubara's expansion and a Padé expansion), how to evolve the system in time, and how to calculate the steady state.

First we will do this in the simplest way, using the built-in implementations of the two bath expansions, `DrudeLorentzBath` and `DrudeLorentzPadeBath`. We will do this both with a truncated expansion and show how to include an approximation to all of the remaining terms in the bath expansion.

Afterwards, we will show how to calculate the bath expansion coefficients and to use those coefficients to construct your own bath description so that you can implement your own bosonic baths.

Finally, we will demonstrate how to simulate a system coupled to multiple independent baths, as occurs, for example, in certain photosynthesis processes.

A notebook containing a complete example similar to this one implemented in BoFiN can be found in example notebook 1a.

Describing the system and bath

First, let us construct the system Hamiltonian, H_{sys}, and the initial system state, `rho0`:

```
from qutip import basis, sigmax, sigmaz

# The system Hamiltonian:
eps = 0.5  # energy of the 2-level system
Del = 1.0  # tunnelling term
H_sys = 0.5 * eps * sigmaz() + 0.5 * Del * sigmax()

# Initial state of the system:
rho0 = basis(2,0) * basis(2,0).dag()
```

Now let us describe the bath properties:

```
# Bath properties:
gamma = 0.5  # cut off frequency
lam = 0.1  # coupling strength
T = 0.5  # temperature

# System-bath coupling operator:
Q = sigmaz()
```

where γ (`gamma`), λ (`lam`) and T are the parameters of a Drude-Lorentz bath, and `Q` is the coupling operator between the system and the bath.

We may the pass these parameters to either *DrudeLorentzBath* or *DrudeLorentzPadeBath* to construct an expansion of the bath correlations:

```
from qutip.nonmarkov.heom import DrudeLorentzBath
from qutip.nonmarkov.heom import DrudeLorentzPadeBath

# Number of expansion terms to retain:
Nk = 2

# Matsubara expansion:
bath = DrudeLorentzBath(Q, lam, gamma, T, Nk)

# Padé expansion:
bath = DrudeLorentzPadeBath(Q, lam, gamma, T, Nk)
```

Where `Nk` is the number of terms to retain within the expansion of the bath.

System and bath dynamics

Now we are ready to construct a solver:

```
from qutip.nonmarkov.heom import HEOMSolver
from qutip import Options

max_depth = 5  # maximum hierarchy depth to retain
options = Options(nsteps=15_000)

solver = HEOMSolver(H_sys, bath, max_depth=max_depth, options=options)
```

and to calculate the system evolution as a function of time:

```
tlist = [0, 10, 20]  # times to evaluate the system state at
result = solver.run(rho0, tlist)
```

The `max_depth` parameter determines how many levels of the hierarchy to retain. As a first approximation hierarchy depth may be thought of as similar to the order of Feynman Diagrams (both classify terms by increasing number of interactions).

The `result` is a standard QuTiP results object with the attributes:

- `times`: the times at which the state was evaluated (i.e. `tlist`)
- `states`: the system states at each time
- `expect`: the values of each `e_ops` at each time
- `ado_states`: see below (an instance of *HierarchyADOsState*)

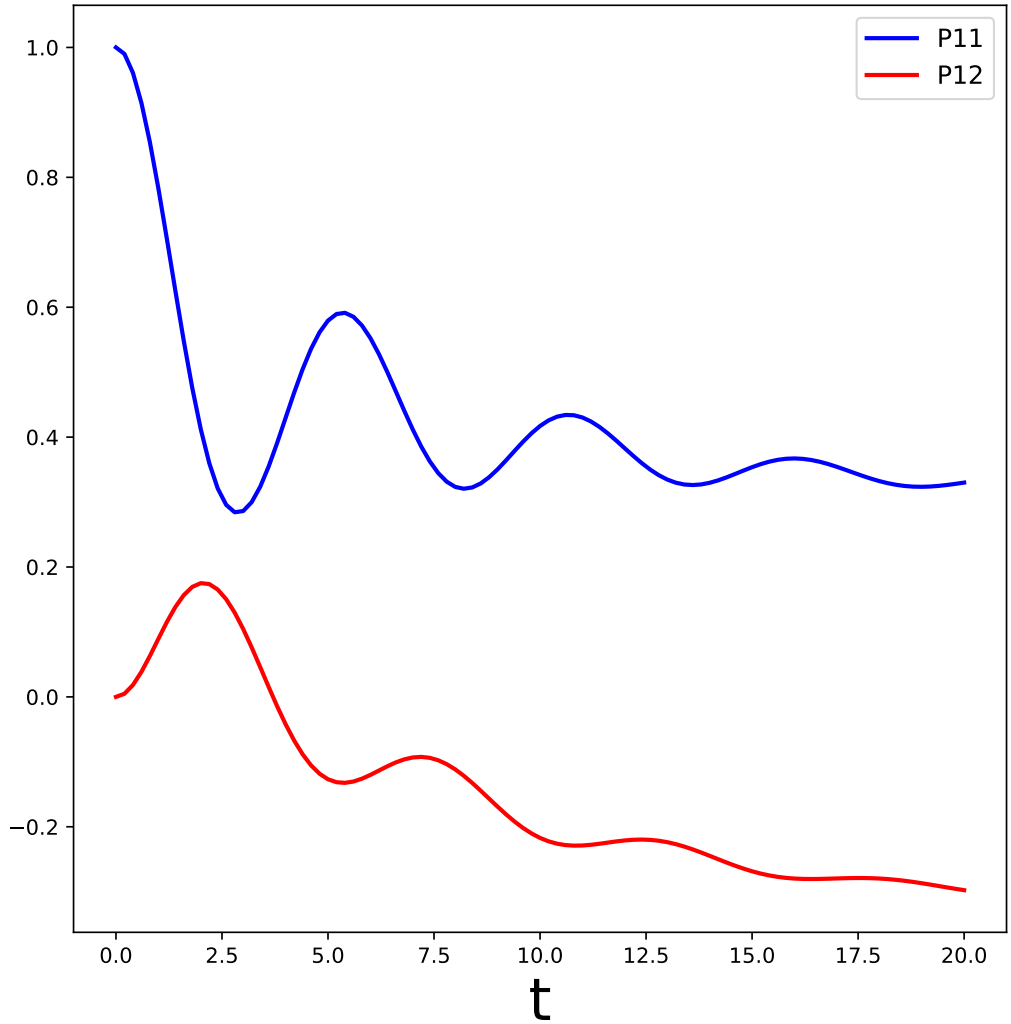
If `ado_return=True` is passed to `.run(...)` the full set of auxilliary density operators (ADOs) that make up the hierarchy at each time will be returned as `.ado_states`. We will describe how to use these to determine other properties, such as system-bath currents, later in the fermionic guide (see *Determining currents*).

If one has a full set of ADOs from a previous call of `.run(...)` you may supply it as the initial state of the solver by calling `.run(result.ado_states[-1], tlist, ado_init=True)`.

As with other QuTiP solvers, if expectation operators or functions are supplied using `.run(..., e_ops=[...])` the expectation values are available in `result.expect`.

Below we run the solver again, but use `e_ops` to store the expectation values of the population of the system states and the coherence:

```
# Define the operators that measure the populations of the two
# system states:
P11p = basis(2,0) * basis(2,0).dag()
P22p = basis(2,1) * basis(2,1).dag()

# Define the operator that measures the 0, 1 element of density matrix
# (corresonding to coherence):
P12p = basis(2,0) * basis(2,1).dag()

# Run the solver:
tlist = np.linspace(0, 20, 101)
result = solver.run(rho0, tlist, e_ops={"11": P11p, "22": P22p, "12": P12p})

# Plot the results:
fig, axes = plt.subplots(1, 1, sharex=True, figsize=(8,8))
axes.plot(result.times, result.expect["11"], 'b', linewidth=2, label="P11")
axes.plot(result.times, result.expect["12"], 'r', linewidth=2, label="P12")
axes.set_xlabel(r't', fontsize=28)
axes.legend(loc=0, fontsize=12)
```



Steady-state

Using the same solver, we can also determine the steady state of the combined system and bath using:

```
steady_state, steady_ados = solver.steady_state()
```

where `steady_state` is the steady state of the system and `steady_ados` if the steady state of the full hierarchy. The ADO states are described more fully in *Determining currents* and `HierarchyADOsState`.

Matsubara Terminator

When constructing the Drude-Lorentz bath we have truncated the expansion at `Nk = 2` terms and ignore the remaining terms.

However, since the coupling to these higher order terms is comparatively weak, we may consider the interaction with them to be Markovian, and construct an additional Lindbladian term that captures their interaction with the system and the lower order terms in the expansion.

This additional term is called the `terminator` because it terminates the expansion.

The `DrudeLorentzBath` and `DrudeLorentzPadeBath` both provide a means of calculating the terminator for a given expansion:

```
# Matsubara expansion:
bath = DrudeLorentzBath(Q, lam, gamma, T, Nk)

# Padé expansion:
bath = DrudeLorentzPadeBath(Q, lam, gamma, T, Nk)

# Add terminator to the system Liouvillian:
delta, terminator = bath.terminator()
HL = liouvillian(H_sys) + terminator

# Construct solver:
solver = HEOMSolver(HL, bath, max_depth=max_depth, options=options)
```

This captures the Markovian effect of the remaining terms in the expansion without having to fully model many more terms.

The value `delta` is an approximation to the strength of the effect of the remaining terms in the expansion (i.e. how strongly the terminator is coupled to the rest of the system).

Matsubara expansion coefficients

So far we have relied on the built-in `DrudeLorentzBath` to construct the Drude-Lorentz bath expansion for us. Now we will calculate the coefficients ourselves and construct a `BosonicBath` directly. A similar procedure can be used to apply `HEOMSolver` to any bosonic bath for which we can calculate the expansion coefficients.

The real and imaginary parts of the correlation function, $C(t)$, for the bosonic bath is expanded in an expontential series:

$$C(t) = C_{real}(t) + iC_{imag}(t)$$

$$C_{real}(t) = \sum_{k=0}^{\infty} c_{k,real} e^{-\nu_{k,real} t}$$

$$C_{imag}(t) = \sum_{k=0}^{\infty} c_{k,imag} e^{-\nu_{k,imag} t}$$

In the specific case of Matsubara expansion for the Drude-Lorentz bath, the coefficients of this expansion are, for the real part, $C_{real}(t)$:

$$\nu_{k,real} = \begin{cases} \gamma & k = 0 \\ 2\pi k/\beta & k \geq 1 \end{cases}$$

$$c_{k,real} = \begin{cases} \lambda\gamma[\cot(\beta\gamma/2) - i] & k = 0 \\ \frac{4\lambda\gamma\nu_k}{(\nu_k^2 - \gamma^2)\beta} & k \geq 1 \end{cases}$$

and the imaginary part, $C_{imag}(t)$:

$$\nu_{k,imag} = \begin{cases} \gamma & k = 0 \\ 0 & k \geq 1 \end{cases}$$

$$c_{k,imag} = \begin{cases} -\lambda\gamma & k = 0 \\ 0 & k \geq 1 \end{cases}$$

And now the same numbers calculated in Python:

```
# Convenience functions and parameters:

def cot(x):
    return 1. / np.tan(x)

beta = 1. / T

# Number of expansion terms to calculate:
Nk = 2

# C_real expansion terms:
ck_real = [lam * gamma / np.tan(gamma / (2 * T))]
ck_real.extend([
    (8 * lam * gamma * T * np.pi * k * T /
        ((2 * np.pi * k * T)**2 - gamma**2))
    for k in range(1, Nk + 1)
])
vk_real = [gamma]
vk_real.extend([2 * np.pi * k * T for k in range(1, Nk + 1)])

# C_imag expansion terms (this is the full expansion):
ck_imag = [lam * gamma * (-1.0)]
vk_imag = [gamma]
```

After all that, constructing the bath is very straight forward:

```
from qutip.nonmarkov.heom import BosonicBath

bath = BosonicBath(Q, ck_real, vk_real, ck_imag, vk_imag)
```

And we're done!

The *BosonicBath* can be used with the *HEOMSolver* in exactly the same way as the baths we constructed previously using the built-in Drude-Lorentz bath expansions.

Multiple baths

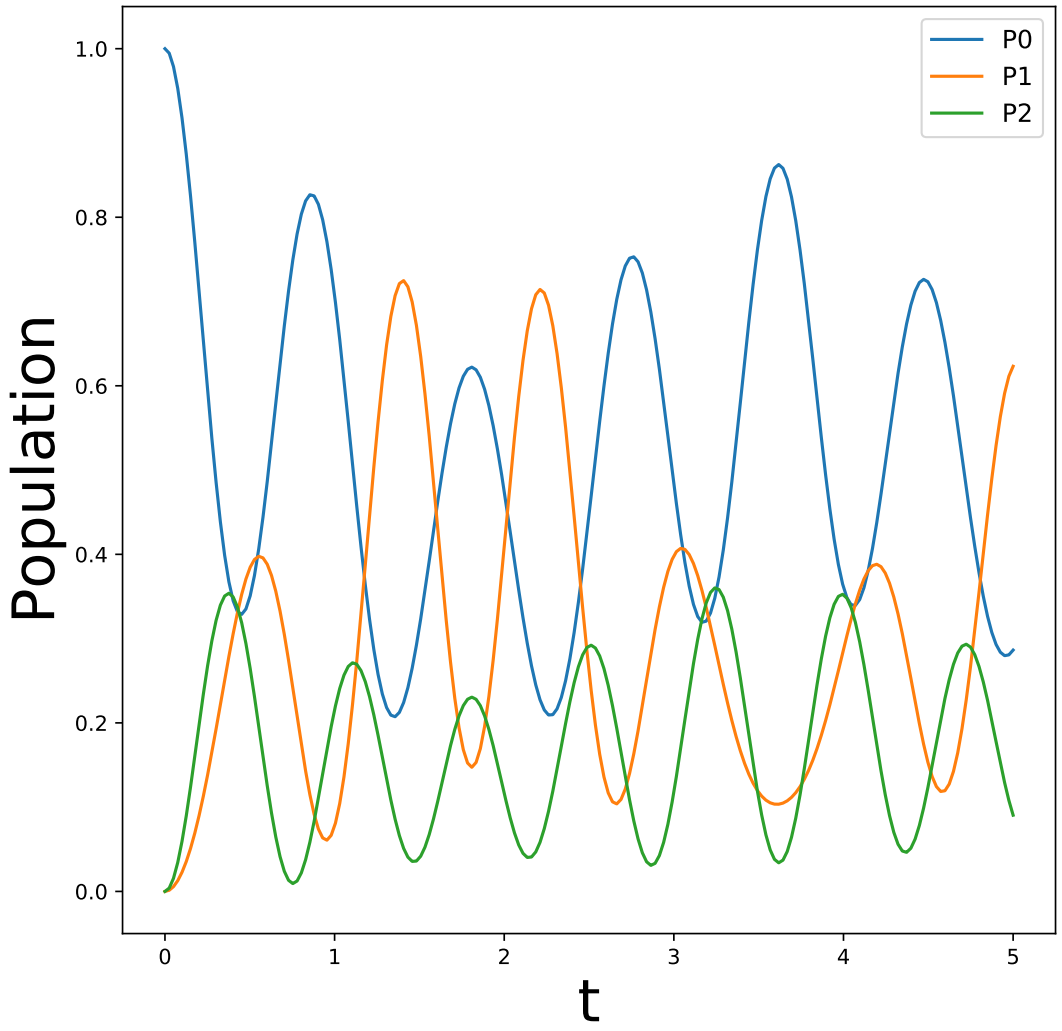
The *HEOMSolver* supports having a system interact with multiple environments. All that is needed is to supply a list of baths instead of a single bath.

In the example below we calculate the evolution of a small system where each basis state of the system interacts with a separate bath. Such an arrangement can model, for example, the Fenna–Matthews–Olson (FMO) pigment-protein complex which plays an important role in photosynthesis (for a full FMO example see the notebook https://github.com/tehruhn/bofin/blob/main/examples/example-2-FMO-example.ipynb).

For each bath expansion, we also include the terminator in the system Liouvillian.

At the end, we plot the populations of the system states as a function of time, and show the long-time beating of quantum state coherence that occurs:

```
# The size of the system:
N_sys = 3

def proj(i, j):
    """ A helper function for creating an interaction operator. """
    return basis(N_sys, i) * basis(N_sys, j).dag()

# Construct one bath for each system state:
baths = []
for i in range(N_sys):
    Q = proj(i, i)
    baths.append(DrudeLorentzBath(Q, lam, gamma, T, Nk))

# Construct the system Liouvillian from the system Hamiltonian and
# bath expansion terminators:
H_sys = sum((i + 0.5) * eps * proj(i, i) for i in range(N_sys))
H_sys += sum(
  (i + j + 0.5) * Del * proj(i, j)
  for i in range(N_sys) for j in range(N_sys)
  if i != j
)
HL = liouvillian(H_sys) + sum(bath.terminator()[1] for bath in baths)

# Construct the solver (pass a list of baths):
solver = HEOMSolver(HL, baths, max_depth=max_depth, options=options)

# Run the solver:
rho0 = basis(N_sys, 0) * basis(N_sys, 0).dag()
tlist = np.linspace(0, 5, 200)
e_ops = {
    f"P{i}": proj(i, i)
    for i in range(N_sys)
}
result = solver.run(rho0, tlist, e_ops=e_ops)

# Plot populations:
fig, axes = plt.subplots(1, 1, sharex=True, figsize=(8,8))
for label, values in result.expect.items():
    axes.plot(result.times, values, label=label)
axes.set_xlabel(r't', fontsize=28)
axes.set_ylabel(r"Population", fontsize=28)
axes.legend(loc=0, fontsize=12)
```



3.6.3 Fermionic Environments

Here we model a single fermion coupled to two electronic leads or reservoirs (e.g., this can describe a single quantum dot, a molecular transistor, etc). The system hamiltonian, H_{sys}, and bath spectral density, J_D, are

$$H_{sys} = c^\dagger c$$
$$J_D = \frac{\Gamma W^2}{(w-\mu)^2 + W^2},$$

We will demonstrate how to describe the bath using two different expansions of the spectral density correlation function (Matsubara's expansion and a Padé expansion), how to evolve the system in time, and how to calculate the steady state.

Since our fermion is coupled to two reservoirs, we will construct two baths – one for each reservoir or lead – and call them the left (L) and right (R) baths for convenience. Each bath will have a different chemical potential μ which we will label μ_L and μ_R.

First we will do this using the built-in implementations of the bath expansions, `LorentzianBath` and `LorentzianPadeBath`.

Afterwards, we will show how to calculate the bath expansion coefficients and to use those coefficients to construct your own bath description so that you can implement your own fermionic baths.

Our implementation of fermionic baths primarily follows the definitions used by Christian Schinabeck in his dissertation (https://opus4.kobv.de/opus4-fau/files/10984/DissertationChristianSchinabeck.pdf) and related publications.

A notebook containing a complete example similar to this one implemented in BoFiN can be found in example notebook 4b.

Describing the system and bath

First, let us construct the system Hamiltonian, H_{sys}, and the initial system state, `rho0`:

```
from qutip import basis, destroy

# The system Hamiltonian:
e1 = 1.  # site energy
H_sys = e1 * destroy(2).dag() * destroy(2)

# Initial state of the system:
rho0 = basis(2,0) * basis(2,0).dag()
```

Now let us describe the bath properties:

```
# Shared bath properties:
gamma = 0.01   # coupling strength
W = 1.0  # cut-off
T = 0.025851991  # temperature
beta = 1. / T

# Chemical potentials for the two baths:
mu_L = 1.
mu_R = -1.

# System-bath coupling operator:
Q = destroy(2)
```

where Γ (`gamma`), W and T are the parameters of an Lorentzian bath, μ_L (`mu_L`) and μ_R (`mu_R`) are the chemical potentials of the left and right baths, and `Q` is the coupling operator between the system and the baths.

We may the pass these parameters to either `LorentzianBath` or `LorentzianPadeBath` to construct an expansion of the bath correlations:

```
from qutip.nonmarkov.heom import LorentzianBath
from qutip.nonmarkov.heom import LorentzianPadeBath

# Number of expansion terms to retain:
Nk = 2

# Matsubara expansion:
bath_L = LorentzianBath(Q, gamma, W, mu_L, T, Nk, tag="L")
bath_R = LorentzianBath(Q, gamma, W, mu_R, T, Nk, tag="R")

# Padé expansion:
bath_L = LorentzianPadeBath(Q, gamma, W, mu_L, T, Nk, tag="L")
bath_R = LorentzianPadeBath(Q, gamma, W, mu_R, T, Nk, tag="R")
```

Where `Nk` is the number of terms to retain within the expansion of the bath.

Note that we haved labelled each bath with a tag (either "L" or "R") so that we can identify the exponents from individual baths later when calculating the currents between the system and the bath.

System and bath dynamics

Now we are ready to construct a solver:

```
from qutip.nonmarkov.heom import HEOMSolver
from qutip import Options

max_depth = 5  # maximum hierarchy depth to retain
options = Options(nsteps=15_000)
baths = [bath_L, bath_R]

solver = HEOMSolver(H_sys, baths, max_depth=max_depth, options=options)
```

and to calculate the system evolution as a function of time:

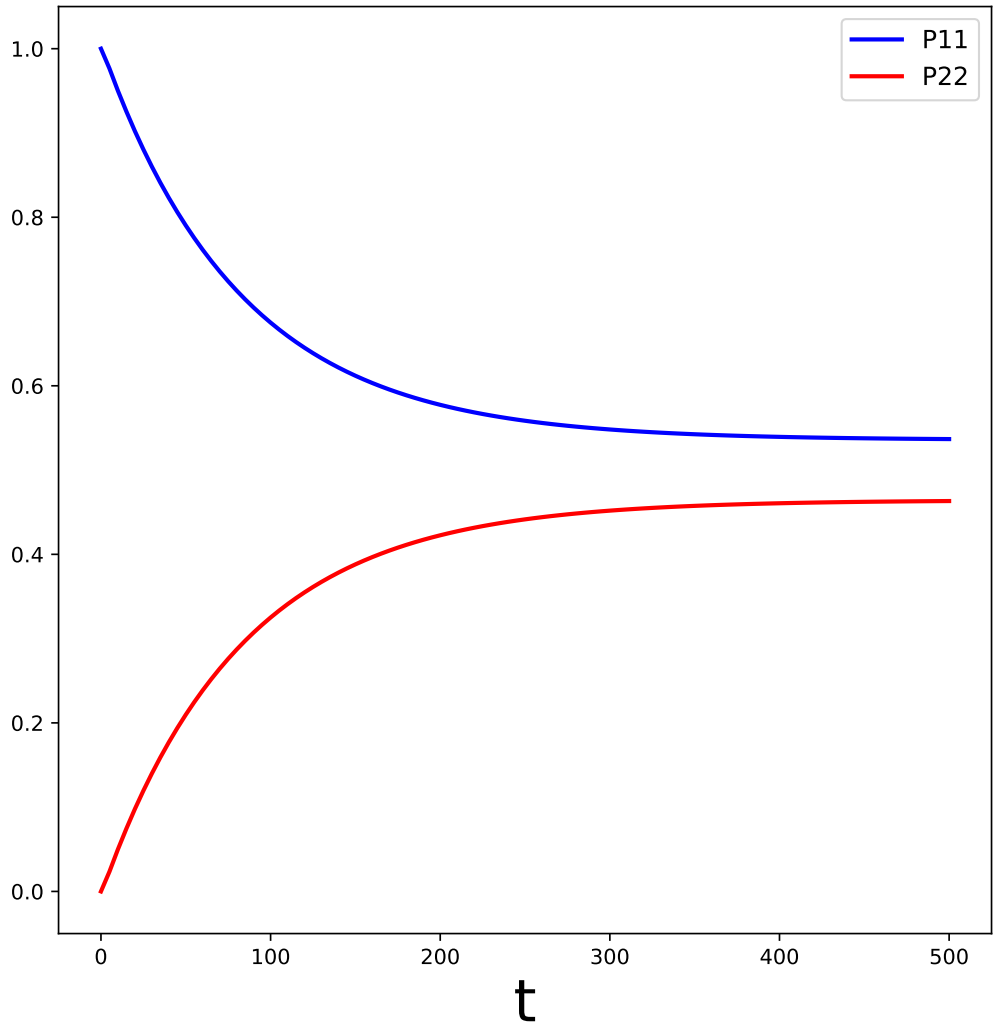
```
tlist = [0, 10, 20]  # times to evaluate the system state at
result = solver.run(rho0, tlist)
```

As in the bosonic case, the `max_depth` parameter determines how many levels of the hierarchy to retain.

As in the bosonic case, we can specify `e_ops` in order to retrieve the expectation values of operators at each given time. See *System and bath dynamics* for a fuller description of the returned `result` object.

Below we run the solver again, but use `e_ops` to store the expectation values of the population of the system states:

```
# Define the operators that measure the populations of the two
# system states:
P11p = basis(2,0) * basis(2,0).dag()
P22p = basis(2,1) * basis(2,1).dag()

# Run the solver:
tlist = np.linspace(0, 500, 101)
result = solver.run(rho0, tlist, e_ops={"11": P11p, "22": P22p})

# Plot the results:
fig, axes = plt.subplots(1, 1, sharex=True, figsize=(8,8))
axes.plot(result.times, result.expect["11"], 'b', linewidth=2, label="P11")
axes.plot(result.times, result.expect["22"], 'r', linewidth=2, label="P22")
axes.set_xlabel(r't', fontsize=28)
axes.legend(loc=0, fontsize=12)
```



The plot above is not very exciting. What we would really like to see in this case are the currents between the system and the two baths. We will plot these in the next section using the auxiliary density operators (ADOs) returned by the solver.

Determining currents

The currents between the system and a fermionic bath may be calculated from the first level auxiliary density operators (ADOs) associated with the exponents of that bath.

The contribution to the current into a given bath from each exponent in that bath is:

$$\mathrm{ContributionfromExponent} = \pm i \mathrm{Tr}(Q^{\pm} \cdot A)$$

where the $\pm$ sign is the sign of the exponent (see the description later in *Padé expansion coefficients*) and $Q^{\pm}$ is Q for + exponents and $Q^{\dagger}$ for − exponents.

The first-level exponents for the left bath are retrieved by calling `.filter(tags=["L"])` on `ado_state` which is an instance of *`HierarchyADOsState`* and also provides access to the methods of *`HierarchyADOs`* which describes the structure of the hierarchy for a given problem.

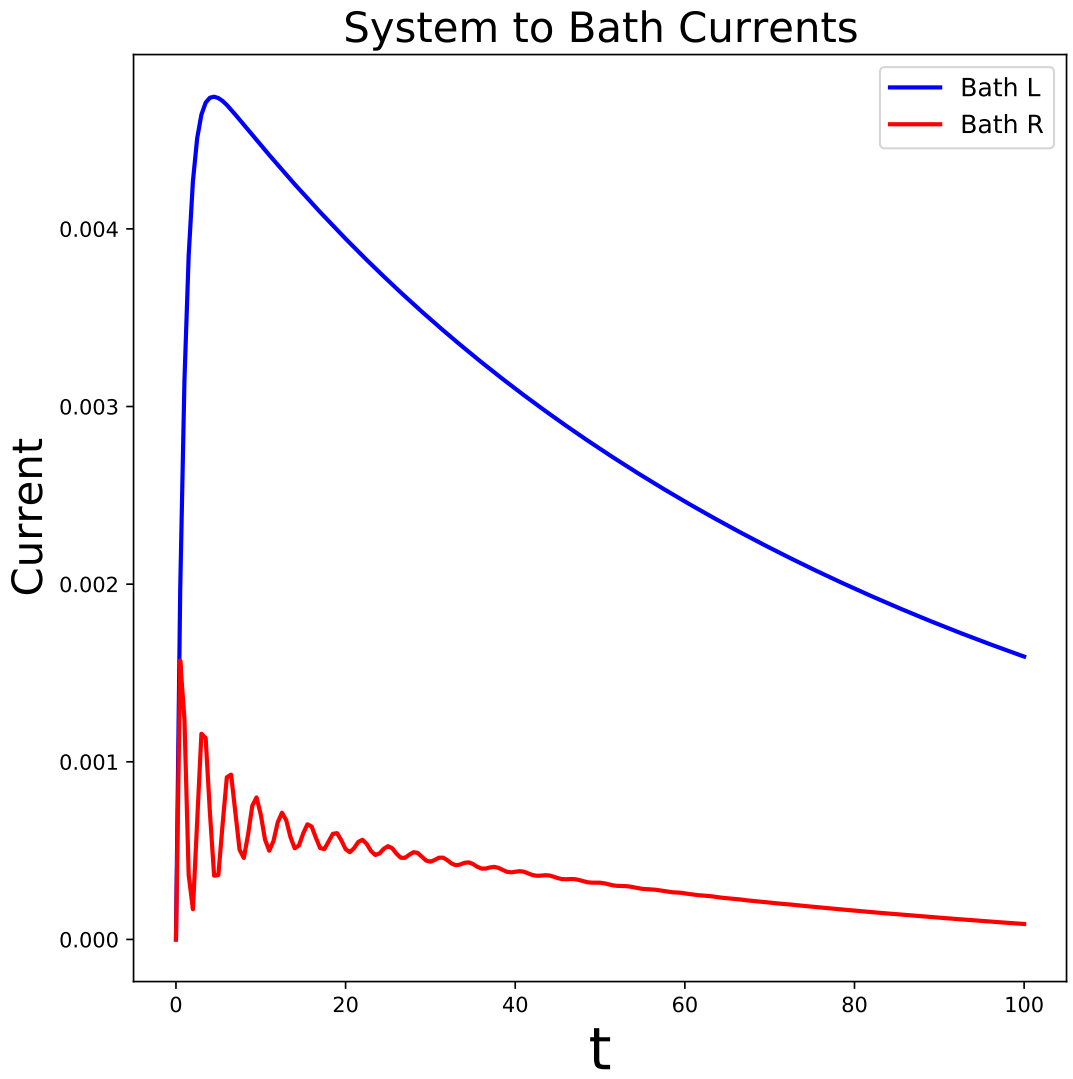
Here the tag "L" matches the tag passed when constructing `bath_L` earlier in this example.

Similarly, we may calculate the current to the right bath from the exponents tagged with "R".

```
def exp_current(aux, exp):
    """ Calculate the current for a single exponent. """
    sign = 1 if exp.type == exp.types["+"] else -1
    op = exp.Q if exp.type == exp.types["+"] else exp.Q.dag()
    return 1j * sign * (op * aux).tr()

def heom_current(tag, ado_state):
    """ Calculate the current between the system and the given bath. """
    level_1_ados = [
        (ado_state.extract(label), ado_state.exps(label)[0])
        for label in ado_state.filter(tags=[tag])
    ]
    return np.real(sum(exp_current(aux, exp) for aux, exp in level_1_ados))

heom_left_current = lambda t, ado_state: heom_current("L", ado_state)
heom_right_current = lambda t, ado_state: heom_current("R", ado_state)
```

Once we have defined functions for retrieving the currents for the baths, we can pass them to `e_ops` and plot the results:

```
# Run the solver (returning ADO states):
tlist = np.linspace(0, 100, 201)
result = solver.run(rho0, tlist, e_ops={
    "left_currents": heom_left_current,
    "right_currents": heom_right_current,
})

# Plot the results:
fig, axes = plt.subplots(1, 1, sharex=True, figsize=(8,8))
axes.plot(
    result.times, result.expect["left_currents"], 'b',
    linewidth=2, label=r"Bath L",
)
axes.plot(
    result.times, result.expect["right_currents"], 'r',
    linewidth=2, label="Bath R",
)
axes.set_xlabel(r't', fontsize=28)
axes.set_ylabel(r'Current', fontsize=20)
axes.set_title(r'System to Bath Currents', fontsize=20)
axes.legend(loc=0, fontsize=12)
```



And now we have a more interesting plot that shows the currents to the left and right baths decaying towards their steady states!

In the next section, we will calculate the steady state currents directly.

Steady state currents

Using the same solver, we can also determine the steady state of the combined system and bath using:

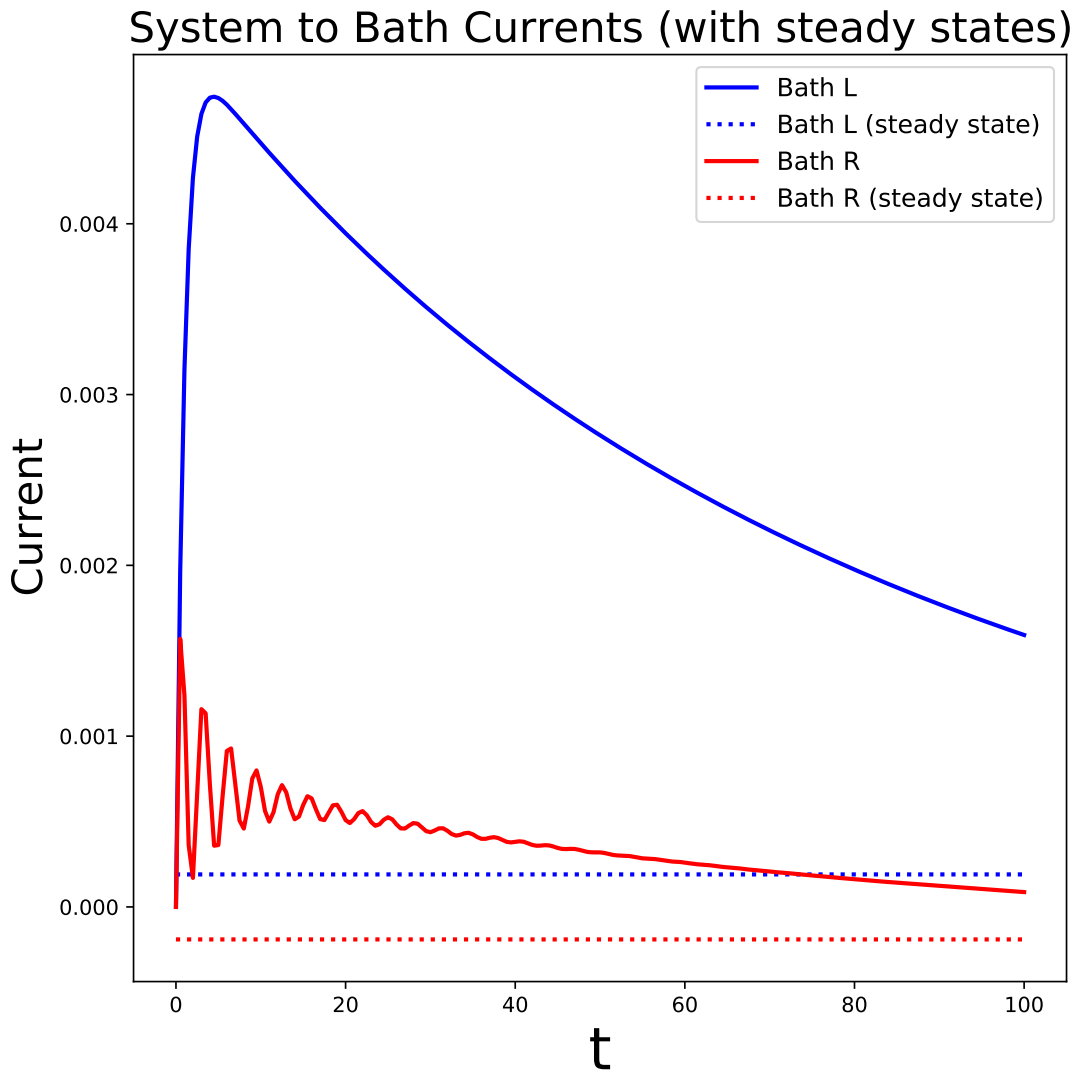
```
steady_state, steady_ados = solver.steady_state()
```

and calculate the steady state currents to the two baths from `steady_ados` using the same `heom_current` function we defined previously:

```
steady_state_current_left = heom_current("L", steady_ados)
steady_state_current_right = heom_current("R", steady_ados)
```

Now we can add the steady state currents to the previous plot:

```
# Plot the results and steady state currents:
fig, axes = plt.subplots(1, 1, sharex=True, figsize=(8,8))
axes.plot(
    result.times, result.expect["left_currents"], 'b',
    linewidth=2, label=r"Bath L",
)
axes.plot(
    result.times, [steady_state_current_left] * len(result.times), 'b:',
    linewidth=2, label=r"Bath L (steady state)",
)
axes.plot(
    result.times, result.expect["right_currents"], 'r',
    linewidth=2, label="Bath R",
)
axes.plot(
    result.times, [steady_state_current_right] * len(result.times), 'r:',
    linewidth=2, label=r"Bath R (steady state)",
)
axes.set_xlabel(r't', fontsize=28)
axes.set_ylabel(r'Current', fontsize=20)
axes.set_title(r'System to Bath Currents (with steady states)', fontsize=20)
axes.legend(loc=0, fontsize=12)
```



As you can see, there is still some way to go beyond `t = 100` before the steady state is reached!

Padé expansion coefficients

We now look at how to calculate the correlation expansion coefficients for the Lorentzian spectral density ourselves. Once we have calculated the coefficients we can construct a *FermionicBath* directly from them. A similar procedure can be used to apply *HEOMSolver* to any fermionic bath for which we can calculate the expansion coefficients.

In the fermionic case we must descriminate between the order in which excitations are created within the bath, so we define two different correlation functions, $C_+(t)$, and $C_-(t)$:

$$C^{\sigma}(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} d\omega e^{\sigma i\omega t} J(\omega) f_F[\sigma\beta(\omega-\mu)]$$

where σ is either + or − and, f_F is the Fermi distribution function, and $J(\omega)$ is the Lorentzian spectral density we defined at the start.

The Fermi distribution function is:

$$f_F(x) = (\exp(x)+1)^{-1}$$

As in the bosonic case we can approximate this integral with a Matsubara or Padé expansion. For the Lorentzian bath the Padé expansion converges much more quickly, so we will calculate the Padé expansion coefficients here.

The Padé decomposition approximates the Fermi distribution as:

$$f_F(x) \approx f_F^{\rm approx}(x) = \frac{1}{2} - \sum_{l=0}^{Nk} \frac{2k_l x}{x^2 + \epsilon_l^2}$$

where k_l and ϵ_l are coefficients defined in J. Chem Phys 133, "Efficient on the fly calculation of time correlation functions in computer simulations", and Nk specifies the cut-off in the expansion.

Evaluating the integral for the correlation functions gives:

$$C^{\sigma}(t) \approx \sum_{l=0}^{Nk} \eta^{\sigma,l} e^{-\gamma_{\sigma,l} t}$$

where:

$$\eta_{\sigma,l} = \begin{cases} \frac{\Gamma W}{2} f_F^{approx}(i\beta W) & l = 0 \\ -i \cdot \frac{k_l}{\beta} \cdot \frac{\Gamma W^2}{-\frac{\epsilon_l^2}{\beta^2} + W^2} & l \neq 0 \end{cases}$$

$$\gamma_{\sigma,l} = \begin{cases} W - \sigma i\mu & l = 0 \\ \frac{\epsilon_l}{\beta} - \sigma i\mu & l \neq 0 \end{cases}$$

and $\beta = \frac{1}{T}$.

And now we calculate the same numbers in Python:

```
# Imports
from numpy.linalg import eigvalsh

# Convenience functions and parameters:
def deltafun(j, k):
    """ Kronecker delta function. """
    return 1.0 if j == k else 0.

def f_approx(x, Nk):
    """ Padé approxmation to Fermi distribution. """
    f = 0.5
    for ll in range(1, Nk + 1):
        # kappa and epsilon are calculated further down
        f = f - 2 * kappa[ll] * x / (x**2 + epsilon[ll]**2)
    return f

def kappa_epsilon(Nk):
    """ Calculate kappa and epsilon coefficients. """

    alpha = np.zeros((2 * Nk, 2 * Nk))
    for j in range(2 * Nk):
        for k in range(2 * Nk):
            alpha[j][k] = (
                (deltafun(j, k + 1) + deltafun(j, k - 1))
                / np.sqrt((2 * (j + 1) - 1) * (2 * (k + 1) - 1))
            )

    eps = [-2. / val for val in eigvalsh(alpha)[:Nk]]

    alpha_p = np.zeros((2 * Nk - 1, 2 * Nk - 1))
    for j in range(2 * Nk - 1):
        for k in range(2 * Nk - 1):
            alpha_p[j][k] = (
```

(continues on next page)

(continued from previous page)

```
                (deltafun(j, k + 1) + deltafun(j, k - 1))
                / np.sqrt((2 * (j + 1) + 1) * (2 * (k + 1) + 1))
            )

    chi = [-2. / val for val in eigvalsh(alpha_p)[:Nk - 1]]

    eta_list = [
        0.5 * Nk * (2 * (Nk + 1) - 1) * (
            np.prod([chi[k]**2 - eps[j]**2 for k in range(Nk - 1)]) /
            np.prod([
                eps[k]**2 - eps[j]**2 + deltafun(j, k) for k in range(Nk)
            ])
        )
        for j in range(Nk)
    ]

    kappa = [0] + eta_list
    epsilon = [0] + eps

    return kappa, epsilon

kappa, epsilon = kappa_epsilon(Nk)

# Phew, we made it to function that calculates the coefficients for the
# correlation function expansions:

def C(sigma, mu, Nk):
    """ Calculate the expansion coefficients for C_\sigma. """
    beta = 1. / T
    ck = [0.5 * gamma * W * f_approx(1.0j * beta * W, Nk)]
    vk = [W - sigma * 1.0j * mu]
    for ll in range(1, Nk + 1):
        ck.append(
            -1.0j * (kappa[ll] / beta) * gamma * W**2
            / (-(epsilon[ll]**2 / beta**2) + W**2)
        )
        vk.append(epsilon[ll] / beta - sigma * 1.0j * mu)
    return ck, vk

ck_plus_L, vk_plus_L = C(1.0, mu_L, Nk)  # C_+, left bath
ck_minus_L, vk_minus_L = C(-1.0, mu_L, Nk)  # C_-, left bath

ck_plus_R, vk_plus_R = C(1.0, mu_R, Nk)  # C_+, right bath
ck_minus_R, vk_minus_R = C(-1.0, mu_R, Nk)  # C_-, right bath
```

Finally we are ready to construct the *FermionicBath*:

```
from qutip.nonmarkov.heom import FermionicBath

# Padé expansion:
bath_L = FermionicBath(Q, ck_plus_L, vk_plus_L, ck_minus_L, vk_minus_L)
bath_R = FermionicBath(Q, ck_plus_R, vk_plus_R, ck_minus_R, vk_minus_R)
```

And we're done!

The *FermionicBath* can be used with the *HEOMSolver* in exactly the same way as the baths we constructed previously using the built-in Lorentzian bath expansions.

3.6.4 Previous implementations

The current HEOM implementation in QuTiP is the latest in a succession of HEOM implementations by various contributors:

HSolverDL

The original HEOM solver was implemented by Neill Lambert, Anubhav Vardhan, and Alexander Pitchford and is still available as *qutip.nonmarkov.dlheom_solver.HSolverDL* and only directly provided support for the Drude-Lorentz bath although there was the possibility of sub-classing the solver to implement other baths.

A compatible interface using the current implementation is available under the same name in *qutip.nonmarkov.heom.HSolverDL*.

BoFiN-HEOM

BoFiN-HEOM (the bosonic and fermionic HEOM solver) was a much more flexible re-write of the original QuTiP `HSolverDL` that added support for both bosonic and fermionic baths and for baths to be specified directly via their correlation function expansion coefficients. Its authors were Neill Lambert, Tarun Raheja, Shahnawaz Ahmed, and Alexander Pitchford.

BoFiN was written outside of QuTiP and is can still be found in its original repository at https://github.com/tehruhn/bofin.

The construction of the right-hand side matrix for BoFiN was slow, so BoFiN-fast, a hybrid C++ and Python implementation, was written that performed the right-hand side construction in C++. It was otherwise identical to the pure Python version. BoFiN-fast can be found at https://github.com/tehruhn/bofin_fast.

BoFiN also came with an extensive set of example notebooks that are available at https://github.com/tehruhn/bofin/tree/main/examples.

Current implementation

The current implementation is a rewrite of BoFiN in pure Python. It's right-hand side construction has similar speed to BoFiN-fast, but is written in pure Python. Built-in implementations of a variety of different baths are provided, and a single solver is used for both fermionic and bosonic baths. Multiple baths of the same kind (either fermionic or bosonic) may be specified in a single problem, and there is good support for working with the auxiliary density operator (ADO) state and extracting information from it.

The code was written by Neill Lambert and Simon Cross.

3.6.5 References

3.7 Solving for Steady-State Solutions

3.7.1 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.

3.7.2 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*).

3.7.3 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 precondioning 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.

3.7.4 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'`.
maxiter	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.

3.7.5 Example: Harmonic Oscillator in Thermal Bath

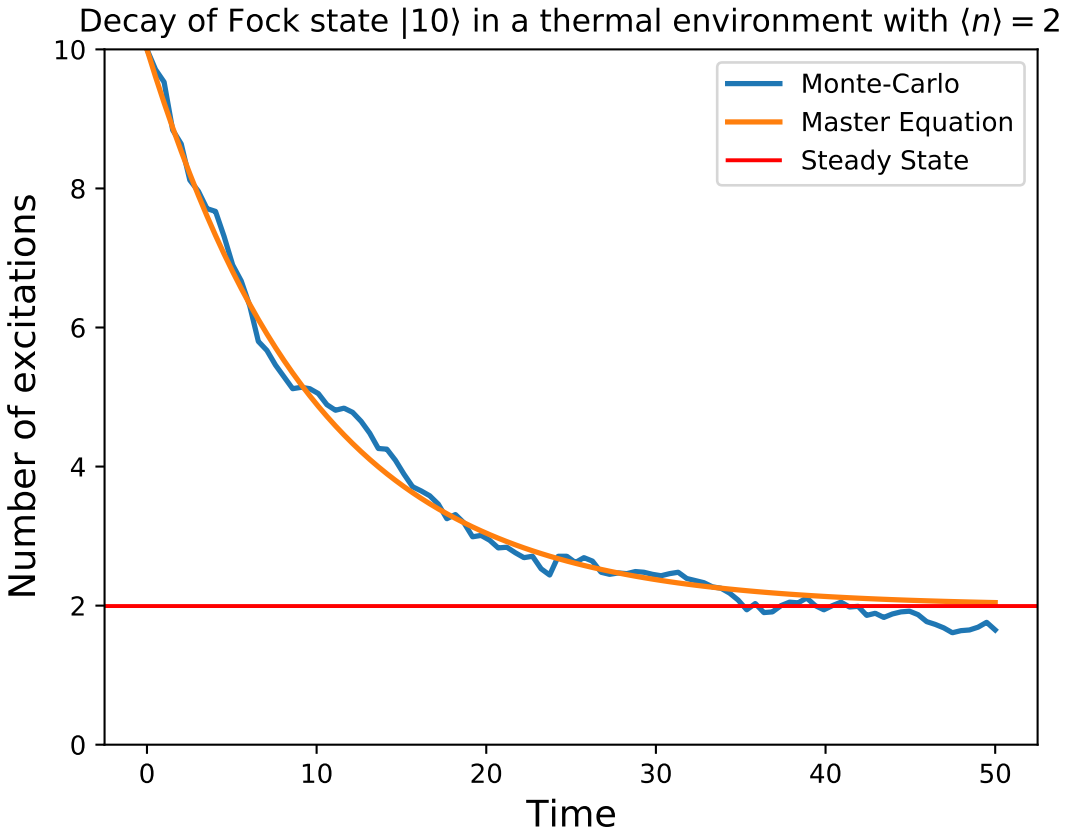
A simple example of a system that reaches a steady state is a harmonic oscillator coupled to a thermal environment. Below we consider a harmonic oscillator, initially in the $|10\rangle$ number state, and weakly coupled to a thermal environment characterized by an average particle expectation value of $\langle n\rangle = 2$. 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.

```
import numpy as np
import matplotlib.pyplot as plt

import qutip

# Define paramters
N = 20  # number of basis states to consider
```

(continues on next page)

(continued from previous page)

```
a = qutip.destroy(N)
H = a.dag() * a
psi0 = qutip.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 = qutip.steadystate(H, c_op_list)
# find expectation value for particle number in steady state
fexpt = qutip.expect(a.dag() * a, final_state)

tlist = np.linspace(0, 50, 100)
# monte-carlo
mcdata = qutip.mcsolve(H, psi0, tlist, c_op_list, [a.dag() * a], ntraj=100)
# master eq.
medata = qutip.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(
    r'Decay of Fock state $\left|10\rangle\right.$'
    r' in a thermal environment with $\langle n\rangle=2$'
)
plt.show()
```



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, $\rho(t) = V(t, t_0)\left\{\rho(t_0)\right\}$, where $V(t, t_0)$ is the propagator defined by the equation of motion. The resulting density matrix can then be used to evaluate the expectation values of arbitrary combinations of *same-time* operators.

To calculate *two-time* correlation functions on the form $\langle A(t+\tau)B(t)\rangle$, we can use the quantum regression theorem (see, e.g., [Gar03]) to write

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

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

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

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

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

QuTiP provides a family of functions that assists in the process of calculating two-time correlation functions. The available functions and their usage is 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`.

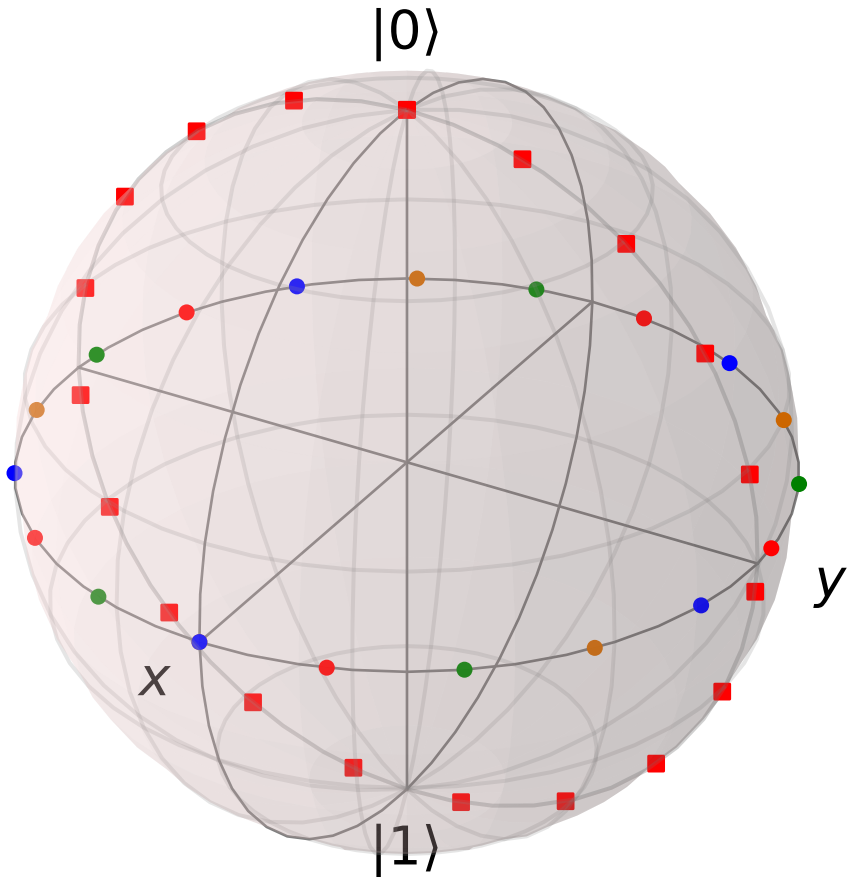
QuTiP function	Correlation function
qutip.correlation.correlation_2op_2t	$\langle A(t+\tau)B(t)\rangle$ or $\langle A(t)B(t+\tau)\rangle$.
qutip.correlation.correlation_2op_1t	$\langle A(\tau)B(0)\rangle$ or $\langle A(0)B(\tau)\rangle$.
qutip.correlation.correlation_3op_1t	$\langle A(0)B(\tau)C(0)\rangle$.
qutip.correlation.correlation_3op_2t	$\langle A(t)B(t+\tau)C(t)\rangle$.

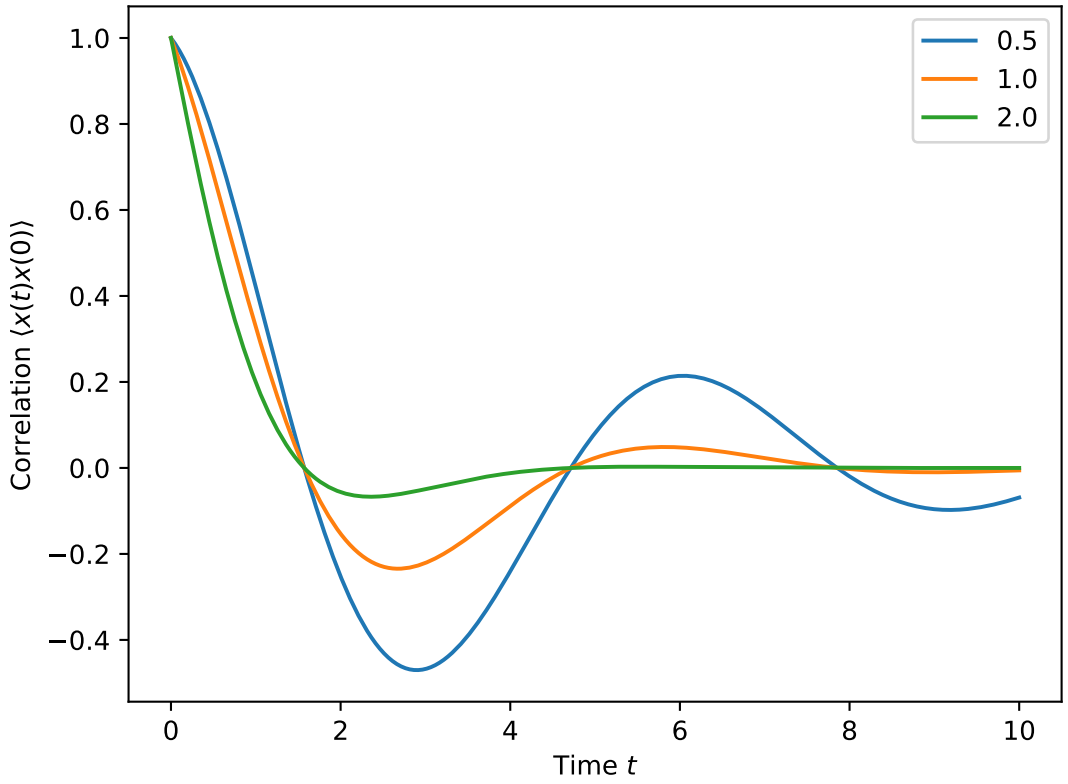
The most common use-case is to calculate correlation functions of the kind $\langle A(\tau)B(0)\rangle$, in which case we use the correlation function solvers that start from the steady state, e.g., the *qutip.correlation.correlation_2op_1t* function. These correlation function solvers return a vector 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.

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

corr1 = correlation_2op_1t(H, None, times, [np.sqrt(0.5) * a], x, x)
corr2 = correlation_2op_1t(H, None, times, [np.sqrt(1.0) * a], x, x)
corr3 = correlation_2op_1t(H, None, times, [np.sqrt(2.0) * a], x, x)

plt.figure()
plt.plot(times, np.real(corr1), times, np.real(corr2), times, np.real(corr3))
plt.legend(['0.5','1.0','2.0'])
plt.xlabel(r'Time $t$')
plt.ylabel(r'Correlation $\left<x(t)x(0)\right>$')
plt.show()
```


|0⟩
y
x
|1⟩



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 one of the time-dependent solvers 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.

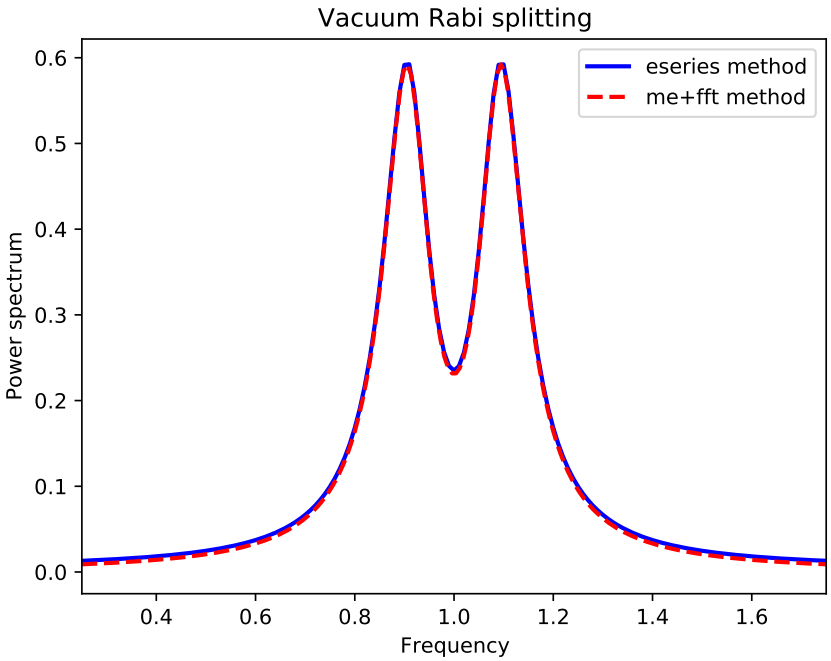
The following example demonstrates how these two functions can be used to obtain the emission power spectrum.

```
import numpy as np
from matplotlib import pyplot
import qutip

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  = qutip.tensor(qutip.destroy(N), qutip.qeye(2))
sm = qutip.tensor(qutip.qeye(N), qutip.destroy(2))
H = wc*a.dag()*a + wa*sm.dag()*sm + g*(a.dag()*sm + a*sm.dag())

# collapse operators
```

(continues on next page)

(continued from previous page)

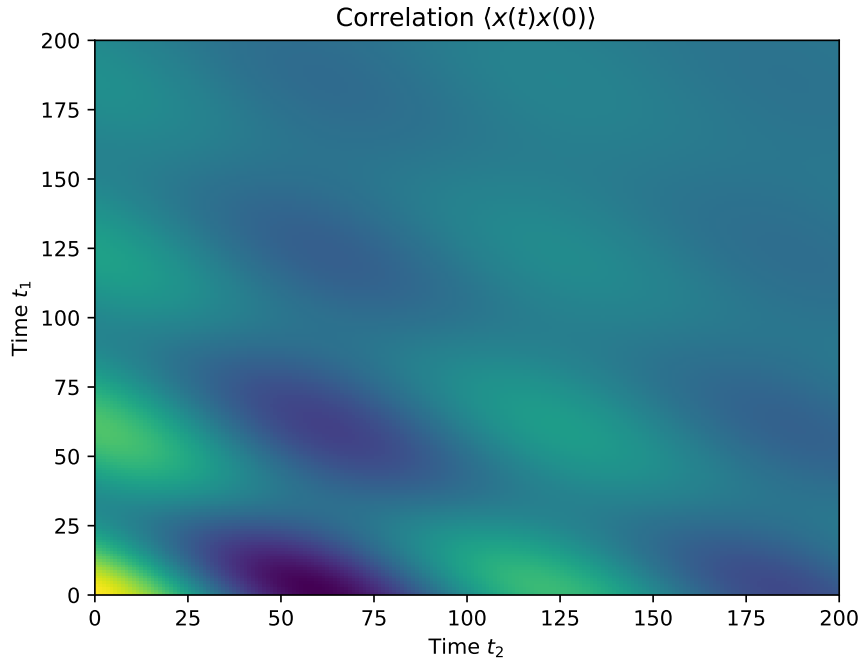
```
n_th = 0.25
c_ops = [
    np.sqrt(kappa * (1 + n_th)) * a,
    np.sqrt(kappa * n_th) * a.dag(),
    np.sqrt(gamma) * sm,
]

# 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 = qutip.correlation_2op_1t(H, None, tlist, c_ops, a.dag(), a)
wlist1, spec1 = qutip.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 = qutip.spectrum(H, wlist2, c_ops, a.dag(), a)

# plot the spectra
fig, ax = pyplot.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 steady state. In QuTiP, we can evaluate 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).

```
import numpy as np
import matplotlib.pyplot as plt
import qutip

times = np.linspace(0, 10.0, 200)
a = qutip.destroy(10)
x = a.dag() + a
H = a.dag() * a
alpha = 2.5
rho0 = qutip.coherent_dm(10, alpha)
corr = qutip.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()
```

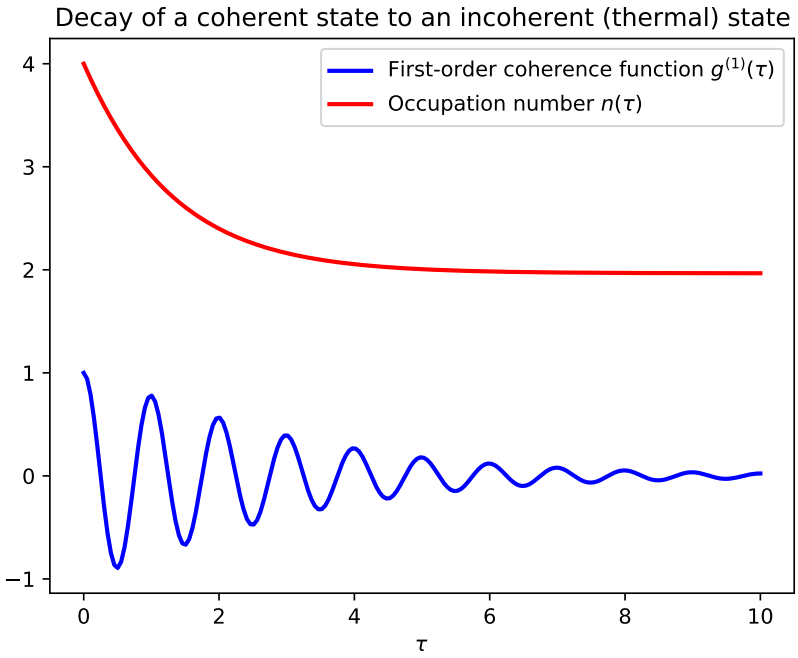


However, in some cases we might be interested in the correlation functions on the form $\langle A(t_1 + t_2)B(t_1)\rangle$, but only as a function of time coordinate t_2. In this case we can also use the *qutip.correlation.correlation_2op_2t* function, if we pass the density matrix at time t_1 as second 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.

```
import numpy as np
import matplotlib.pyplot as plt
import qutip

N = 15
taus = np.linspace(0,10.0,200)
a = qutip.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 = qutip.coherent_dm(N, 2.0)

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

# calculate the correlation function G1 and normalize with n to obtain g1
G1 = qutip.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_3op_1t*, which computes a correlation function on the form $\langle A(0)B(\tau)C(0)\rangle$ (three operators, one delay-time vector). We first have to combine the central two operators into one single one as they are evaluated at the same time, e.g. here we do $a^\dagger(\tau)a(\tau) = (a^\dagger a)(\tau)$.

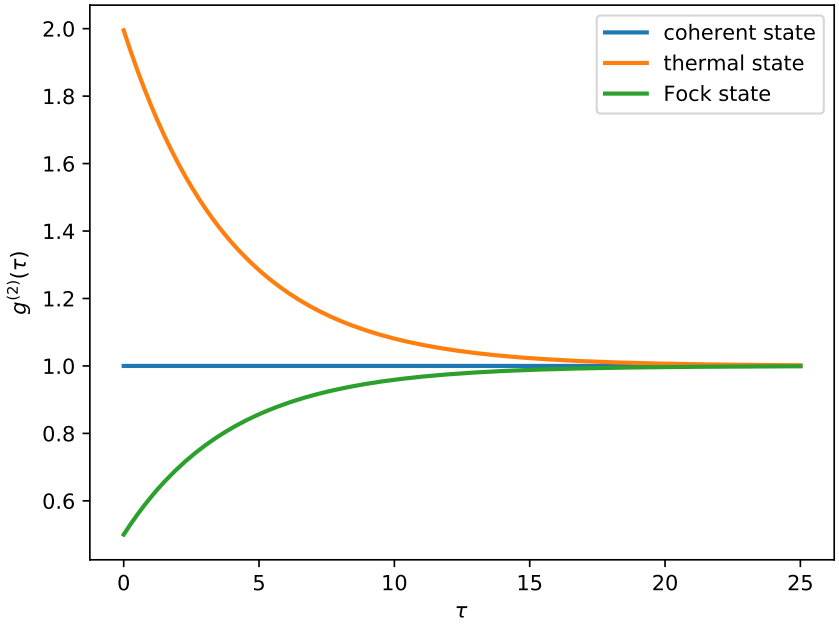
The following code calculates and plots $g^{(2)}(\tau)$ as a function of τ for a coherent, thermal and Fock state.

```
import numpy as np
import matplotlib.pyplot as plt
import qutip

N = 25
taus = np.linspace(0, 25.0, 200)
a = qutip.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 = [
```

(continues on next page)

(continued from previous page)

```
    {'state': qutip.coherent_dm(N, np.sqrt(2)), 'label': "coherent state"},
    {'state': qutip.thermal_dm(N, 2), 'label': "thermal state"},
    {'state': qutip.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 = qutip.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 = qutip.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.

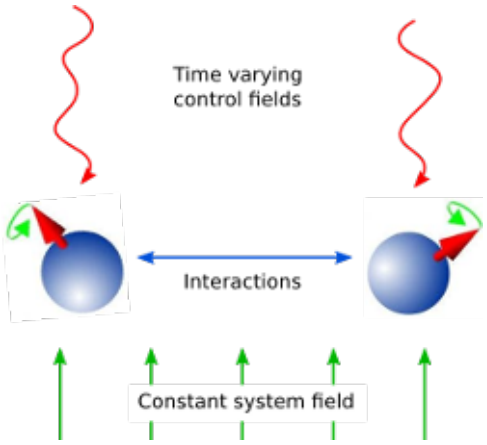


Fig. 3: Schematic showing the principle of quantum control.

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 [NKanej]. 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.

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

where d is the system dimension. In this figure of merit the absolute value is taken to ignore any differences in global phase, and $0 \le f \le 1$. 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*

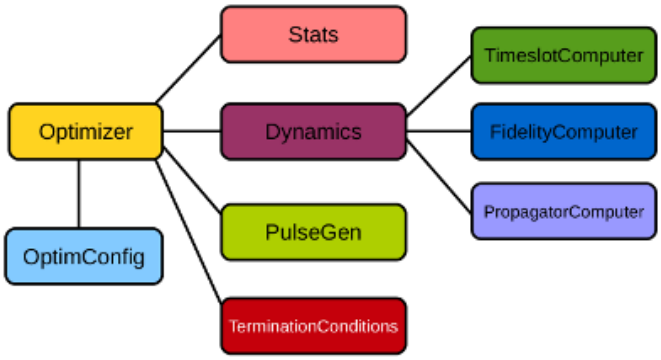


Fig. 4: Qtrl code object model.

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

3.10.1 Introduction

When studying the dynamics of a two-level system, it is often convenient 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.Bloch* class, uses Matplotlib to render the Bloch sphere, where as *qutip.bloch3d.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:

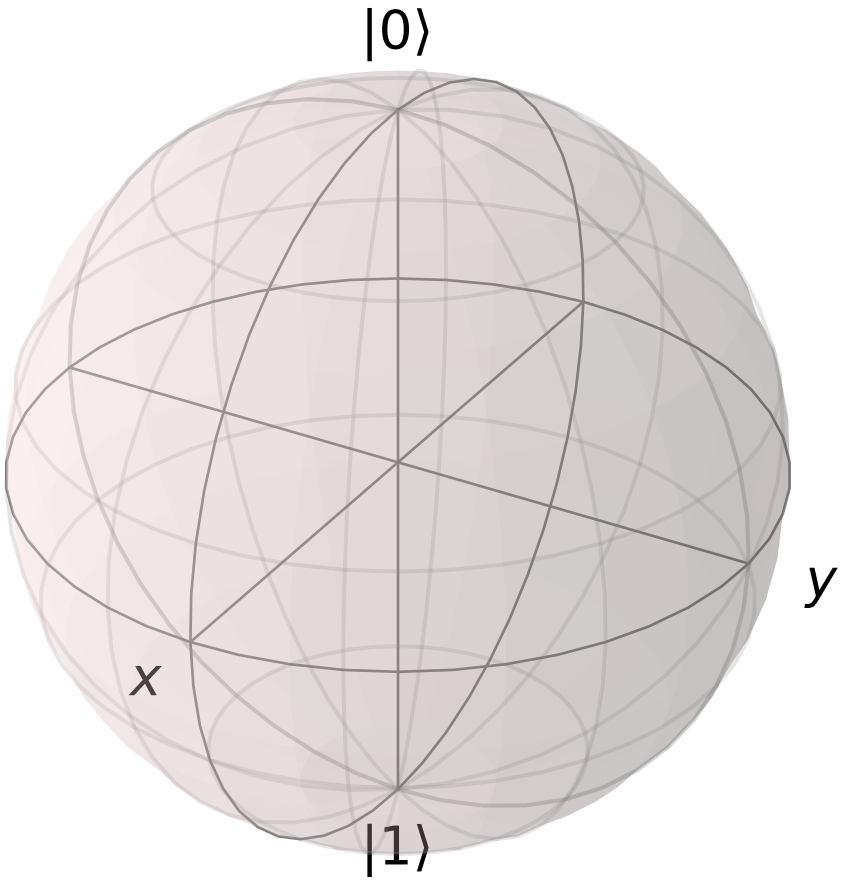
```
b = qutip.Bloch()
```

which will load an instance of the *qutip.bloch.Bloch* class, or using

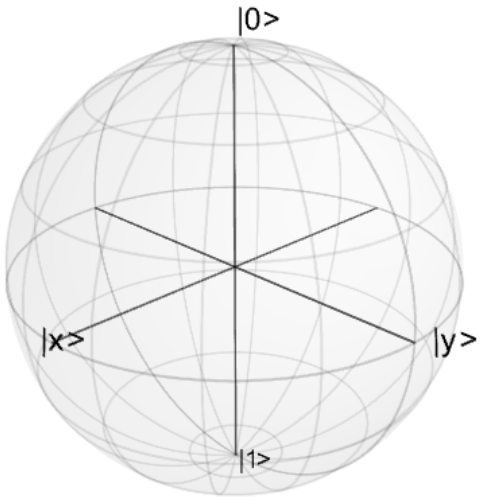
```
>>> b3d = qutip.Bloch3d()
```

that loads the *qutip.bloch3d.Bloch3d* version. Before getting into the details of these objects, we can simply plot the blank Bloch sphere associated with these instances via:

```
b.make_sphere()
```

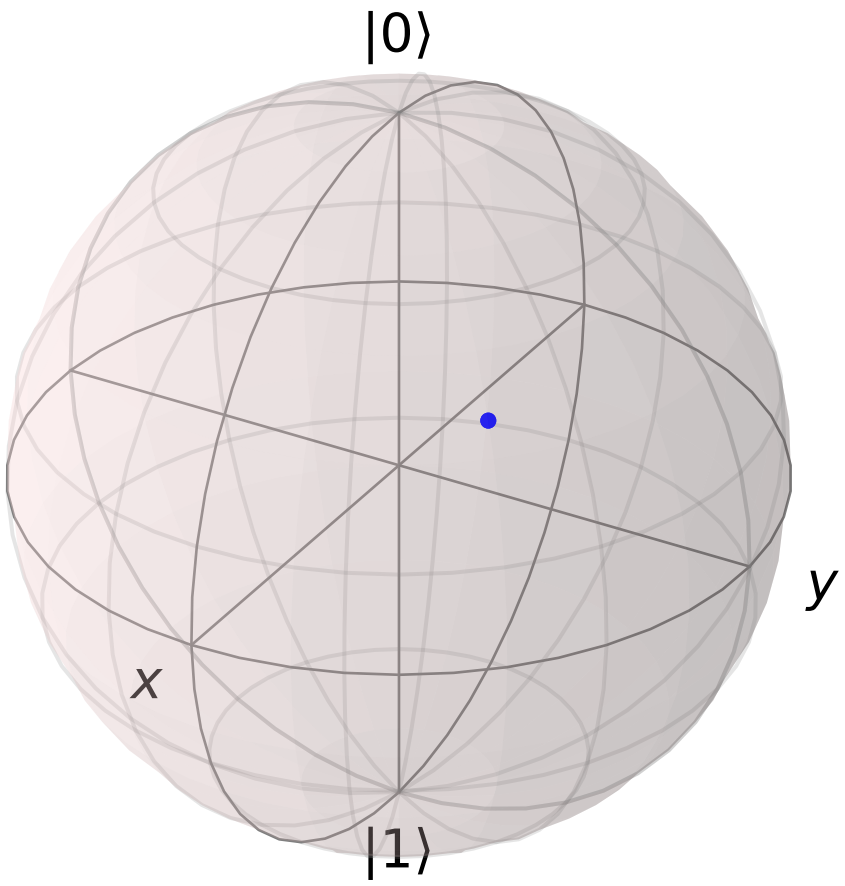


or



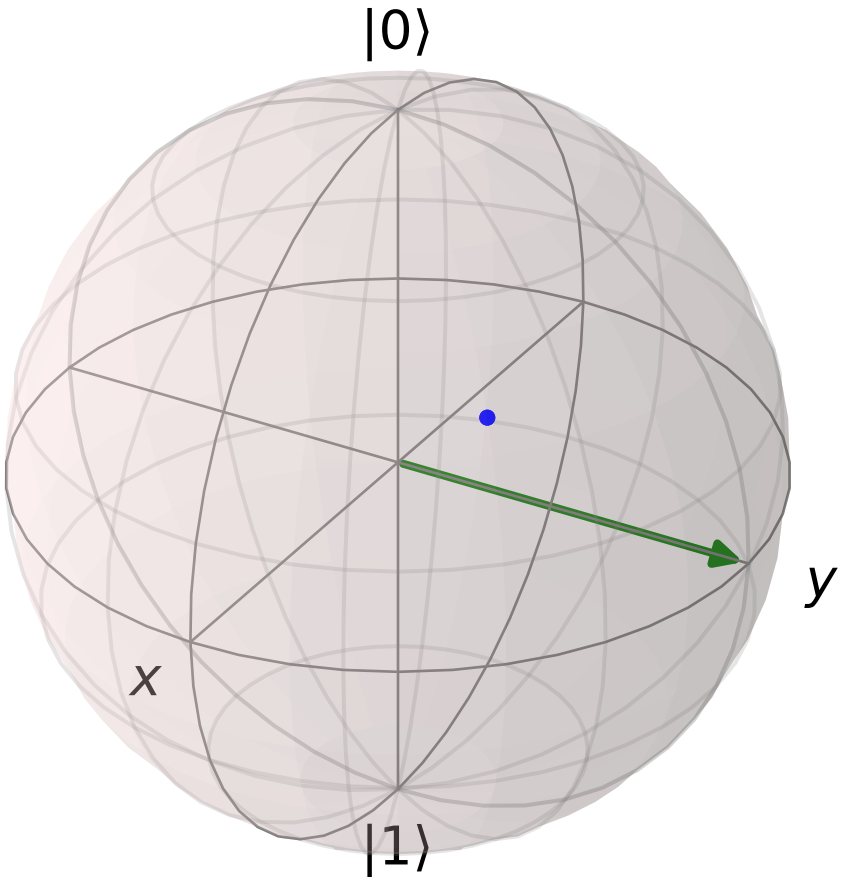
In addition to the `show` command, see the API documentation for *Bloch* for a full list of other available functions. As an example, we can add a single data point:

```
pnt = [1/np.sqrt(3), 1/np.sqrt(3), 1/np.sqrt(3)]
b.add_points(pnt)
b.render()
```



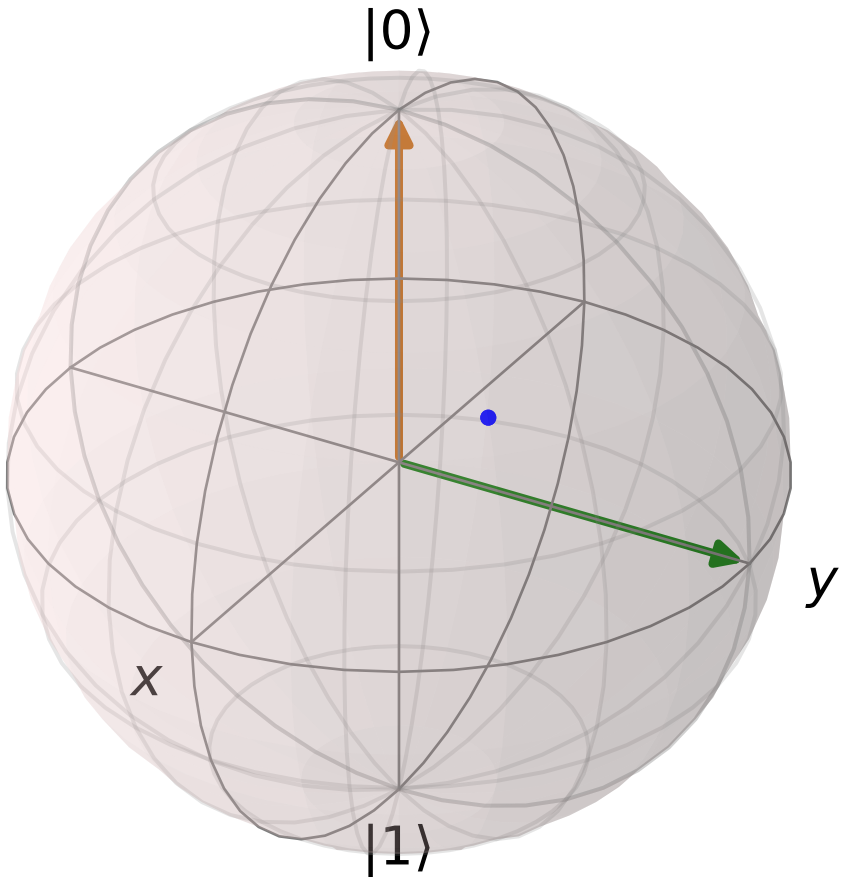
and then a single vector:

```
b.fig.clf()
vec = [0, 1, 0]
b.add_vectors(vec)
b.render()
```



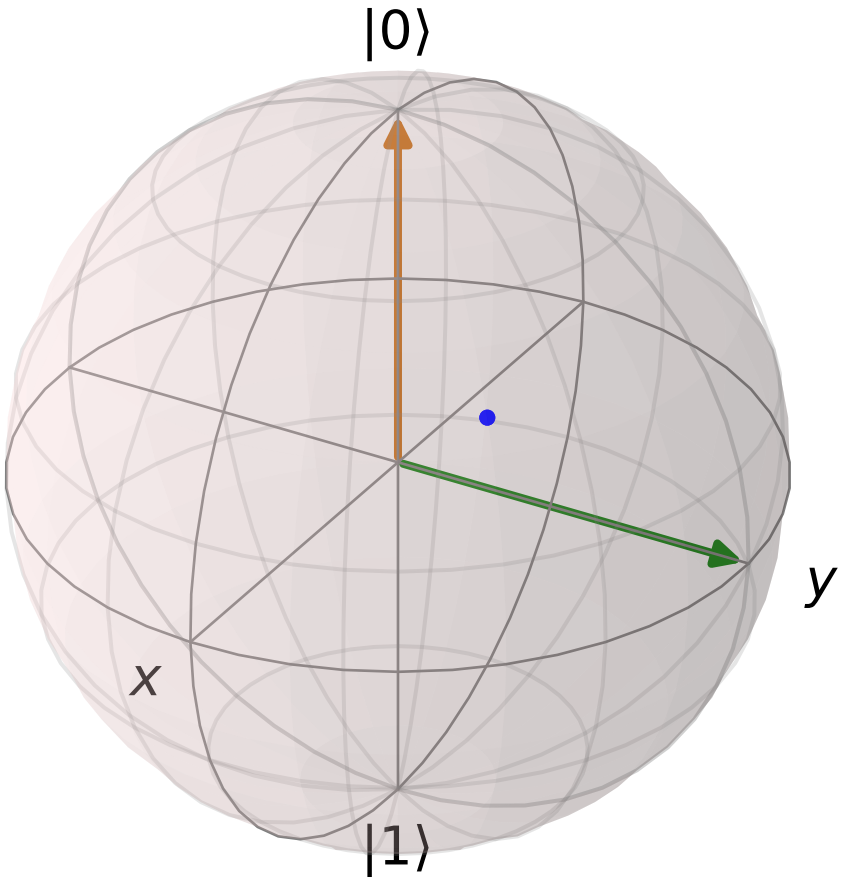
and then add another vector corresponding to the $|\mathrm{up}\rangle$ state:

```
up = qutip.basis(2, 0)
b.add_states(up)
b.render()
```

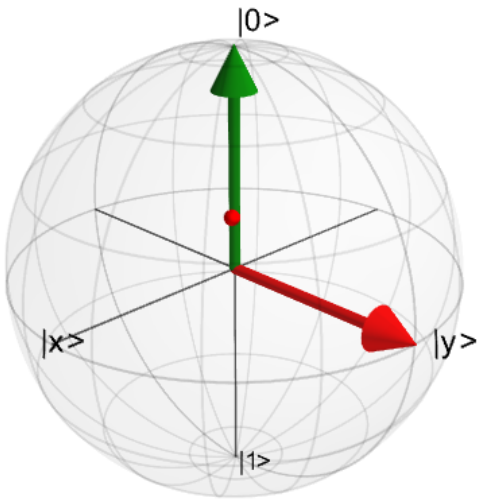


Notice that when we add more than a single vector (or data point), a different color will automatically be applied to the later data set (mod 4). In total, the code for constructing our Bloch sphere with one vector, one state, and a single data point is:

```
b = qutip.Bloch()

pnt = [1./np.sqrt(3), 1./np.sqrt(3), 1./np.sqrt(3)]
b.add_points(pnt)
vec = [0, 1, 0]
b.add_vectors(vec)
up = qutip.basis(2, 0)
b.add_states(up)
b.render()
```

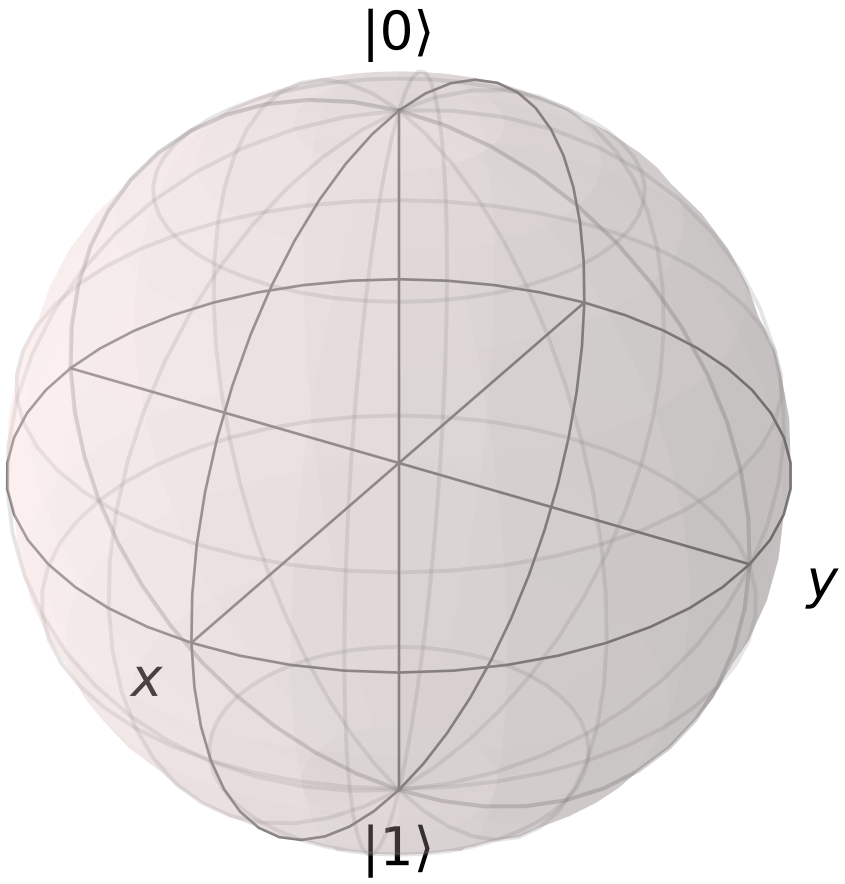


where we have removed the extra `show()` commands. Replacing `b=Bloch()` with `b=Bloch3d()` in the above code generates the following 3D Bloch sphere.



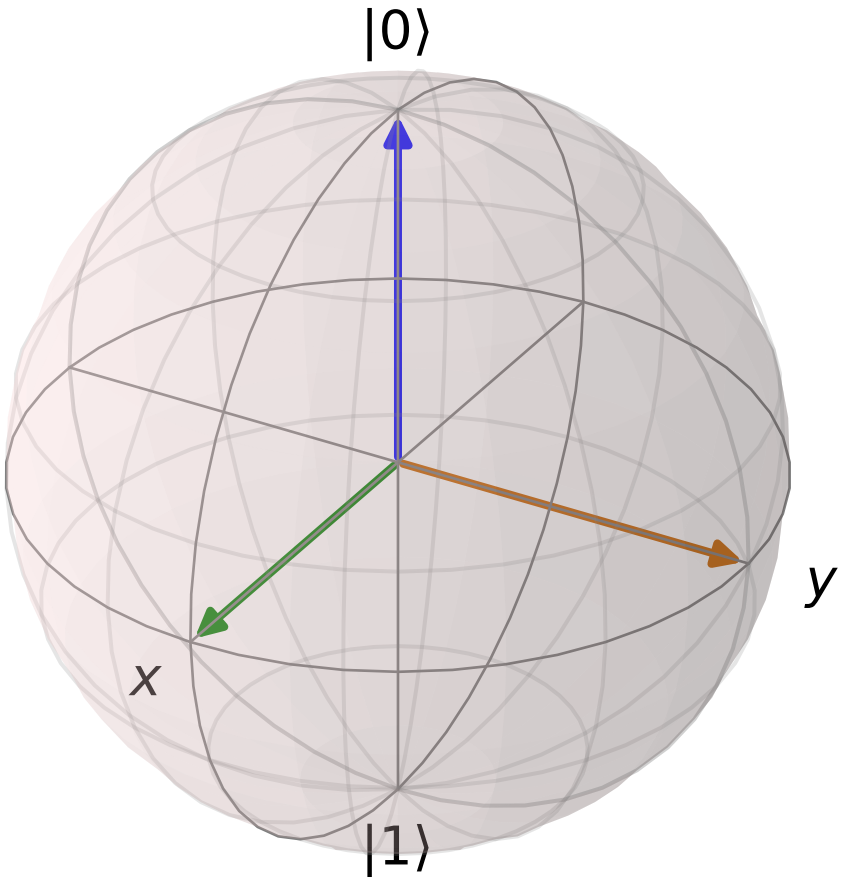
We can also plot multiple points, vectors, and states at the same time by passing list or arrays instead of individual elements. Before giving an example, we can use the *clear()* command to remove the current data from our Bloch sphere instead of creating a new instance:

```
b.clear()
b.render()
```



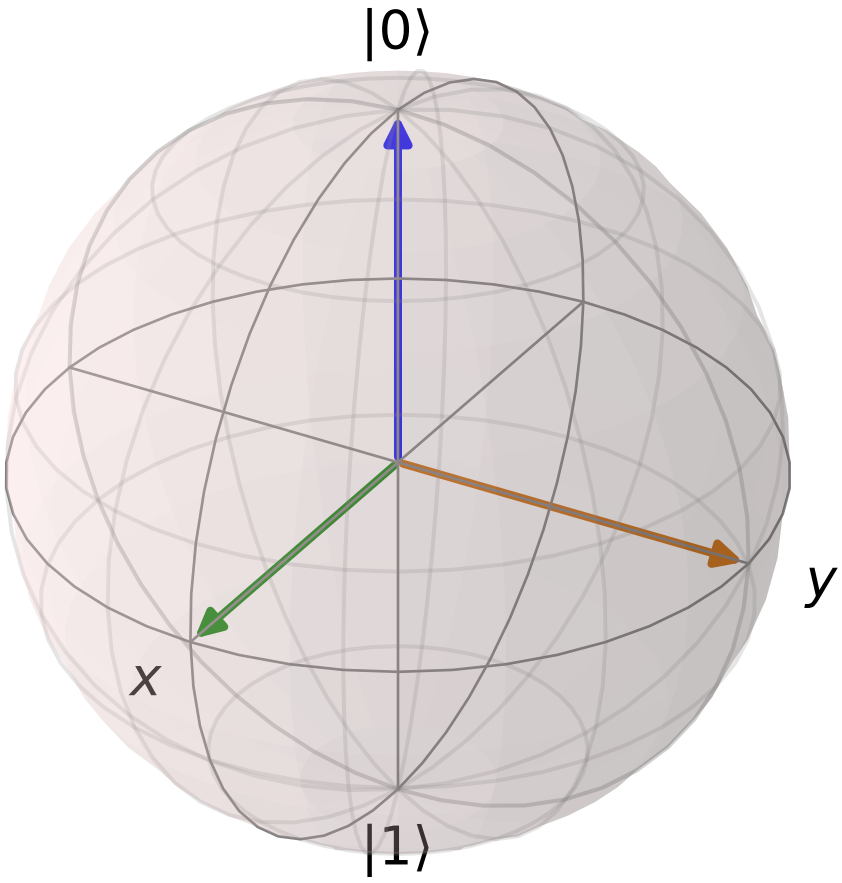
Now on the same Bloch sphere, we can plot the three states associated with the x, y, and z directions:

```
x = (qutip.basis(2, 0) + (1+0j)*qutip.basis(2, 1)).unit()
y = (qutip.basis(2, 0) + (0+1j)*qutip.basis(2, 1)).unit()
z = (qutip.basis(2, 0) + (0+0j)*qutip.basis(2, 1)).unit()

b.add_states([x, y, z])
b.render()
```



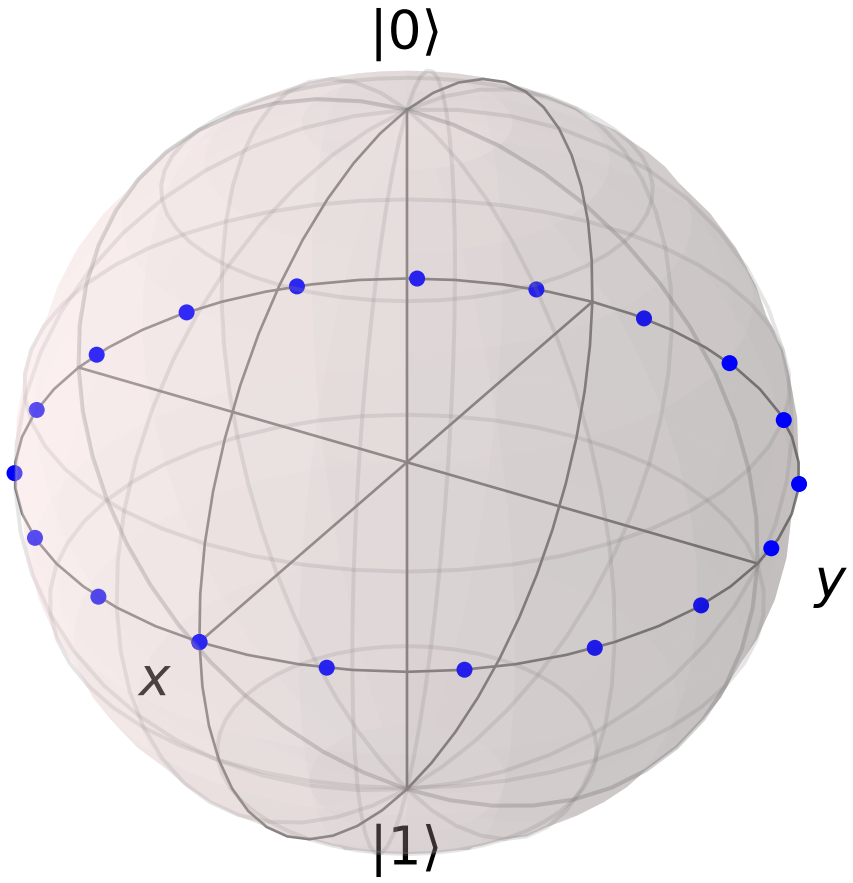
a similar method works for adding vectors:

```
b.clear()
vec = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
b.add_vectors(vec)
b.render()
```



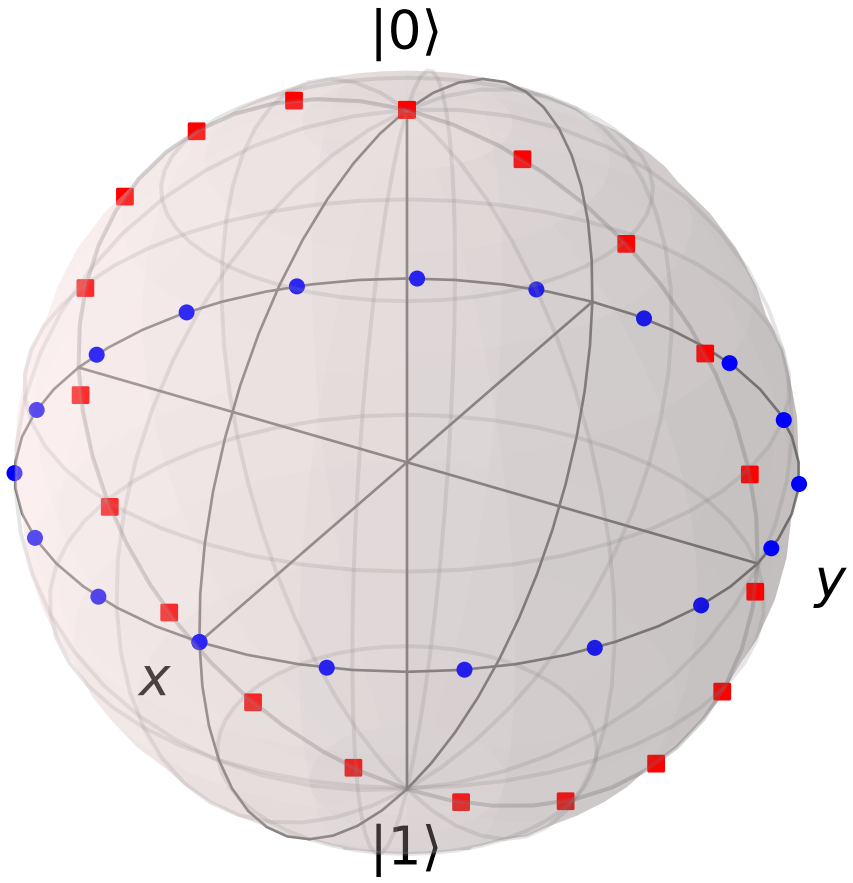
Adding multiple points to the Bloch sphere works slightly differently than adding multiple states or vectors. For example, lets add a set of 20 points around the equator (after calling *clear()*):

```
b.clear()

th = np.linspace(0, 2*np.pi, 20)
xp = np.cos(th)
yp = np.sin(th)
zp = np.zeros(20)

pnts = [xp, yp, zp]
b.add_points(pnts)
b.render()
```



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:

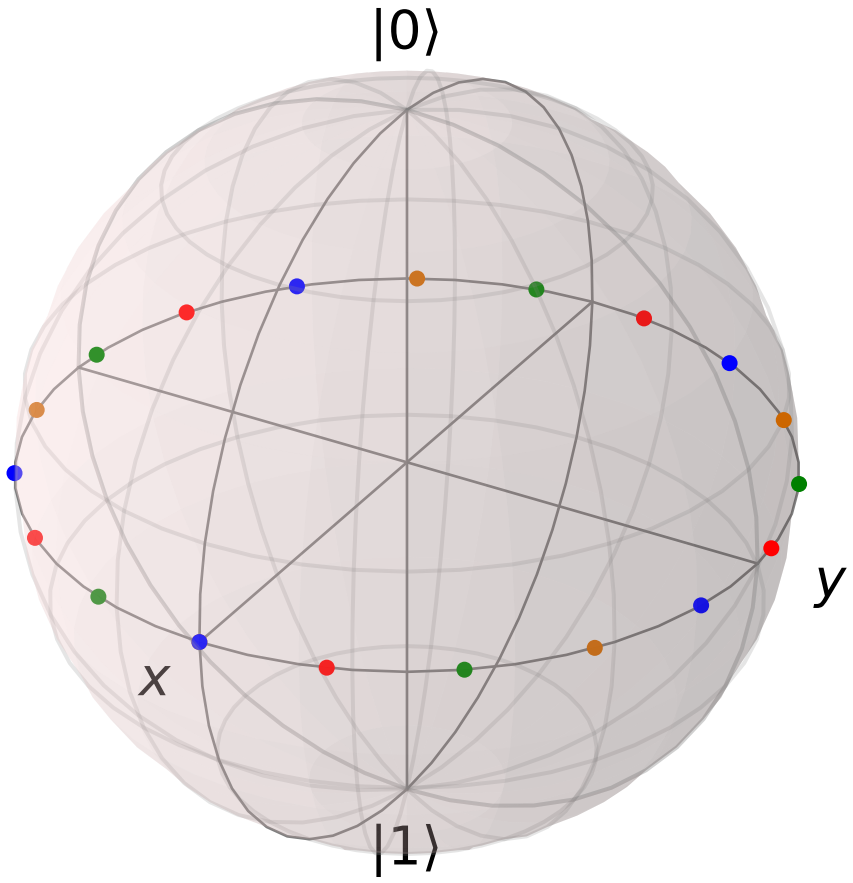
```
xz = np.zeros(20)
yz = np.sin(th)
zz = np.cos(th)
b.add_points([xz, yz, zz])
b.render()
```



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.

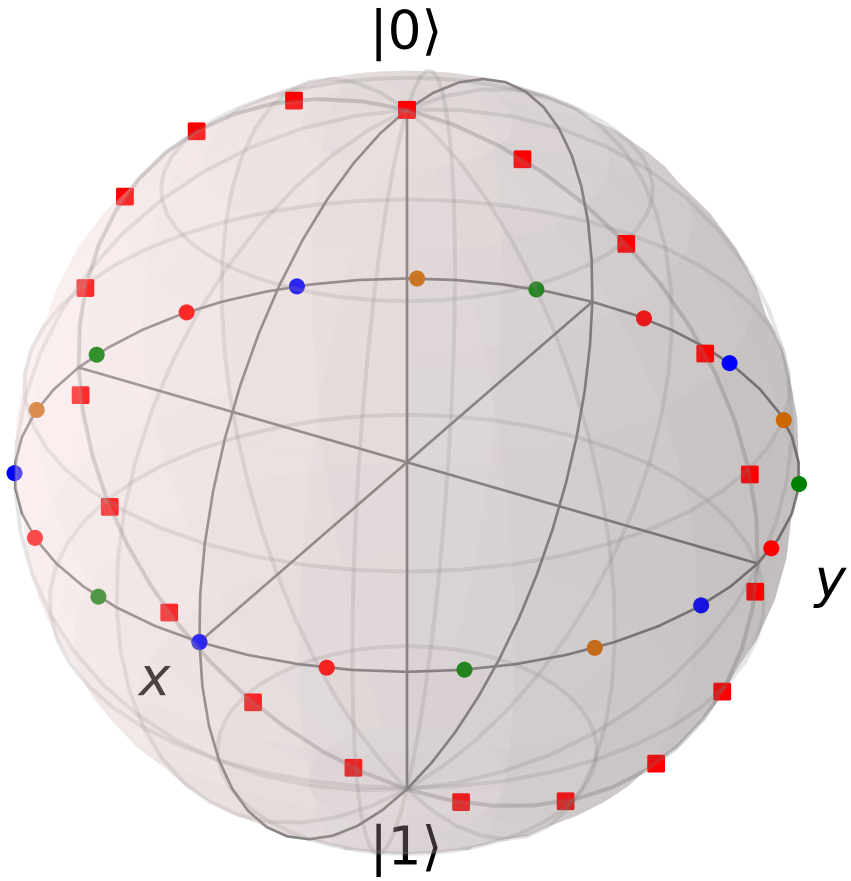
What if we want to vary the color of our points. We can tell the *`qutip.bloch.Bloch`* class to vary the color of each point according to the colors listed in the `b.point_color` list (see *Configuring the Bloch sphere* below). Again after `clear()`:

```
b.clear()

xp = np.cos(th)
yp = np.sin(th)
zp = np.zeros(20)
pnts = [xp, yp, zp]
b.add_points(pnts, 'm')  # <-- add a 'm' string to signify 'multi' colored points
b.render()
```

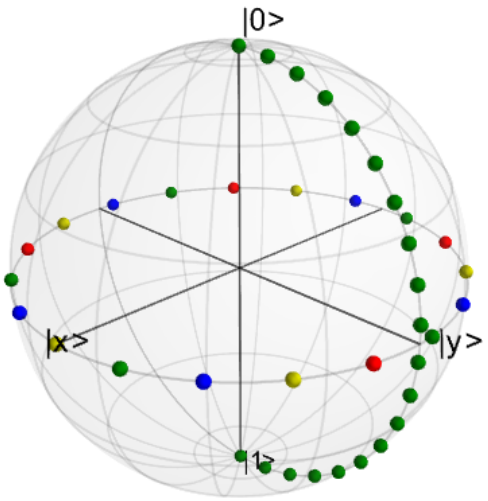


Now, the data points cycle through a variety of predefined colors. Now lets add another set of points, but this time we want the set to be a single color, representing say a qubit going from the $|\mathrm{up}\rangle$ state to the $|\mathrm{down}\rangle$ state in the y-z plane:

```
xz = np.zeros(20)
yz = np.sin(th)
zz = np.cos(th)

b.add_points([xz, yz, zz])  # no 'm'
b.render()
```



Again, the same plot can be generated using the *qutip.bloch3d.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:

```
>>> b = qutip.Bloch()

>>> 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
```

(continues on next page)

(continued from previous page)

```
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]
```

3.10.4 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.

```
import numpy as np
import qutip

def qubit_integrate(w, theta, gamma1, gamma2, psi0, tlist):
    # operators and the hamiltonian
    sx = qutip.sigmax()
    sy = qutip.sigmay()
    sz = qutip.sigmaz()
    sm = qutip.sigmam()
    H = w * (np.cos(theta) * sz + np.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(np.sqrt(rate) * sm)
    rate = gamma1 * n_th
    if rate > 0.0: c_op_list.append(np.sqrt(rate) * sm.dag())
    rate = gamma2
    if rate > 0.0: c_op_list.append(np.sqrt(rate) * sz)
    # evolve and calculate expectation values
    output = qutip.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 * np.pi  # qubit angular frequency
theta = 0.2 * np.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*qutip.basis(2, 0) + (1-a)*qutip.basis(2, 1))/np.sqrt(a**2 + (1-a)**2)
tlist = np.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:

```
import numpy as np
b = qutip.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 -i temp/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 matplotlib import pyplot, animation
from mpl_toolkits.mplot3d import Axes3D

fig = pyplot.figure()
ax = Axes3D(fig, azim=-40, elev=30)
sphere = qutip.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=False, repeat=False)
ani.save('bloch_sphere.mp4', fps=20)
```

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, $\mathrm{Tr}[\rho a^\dagger a] = 2$. Given this information we cannot say whether the oscillator is in a Fock state, a thermal state, a coherent state, etc. By visualizing the photon distribution in the Fock state basis important clues about the underlying state can be obtained.

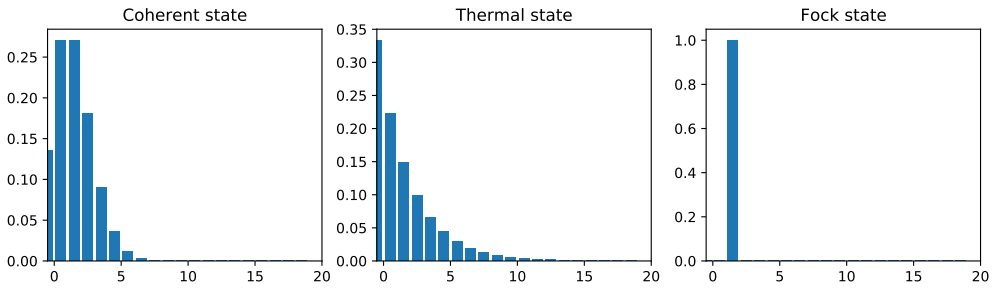
One convenient way to visualize a probability distribution is to use histograms. Consider the following histogram visualization of the number-basis probability distribution, which can be obtained from the diagonal of the density matrix, for a few possible oscillator states with on average occupation of two photons.

First we generate the density matrices for the coherent, thermal and fock states.

```
N = 20

rho_coherent = coherent_dm(N, np.sqrt(2))

rho_thermal = thermal_dm(N, 2)

rho_fock = fock_dm(N, 2)
```

Next, we plot histograms of the diagonals of the density matrices:

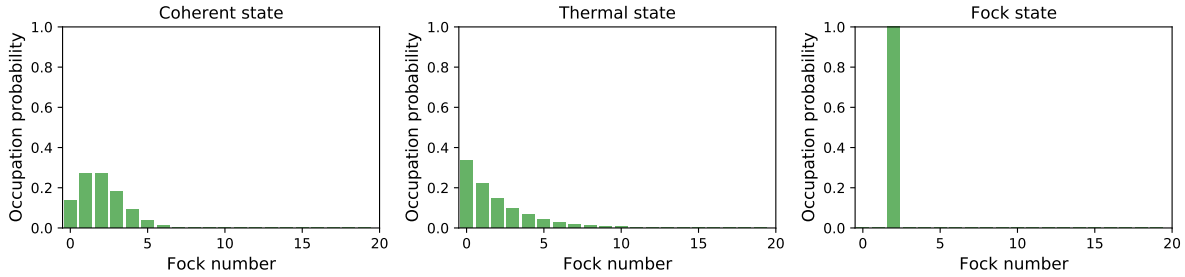
```
fig, axes = plt.subplots(1, 3, figsize=(12,3))

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

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

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

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

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

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

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

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

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

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 convenience function for doing this, see *qutip.visualization.plot_fock_distribution*, and the following example:

```
fig, axes = plt.subplots(1, 3, figsize=(12,3))

plot_fock_distribution(rho_coherent, fig=fig, ax=axes[0], title="Coherent state");

plot_fock_distribution(rho_thermal, fig=fig, ax=axes[1], title="Thermal state");

plot_fock_distribution(rho_fock, fig=fig, ax=axes[2], title="Fock state");

fig.tight_layout()

plt.show()
```



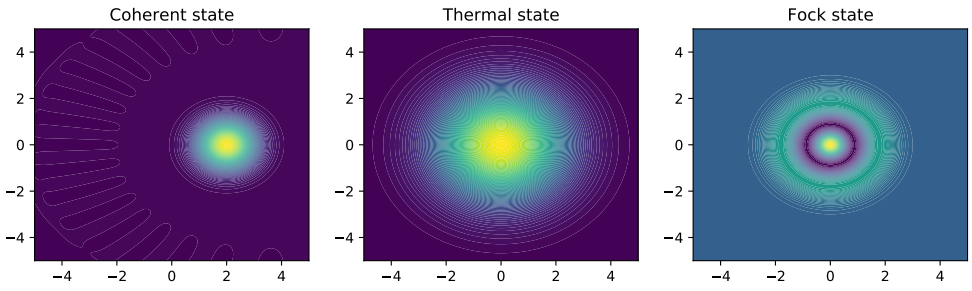
3.11.2 Quasi-probability distributions

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.

```
xvec = np.linspace(-5,5,200)

W_coherent = wigner(rho_coherent, xvec, xvec)

W_thermal = wigner(rho_thermal, xvec, xvec)

W_fock = wigner(rho_fock, xvec, xvec)

# plot the results

fig, axes = plt.subplots(1, 3, figsize=(12,3))

cont0 = axes[0].contourf(xvec, xvec, W_coherent, 100)

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

cont1 = axes[1].contourf(xvec, xvec, W_thermal, 100)

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

cont0 = axes[2].contourf(xvec, xvec, W_fock, 100)

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

plt.show()
```



Custom Color Maps

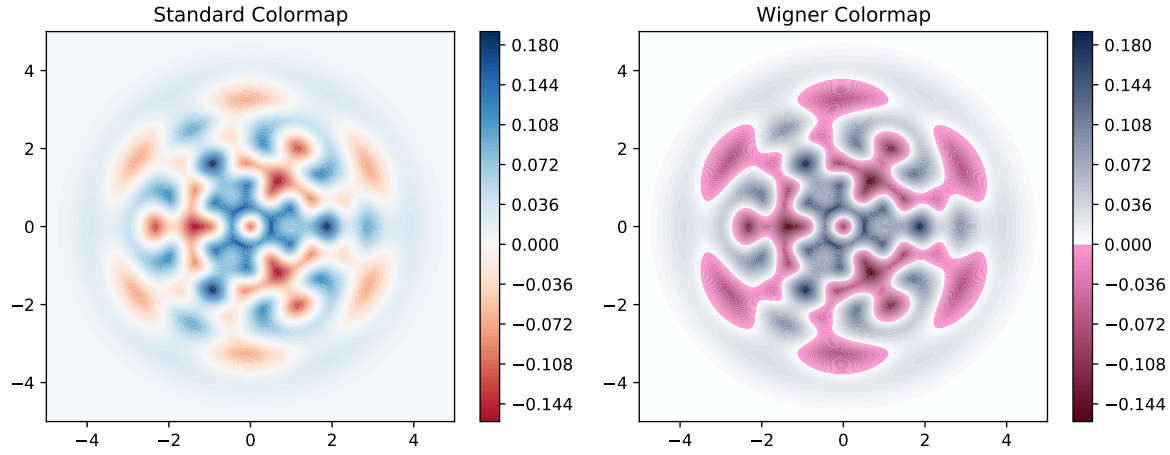
The main objective when plotting a Wigner function is to demonstrate that the underlying state is nonclassical, as indicated by negative values in the Wigner function. Therefore, making these negative values stand out in a figure is helpful for both analysis and publication purposes. Unfortunately, all 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.matplotlib_utilities.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:

```
import matplotlib as mpl

from matplotlib import cm

psi = (basis(10, 0) + basis(10, 3) + basis(10, 9)).unit()
```

(continues on next page)

(continued from previous page)

```
xvec = np.linspace(-5, 5, 500)

W = wigner(psi, xvec, xvec)

wmap = wigner_cmap(W)  # Generate Wigner colormap

nrm = mpl.colors.Normalize(-W.max(), W.max())

fig, axes = plt.subplots(1, 2, figsize=(10, 4))

plt1 = axes[0].contourf(xvec, xvec, W, 100, cmap=cm.RdBu, norm=nrm)

axes[0].set_title("Standard Colormap");

cb1 = fig.colorbar(plt1, ax=axes[0])

plt2 = axes[1].contourf(xvec, xvec, W, 100, cmap=wmap)  # Apply Wigner colormap

axes[1].set_title("Wigner Colormap");

cb2 = fig.colorbar(plt2, ax=axes[1])

fig.tight_layout()

plt.show()
```



Husimi Q-function

The Husimi Q function is, like the Wigner function, a quasiprobability distribution for harmonic modes. It is defined as

$$Q(\alpha) = \frac{1}{\pi} \langle \alpha | \rho | \alpha \rangle$$

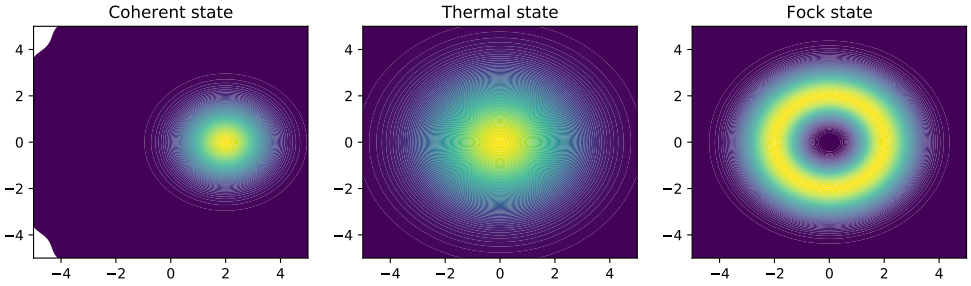
where $|\alpha\rangle$ is a coherent state and $\alpha = x + iy$. In QuTiP, the Husimi Q function can be computed given a state ket or density matrix using the function *qfunc*, as demonstrated below.

```
Q_coherent = qfunc(rho_coherent, xvec, xvec)
Q_thermal = qfunc(rho_thermal, xvec, xvec)
Q_fock = qfunc(rho_fock, xvec, xvec)
fig, axes = plt.subplots(1, 3, figsize=(12,3))
cont0 = axes[0].contourf(xvec, xvec, Q_coherent, 100)
```

(continues on next page)

(continued from previous page)

```
lbl0 = axes[0].set_title("Coherent state")
cont1 = axes[1].contourf(xvec, xvec, Q_thermal, 100)
lbl1 = axes[1].set_title("Thermal state")
cont0 = axes[2].contourf(xvec, xvec, Q_fock, 100)
lbl2 = axes[2].set_title("Fock state")
plt.show()
```



If you need to calculate the Q function for many states with the same phase-space coordinates, it is more efficient to use the *QFunc* class. This stores various intermediary results to achieve an order-of-magnitude improvement compared to calling *qfunc* in a loop.

```
xs = np.linspace(-1, 1, 101)
qfunc_calculator = qutip.QFunc(xs, xs)
q_state1 = qfunc_calculator(qutip.rand_dm(5))
q_state2 = qfunc_calculator(qutip.rand_ket(100))
```

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* as first argument, and optional arguments to, for example, set the axis labels and figure title (see the function's documentation for details).

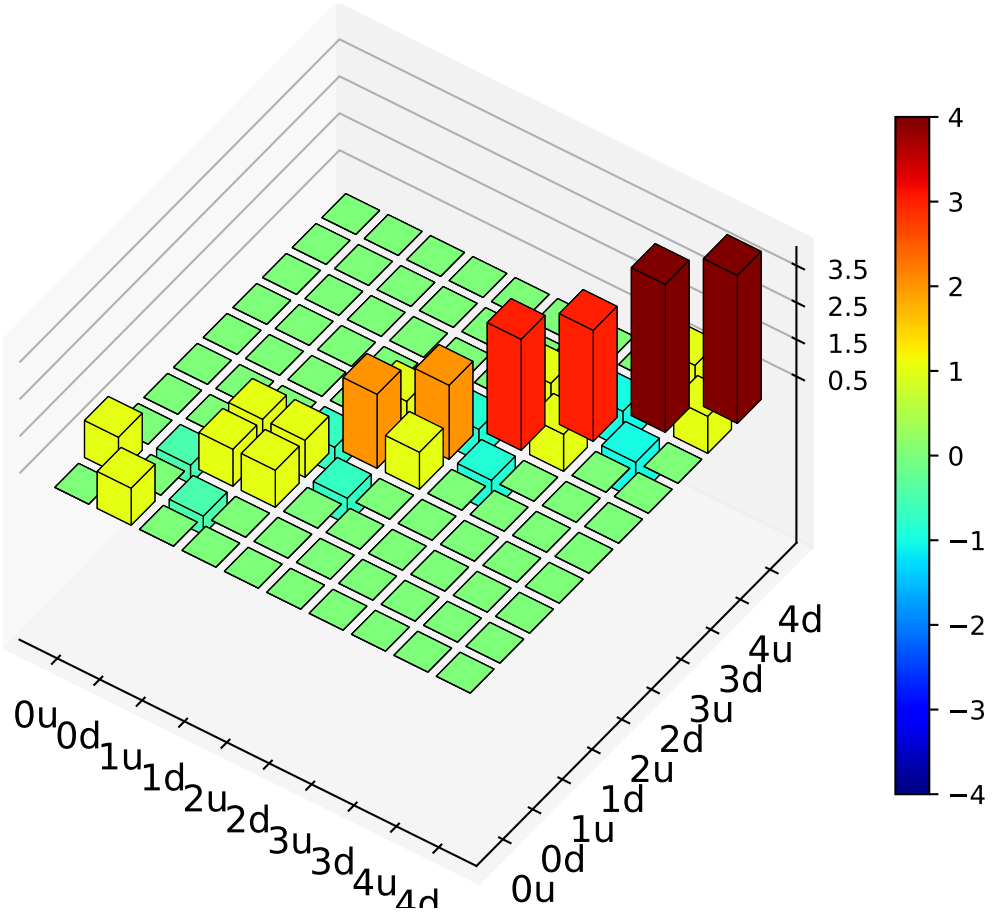
For example, to illustrate the use of *qutip.visualization.matrix_histogram*, let's visualize of the Jaynes-Cummings Hamiltonian:

```
N = 5

a = tensor(destroy(N), qeye(2))

b = tensor(qeye(N), destroy(2))

sx = tensor(qeye(N), sigmax())

H = a.dag() * a + sx - 0.5 * (a * b.dag() + a.dag() * b)

# visualize H

lbls_list = [[str(d) for d in range(N)], ["u", "d"]]

xlabels = []
```

(continues on next page)

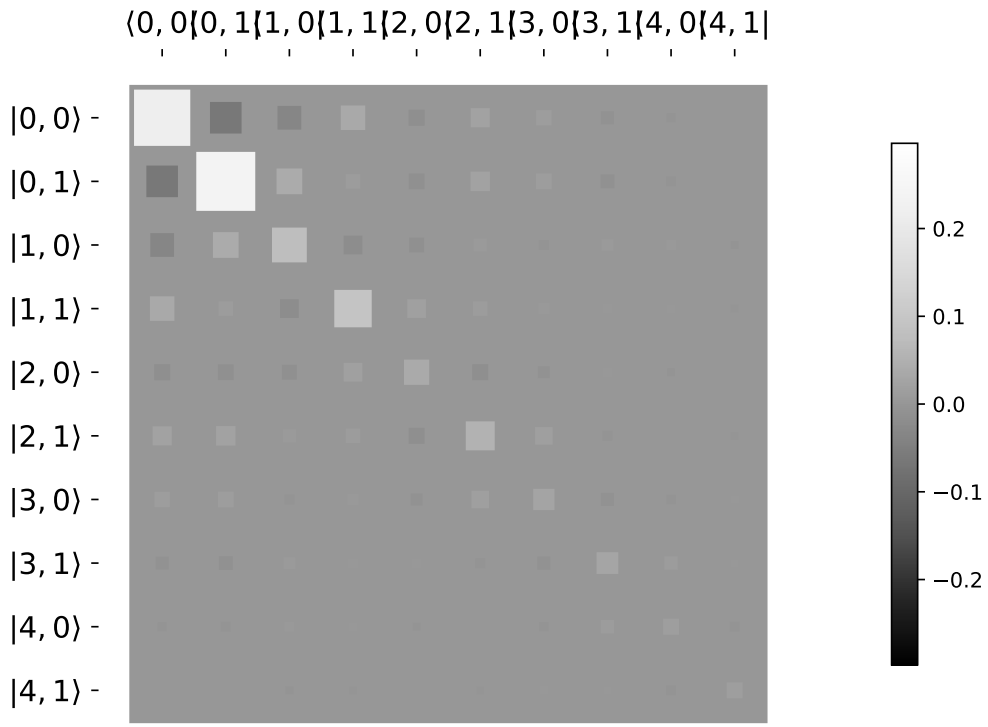
(continued from previous page)

```
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))]))

fig, ax = matrix_histogram(H, xlabels, xlabels, limits=[-4,4])

ax.view_init(azim=-55, elev=45)

plt.show()
```



Similarly, we can use the function *qutip.visualization.hinton*, which is used below to visualize the corresponding steadystate density matrix:

```
rho_ss = steadystate(H, [np.sqrt(0.1) * a, np.sqrt(0.4) * b.dag()])

hinton(rho_ss)

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 $\epsilon(\rho_{\rm in}) = \rho_{\rm out}$, which can be written

$$\epsilon(\rho_{\rm in}) = \rho_{\rm out} = \sum_{i}^{N^2} A_i \rho_{\rm in} A_i^\dagger, \tag{3.40}$$

where N is the number of states of the system (that is, ρ is represented by an $[N \times N]$ matrix). Given an orthogonal operator basis of our choice $\{B_i\}_i^{N^2}$, which satisfies $\mathrm{Tr}[B_i^\dagger B_j] = N\delta_{ij}$, we can write the map as

$$\epsilon(\rho_{\rm in}) = \rho_{\rm out} = \sum_{mn} \chi_{mn} B_m \rho_{\rm in} B_n^\dagger. \tag{3.41}$$

where $\chi_{mn} = \sum_{ij} b_{im} b_{jn}^*$ and $A_i = \sum_m b_{im} B_m$. Here, matrix χ is the transformation matrix we are after, since it describes how much $B_m \rho_{\rm in} B_n^\dagger$ contributes to $\rho_{\rm out}$.

In a numerical simulation of a quantum process we usually do not have access to the quantum map in the form Eq. (3.40). Instead, what we usually can do is to calculate the propagator U for the density matrix in superoperator

form, using for example the QuTiP function *qutip.propagator.propagator*. We can then write

$$\epsilon(\tilde{\rho}_{\rm in}) = U\tilde{\rho}_{\rm in} = \tilde{\rho}_{\rm out}$$

where $\tilde{\rho}$ is the vector representation of the density matrix ρ. If we write Eq. (3.41) in superoperator form as well we obtain

$$\tilde{\rho}_{\rm out} = \sum_{mn} \chi_{mn} \tilde{B}_m \tilde{B}_n^\dagger \tilde{\rho}_{\rm in} = U\tilde{\rho}_{\rm in}.$$

so we can identify

$$U = \sum_{mn} \chi_{mn} \tilde{B}_m \tilde{B}_n^\dagger.$$

Now this is a linear equation systems for the $N^2 \times N^2$ elements in χ. We can solve it by writing χ and the superoperator propagator as $[N^4]$ vectors, and likewise write the superoperator product $\tilde{B}_m \tilde{B}_n^\dagger$ as a $[N^4 \times N^4]$ matrix M:

$$U_I = \sum_J^{N^4} M_{IJ} \chi_J$$

with the solution

$$\chi = M^{-1} U.$$

Note that to obtain χ with this method we have to construct a matrix M with a size that is the square of the size of the superoperator for the system. Obviously, this scales very badly with increasing system size, but this method can still be a very useful for small systems (such as system comprised of a small number of coupled qubits).

Implementation in QuTiP

In QuTiP, the procedure described above is implemented in the function *qutip.tomography.qpt*, which returns the χ matrix given a density matrix propagator. To illustrate how to use this function, let's consider the i-SWAP gate for two qubits. In QuTiP the function *qutip.qip.operations.iswap* generates the unitary transformation for the state kets:

```
from qutip.qip.operations import iswap

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:

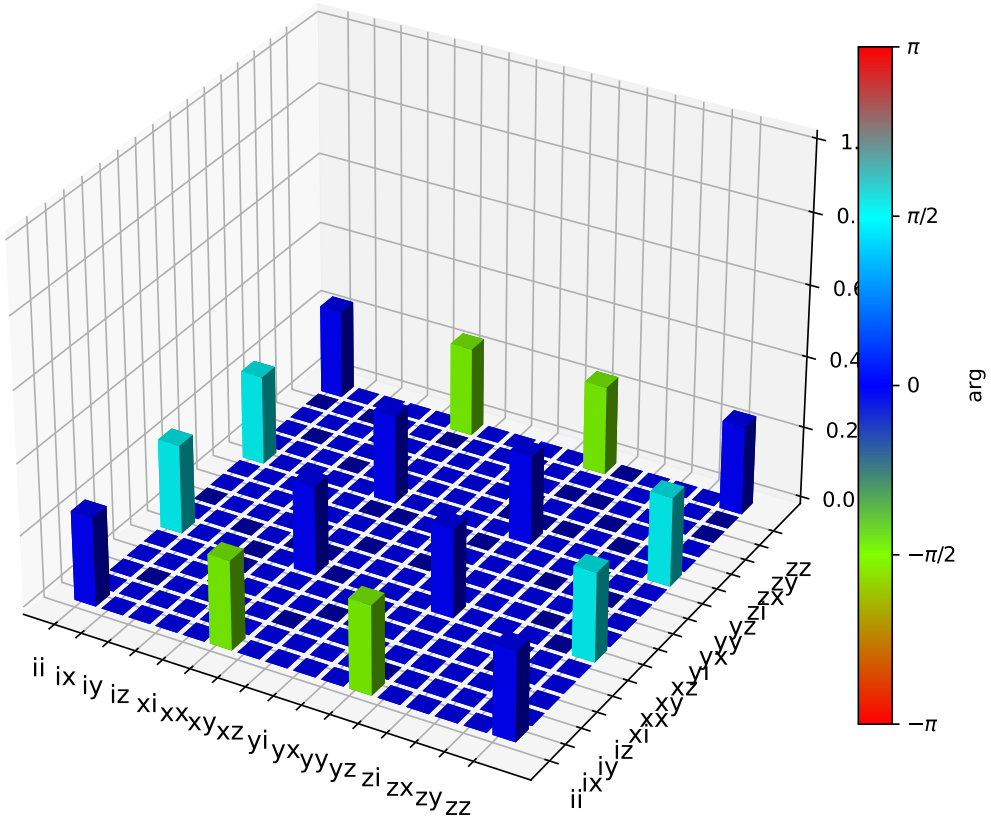
```
U_rho = spre(U_psi) * spost(U_psi.dag())
```

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.

```
op_basis = [[qeye(2), sigmax(), sigmay(), sigmaz()]] * 2
op_label = [["i", "x", "y", "z"]] * 2
```

We are now ready to compute χ using *qutip.tomography.qpt*, and to plot it using *qutip.tomography.qpt_plot_combined*.

```
chi = qpt(U_rho, op_basis)

fig = qpt_plot_combined(chi, op_label, r'$i$SWAP')

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:

```
>>> def func1(x): return x, x**2, x**3

>>> a, b, c = parfor(func1, range(10))

>>> print(a)
[0 1 2 3 4 5 6 7 8 9]

>>> print(b)
[ 0  1  4  9 16 25 36 49 64 81]
```

(continues on next page)

(continued from previous page)

```
>>> print(c)
[  0   1   8  27  64 125 216 343 512 729]
```

or

```
>>> result = parallel_map(func1, range(10))

>>> result_array = np.array(result)

>>> print(result_array[:, 0])  # == a
[0 1 2 3 4 5 6 7 8 9]

>>> print(result_array[:, 1])  # == b
[ 0  1  4  9 16 25 36 49 64 81]

>>> 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:

```
>>> result = parfor(func1, range(5))

>>> print(result)
[array([0, 1, 2, 3, 4]), array([ 0,  1,  4,  9, 16]), array([ 0,  1,  8, 27, 64])]

>>> result = parallel_map(func1, range(5))

>>> 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:

```
>>> def func2(x): return x, Qobj(x), 'a' * x

>>> a, b, c = parfor(func2, range(5))

>>> print(a)
[0 1 2 3 4]

>>> 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.]]]
```

(continues on next page)

(continued from previous page)

```
>>>print(c)
['' 'a' 'aa' 'aaa' 'aaaa']
```

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:

```
 >>> def sum_diff(x, y, z=0): return x + y, x - y, z

 >>> parfor(sum_diff, [1, 2, 3], [4, 5, 6], z=5.0)
 [array([5, 7, 9]), array([-3, -3, -3]), array([5., 5., 5.])]

 >>> parallel_map(sum_diff, [1, 2, 3], task_args=(np.array([4, 5, 6]),), task_
↪kwargs=dict(z=5.0))
 [(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.

```
>>> import time

>>> def func(x): time.sleep(1)

>>> result = parallel_map(func, range(50), progress_bar=True)

10.0%. Run time:   3.10s. Est. time left: 00:00:00:27
20.0%. Run time:   5.11s. Est. time left: 00:00:00:20
30.0%. Run time:   8.11s. Est. time left: 00:00:00:18
40.0%. Run time:  10.15s. Est. time left: 00:00:00:15
50.0%. Run time:  13.15s. Est. time left: 00:00:00:13
60.0%. Run time:  15.15s. Est. time left: 00:00:00:10
70.0%. Run time:  18.15s. Est. time left: 00:00:00:07
80.0%. Run time:  20.15s. Est. time left: 00:00:00:05
90.0%. Run time:  23.15s. Est. time left: 00:00:00:02
100.0%. Run time:  25.15s. Est. time left: 00:00:00:00
Total run time:  28.91s
```

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

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 IPython's 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

```
>>> a = destroy(10); H = a.dag() * a
>>> c_ops = [np.sqrt(0.5) * a, np.sqrt(0.25) * a.dag()]
>>> 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

```
>>> qsave(rho_ss, 'steadystate')
>>> !ls *.qu
density_matrix_vs_time.qu  steadystate.qu
```

and it can later be loaded again, and used in further calculations

```
>>> rho_ss_loaded = qload('steadystate')
Loaded Qobj object:
Quantum object: dims = [[10], [10]], shape = (10, 10), type = oper, isHerm = True
>>> a = destroy(10)
>>> np.testing.assert_almost_equal(expect(a.dag() * a, rho_ss_loaded), 0.
↪9902248289345061)
```

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*

```
>>> a = destroy(10); H = a.dag() * a ; c_ops = [np.sqrt(0.5) * a, np.sqrt(0.25) *␣
↪a.dag()]
>>> psi0 = rand_ket(10)
>>> times = np.linspace(0, 10, 10)
>>> dm_list = mesolve(H, psi0, times, c_ops, [])
>>> qsave(dm_list, 'density_matrix_vs_time')
```

And it can then be loaded and used again, for example in an other program

```
>>> dm_list_loaded = qload('density_matrix_vs_time')
Loaded Result object:
Result object with mesolve data.
--------------------------------
states = True
num_collapse = 0
>>> a = destroy(10)
>>> expect(a.dag() * a, dm_list_loaded.states)
array([4.63317086, 3.59150315, 2.90590183, 2.41306641, 2.05120716,
   1.78312503, 1.58357995, 1.4346382 , 1.32327398, 1.23991233])
```

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

```
>>> a = destroy(10); H = a.dag() * a ; c_ops = [np.sqrt(0.5) * a, np.sqrt(0.25) *␣
↪a.dag()]
>>> psi0 = rand_ket(10)
>>> times = np.linspace(0, 100, 100)
>>> medata = mesolve(H, psi0, times, c_ops, [a.dag() * a, a + a.dag(), -1j * (a -␣
↪a.dag())])
>>> np.shape(medata.expect)
(3, 100)
>>> times.shape
(100,)
>>> output_data = np.vstack((times, medata.expect))    # join time and expt data
>>> file_data_store('expect.dat', output_data.T) # Note the .T for transpose!
>>> with open("expect.dat", "r") as f:
...     print('\n'.join(f.readlines()[:10]))
# Generated by QuTiP: 100x4 complex matrix in decimal format [',' separated␣
↪values].
0.0000000000+0.0000000000j,3.2109553666+0.0000000000j,0.3689771549+0.0000000000j,0.
↪0185002867+0.0000000000j
1.0101010101+0.0000000000j,2.6754598872+0.0000000000j,0.1298251132+0.0000000000j,-
↪0.3303672956+0.0000000000j
2.0202020202+0.0000000000j,2.2743186810+0.0000000000j,-0.2106241300+0.0000000000j,-
↪0.2623894277+0.0000000000j
3.0303030303+0.0000000000j,1.9726633457+0.0000000000j,-0.3037311621+0.0000000000j,
↪0.0397330921+0.0000000000j
4.0404040404+0.0000000000j,1.7435892209+0.0000000000j,-0.1126550232+0.0000000000j,
↪0.2497182058+0.0000000000j
```

(continues on next page)

(continued from previous page)

```
5.0505050505+0.0000000000j,1.5687324121+0.0000000000j,0.1351622725+0.0000000000j,0.
↪2018398581+0.0000000000j
6.0606060606+0.0000000000j,1.4348632045+0.0000000000j,0.2143080535+0.0000000000j,-
↪0.0067820038+0.0000000000j
7.0707070707+0.0000000000j,1.3321818015+0.0000000000j,0.0950352763+0.0000000000j,-
↪0.1630920429+0.0000000000j
8.0808080808+0.0000000000j,1.2533244850+0.0000000000j,-0.0771210981+0.0000000000j,-
↪0.1468923919+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

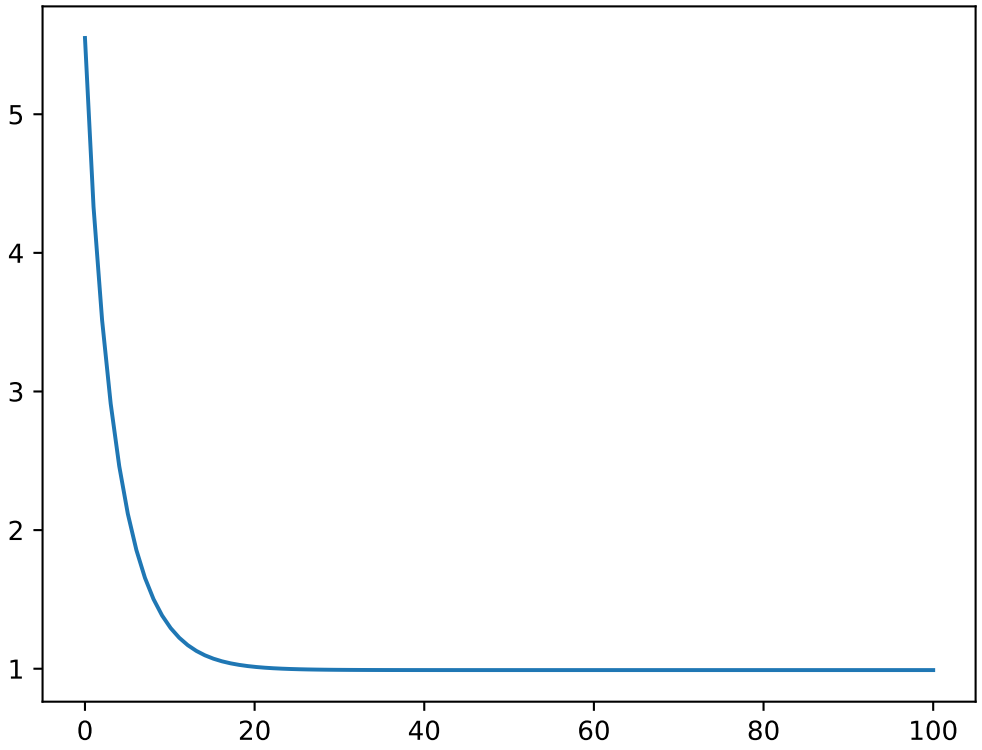
```
>>> file_data_store('expect.dat', output_data.T, numtype="real")
>>> with open("expect.dat", "r") as f:
...     print('\n'.join(f.readlines()[:5]))
# Generated by QuTiP: 100x4 real matrix in decimal format [',' separated values].
0.0000000000,3.2109553666,0.3689771549,0.0185002867
1.0101010101,2.6754598872,0.1298251132,-0.3303672956
2.0202020202,2.2743186810,-0.2106241300,-0.2623894277
3.0303030303,1.9726633457,-0.3037311621,0.0397330921
```

and if we prefer scientific notation we can request that using the `numformat="exp"` option

```
>>> file_data_store('expect.dat', output_data.T, numtype="real", numformat="exp")
```

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.

```
input_data = file_data_read('expect.dat')
plt.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:

```
>>> rand_herm(5)
Quantum object: dims = [[5], [5]], shape = (5, 5), type = oper, isherm = True
Qobj data =
[[-0.25091976+0.j          0.        +0.j          0.        +0.j
  -0.21793701+0.47037633j -0.23212846-0.61607187j]
 [ 0.        +0.j         -0.88383278+0.j          0.836086  -0.23956218j
  -0.09464275+0.45370863j -0.15243356+0.65392096j]
 [ 0.        +0.j          0.836086  +0.23956218j  0.66488528+0.j
  -0.26290446+0.64984451j -0.52603038-0.07991553j]
 [-0.21793701-0.47037633j -0.09464275-0.45370863j -0.26290446-0.64984451j
  -0.13610996+0.j         -0.34240902-0.2879303j ]
 [-0.23212846+0.61607187j -0.15243356-0.65392096j -0.52603038+0.07991553j
  -0.34240902+0.2879303j   0.        +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 specifies the 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.

```
>>> rand_super_bcsz(7).dims
[[[7], [7]], [[7], [7]]]
>>> rand_super_bcsz(6, dims=[[[2, 3], [2, 3]], [[2, 3], [2, 3]]]).dims
[[[2, 3], [2, 3]], [[2, 3], [2, 3]]]
```

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,

```
>>> rand_dm(5, density=0.5)
Quantum object: dims = [[5], [5]], shape = (5, 5), type = oper, isherm = True
Qobj data =
[[ 0.05157906+0.j          0.04491736+0.01043329j  0.06966148+0.00344713j
   0.        +0.j          0.04031493-0.01886791j]
 [ 0.04491736-0.01043329j  0.33632352+0.j         -0.08046093+0.02954712j
   0.0037455 +0.03940256j -0.05679126-0.01322392j]
 [ 0.06966148-0.00344713j -0.08046093-0.02954712j  0.2938209 +0.j
   0.0029377 +0.04463531j  0.05318743-0.02817689j]
 [ 0.        +0.j          0.0037455 -0.03940256j  0.0029377 -0.04463531j
   0.22553181+0.j          0.01657495+0.06963845j]
 [ 0.04031493+0.01886791j -0.05679126+0.01322392j  0.05318743+0.02817689j
   0.01657495-0.06963845j  0.09274471+0.j        ]]

>>> rand_dm_ginibre(5, rank=2)
Quantum object: dims = [[5], [5]], shape = (5, 5), type = oper, isherm = True
Qobj data =
[[ 0.07318288+2.60675616e-19j  0.10426866-6.63115850e-03j
  -0.05377455-2.66949369e-02j -0.01623153+7.66824687e-02j
  -0.12255602+6.11342416e-02j]
 [ 0.10426866+6.63115850e-03j  0.30603789+1.44335373e-18j
  -0.03129486-4.16194216e-03j -0.09832531+1.74110000e-01j
  -0.27176358-4.84608761e-02j]
 [-0.05377455+2.66949369e-02j -0.03129486+4.16194216e-03j
   0.07055265-8.76912454e-19j -0.0183289 -2.72720794e-02j
   0.01196277-1.01037189e-01j]
 [-0.01623153-7.66824687e-02j -0.09832531-1.74110000e-01j
```

(continues on next page)

(continued from previous page)

```
  -0.0183289 +2.72720794e-02j  0.14168414-1.51340961e-19j
   0.07847628+2.07735199e-01j]
 [-0.12255602-6.11342416e-02j -0.27176358+4.84608761e-02j
   0.01196277+1.01037189e-01j  0.07847628-2.07735199e-01j
   0.40854244-6.75775934e-19j]]
```

See the API documentation: *Quantum Objects* 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

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,

```
>>> eigs = np.arange(5)

>>> H = rand_herm(eigs, density=0.5)

>>> H
Quantum object: dims = [[5], [5]], shape = (5, 5), type = oper, isherm = True
Qobj data =
[[ 2.51387054-5.55111512e-17j  0.81161447+2.02283642e-01j
   0.        +0.00000000e+00j  0.875     +3.35634092e-01j
   0.81161447+2.02283642e-01j]
 [ 0.81161447-2.02283642e-01j  1.375     +0.00000000e+00j
   0.        +0.00000000e+00j -0.76700198+5.53011066e-01j
   0.375     +0.00000000e+00j]
 [ 0.        +0.00000000e+00j  0.        +0.00000000e+00j
   2.        +0.00000000e+00j  0.        +0.00000000e+00j
   0.        +0.00000000e+00j]
 [ 0.875     -3.35634092e-01j -0.76700198-5.53011066e-01j
   0.        +0.00000000e+00j  2.73612946+0.00000000e+00j
  -0.76700198-5.53011066e-01j]
 [ 0.81161447-2.02283642e-01j  0.375     +0.00000000e+00j
   0.        +0.00000000e+00j -0.76700198+5.53011066e-01j
   1.375     +0.00000000e+00j]]


>>> H.eigenenergies()
array([7.70647994e-17, 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:

```
>>> rand_dm(4, 0.5, dims=[[2,2], [2,2]])
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper, isherm = True
Qobj data =
[[ 0.13622928+0.j          0.        +0.j          0.01180807-0.01739166j
   0.        +0.j        ]
 [ 0.        +0.j          0.14600238+0.j          0.10335328+0.21790786j
  -0.00426027-0.02193627j]
 [ 0.01180807+0.01739166j  0.10335328-0.21790786j  0.57566072+0.j
  -0.0670631 +0.04124094j]
 [ 0.        +0.j         -0.00426027+0.02193627j -0.0670631 -0.04124094j
   0.14210761+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 multiprocessing.	*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.settings.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
```

3.16 Quantum Information Processing

3.16.1 Quantum Information Processing

Introduction

The Quantum Information Processing (QIP) module aims at providing basic tools for quantum computing simulation both for simple quantum algorithm design and for experimental realization. It offers two different approaches, one with *QubitCircuit* calculating unitary evolution under quantum gates by matrix product, another called *Processor* using open system solvers in QuTiP to simulate noisy quantum device.

Quantum Circuit

The most common model for quantum computing is the quantum circuit model. In QuTiP, we use *QubitCircuit* to represent a quantum circuit. The circuit is characterized by registers and gates:

- **Registers**: The argument `N` specifies the number of qubit registers in the circuit and the argument `num_cbits` (optional) specifies the number of classical bits available for measurement and control.
- **Gates**: Each quantum gate is saved as a class object *Gate* with information such as gate name, target qubits and arguments. Gates can also be controlled on a classical bit by specifying the register number with the argument `classical_controls`.
- **Measurements**: We can also carry out measurements on individual qubit (both in the middle and at the end of the circuit). Each measurement is saved as a class object *Measurement* with parameters such as *targets*, the target qubit on which the measurement will be carried out, and *classical_store*, the index of the classical register which stores the result of the measurement.

A circuit with the various gates and registers available is demonstrated below:

```
from qutip.qip.circuit import QubitCircuit, Gate
from qutip import tensor, basis

qc = QubitCircuit(N=2, num_cbits=1)
swap_gate = Gate(name="SWAP", targets=[0, 1])
```

(continues on next page)

(continued from previous page)

```
qc.add_gate(swap_gate)
qc.add_measurement("M0", targets=[1], classical_store=0) # measurement gate
qc.add_gate("CNOT", controls=0, targets=1)
qc.add_gate("X", targets=0, classical_controls=[0]) # classically controlled gate
qc.add_gate(swap_gate)

print(qc.gates)
```

Output:

```
[Gate(SWAP, targets=[0, 1], controls=None, classical controls=None, control_
↪value=None),
 Measurement(M0, target=[1], classical_store=0),
 Gate(CNOT, targets=[1], controls=[0], classical controls=None, control_
↪value=None),
 Gate(X, targets=[0], controls=None, classical controls=[0], control_value=None),
 Gate(SWAP, targets=[0, 1], controls=None, classical controls=None, control_
↪value=None)]
```

Unitaries

There are a few useful functions associated with the circuit object. For example, the *propagators* method returns a list of the unitaries associated with the sequence of gates in the circuit. By default, the unitaries are expanded to the full dimension of the circuit:

```
U_list = qc.propagators()
print(U_list)
```

Output:

```
[Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper, isherm =␣
↪True
Qobj data =
[[1. 0. 0. 0.]
 [0. 0. 1. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]], Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type =␣
↪oper, isherm = True
Qobj data =
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]], Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type =␣
↪oper, isherm = True
Qobj data =
[[0. 0. 1. 0.]
 [0. 0. 0. 1.]
 [1. 0. 0. 0.]
 [0. 1. 0. 0.]], Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type =␣
↪oper, isherm = True
Qobj data =
[[1. 0. 0. 0.]
 [0. 0. 1. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]]]
```

Another option is to only return the unitaries in their original dimension. This can be achieved with the argument `expand=False` specified to the *propagators*.

```
U_list = qc.propagators(expand=False)
print(U_list)
```

Output:

```
[Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper, isherm =␣
↪True
Qobj data =
[[1. 0. 0. 0.]
 [0. 0. 1. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]], Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type =␣
↪oper, isherm = True
Qobj data =
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 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], [2, 2]], shape = (4, 4), type = oper,␣
↪isherm = True
Qobj data =
[[1. 0. 0. 0.]
 [0. 0. 1. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]]]
```

Gates

The pre-defined gates for the class *qutip.qip.Gate* are shown in the table below:

Gate name	Description
"RX"	Rotation around x axis
"RY"	Rotation around y axis
"RZ"	Rotation around z axis
"X"	Pauli-X gate
"Y"	Pauli-Y gate
"Z"	Pauli-Z gate
"S"	Single-qubit rotation or Z90
"T"	Square root of S gate
"SQRTNOT"	Square root of NOT gate
"SNOT"	Hardmard gate
"PHASEGATE"	Add a phase one the state 1
"CRX"	Controlled rotation around x axis
"CRY"	Controlled rotation around y axis
"CRZ"	Controlled rotation around z axis
"CX"	Controlled X gate
"CY"	Controlled Y gate
"CZ"	Controlled Z gate
"CS"	Controlled S gate
"CT"	Controlled T gate
"CPHASE"	Controlled phase gate
"CNOT"	Controlled NOT gate
"CSIGN"	Same as CPHASE

continues on next page

Table 3 – continued from previous page

Gate name	Description
“QASMU”	U rotation gate used as a primitive in the QASM standard
“BERKELEY”	Berkeley gate
“SWAPalpha”	SWAPalpha gate
“SWAP”	Swap the states of two qubits
“ISWAP”	Swap gate with additional phase for 01 and 10 states
“SQRTSWAP”	Square root of the SWAP gate
“SQRTISWAP”	Square root of the ISWAP gate
“FREDKIN”	Fredkin gate
“TOFFOLI”	Toffoli gate
“GLOBALPHASE”	Global phase

For some of the gates listed above, *QubitCircuit* also has a primitive *resolve_gates* method that decomposes them into elementary gate sets such as CNOT or SWAP with single-qubit gates (RX, RY and RZ). However, this method is not fully optimized. It is very likely that the depth of the circuit can be further reduced by merging quantum gates. It is required that the gate resolution be carried out before the measurements to the circuit are added.

Custom Gates

In addition to these pre-defined gates, QuTiP also allows the user to define their own gate. The following example shows how to define a customized gate. The key step is to define a gate function returning a *qutip.Qobj* and save it in the attribute `user_gates`.

```
from qutip.qip.circuit import Gate
from qutip.qip.operations import rx

def user_gate1(arg_value):
     # controlled rotation X
     mat = np.zeros((4, 4), dtype=np.complex)
     mat[0, 0] = mat[1, 1] = 1.
     mat[2:4, 2:4] = rx(arg_value)
     return Qobj(mat, dims=[[2, 2], [2, 2]])


def user_gate2():
     # S gate
     mat = np.array([[1.,   0],
                     [0., 1.j]])
     return Qobj(mat, dims=[[2], [2]])

qc = QubitCircuit(2)
qc.user_gates = {"CTRLRX": user_gate1,
                 "S"     : user_gate2}

# qubit 0 controls qubit 1
qc.add_gate("CTRLRX", targets=[0,1], arg_value=np.pi/2)

# qubit 1 controls qubit 0
qc.add_gate("CTRLRX", targets=[1,0], arg_value=np.pi/2)

# we also add a gate using a predefined Gate object
g_T = Gate("S", targets=[1])
qc.add_gate(g_T)
props = qc.propagators()

print(props[0])
```

Output:

```
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          1.         +0.j          0.         +0.j
  0.         +0.j        ]
 [0.         +0.j          0.         +0.j          0.70710678+0.j
  0.         -0.70710678j]
 [0.         +0.j          0.         +0.j          0.         -0.70710678j
  0.70710678+0.j        ]]
```

```
print(props[1])
```

Output:

```
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          0.70710678+0.j          0.         +0.j
  0.         -0.70710678j]
 [0.         +0.j          0.         +0.j          1.         +0.j
  0.         +0.j        ]
 [0.         +0.j          0.         -0.70710678j 0.         +0.j
  0.70710678+0.j        ]]
```

```
print(props[2])
```

Output:

```
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.+1.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.+1.j]]
```

Plotting a Quantum Circuit

A quantum circuit (described above) can directly be plotted using the QCircuit library (https://github.com/CQuIC/qcircuit). QCiruit is a quantum circuit drawing application and is implemented directly into QuTiP.

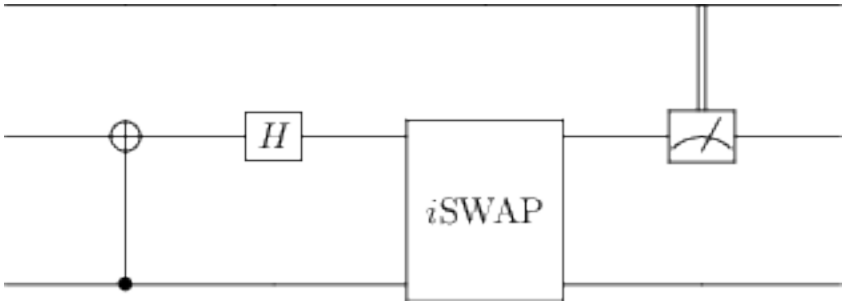
The circuit image visualization requires LaTeX and ImageMagick for display. The module automatically generates the LaTeX code for plotting the circuit, produces the pdf and converts it to the png format. On Mac and Linux, ImageMagick can be easily installed with the command conda install imagemagick if you have conda installed. Otherwise, please follow the installation instructions on the ImageMagick documentation.

On windows, you need to download and install ImageMagick installer. In addition, you also need perl (for `pdfcrop`) and Ghostscript (additional dependency of ImageMagick for png conversion).

If you want to check whether all dependencies are installed, see if the following three commands work correctly: `pdflatex`, `pdfcrop` and `magick anypdf.pdf anypdf.png`, where `anypdf.pdf` is any pdf file you have.

An example code for plotting the example quantum circuit from above is given:

```
from qutip.qip.circuit import QubitCircuit, Gate
# create the quantum circuit
qc = QubitCircuit(2, num_cbits=1)
qc.add_gate("CNOT", controls=0, targets=1)
qc.add_gate("H", targets=1)
qc.add_gate("ISWAP", targets=[0,1])
qc.add_measurement("M0", targets=1, classical_store=0)
# plot the quantum circuit
qc.png
```



Circuit simulation

There are two different ways to simulate the action of quantum circuits using QuTiP:

- The first method utilizes unitary application through matrix products on the input states. This method simulates circuits exactly in a deterministic manner. This is achieved through `CircuitSimulator`. A short guide to exact simulation can be found at *Operator-level circuit simulation*. The teleportation notebook is also useful as an example.
- A different method of circuit simulation employs driving Hamiltonians with the ability to simulate circuits in the presence of noise. This can be achieved through the various classes in `qutip.qip.device`. A short guide to processors for QIP simulation can be found at *Pulse-level circuit simulation*.

3.16.2 Operator-level circuit simulation

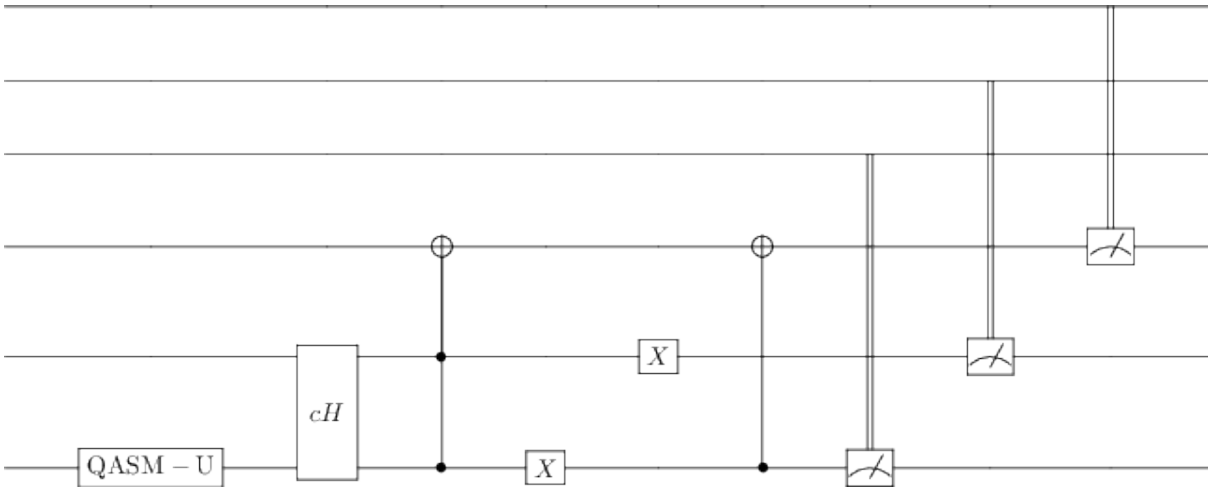
Note: New in QuTiP 4.6

Run a quantum circuit

Let's start off by defining a simple circuit which we use to demonstrate a few examples of circuit evolution. We take a circuit from OpenQASM 2

```
from qutip.qip.circuit import QubitCircuit, Gate
from qutip.qip.operations import controlled_gate, hadamard_transform
def controlled_hadamard():
    # Controlled Hadamard
    return controlled_gate(
        hadamard_transform(1), 2, control=0, target=1, control_value=1)
qc = QubitCircuit(N=3, num_cbits=3)
qc.user_gates = {"cH": controlled_hadamard}
qc.add_gate("QASMU", targets=[0], arg_value=[1.91063, 0, 0])
qc.add_gate("cH", targets=[0,1])
qc.add_gate("TOFFOLI", targets=[2], controls=[0, 1])
qc.add_gate("X", targets=[0])
qc.add_gate("X", targets=[1])
qc.add_gate("CNOT", targets=[1], controls=0)
```

It corresponds to the following circuit:



We will add the measurement gates later. This circuit prepares the W-state $(|001\rangle + |010\rangle + |100\rangle)/\sqrt{3}$. The simplest way to carry out state evolution through a quantum circuit is providing a input state to the *run* method.

```
from qutip import tensor
zero_state = tensor(basis(2, 0), basis(2, 0), basis(2, 0))
result = qc.run(state=zero_state)
wstate = result

print(wstate)
```

Output:

```
Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = (8, 1), type = ket
Qobj data =
[[0.        ]
 [0.57734961]
 [0.57734961]
 [0.        ]
 [0.57735159]
 [0.        ]
 [0.        ]
 [0.        ]]
```

As expected, the state returned is indeed the required W-state.

As soon as we introduce measurements into the circuit, it can lead to multiple outcomes with associated probabilities. We can also carry out circuit evolution in a manner such that it returns all the possible state outputs along with their corresponding probabilities. Suppose, in the previous circuit, we measure each of the three qubits at the end.

```
qc.add_measurement("M0", targets=[0], classical_store=0)
qc.add_measurement("M1", targets=[1], classical_store=1)
qc.add_measurement("M2", targets=[2], classical_store=2)
```

To get all the possible output states along with the respective probability of observing the outputs, we can use the *run_statistics* function:

```
result = qc.run_statistics(state=tensor(basis(2, 0), basis(2, 0), basis(2, 0)))
states = result.get_final_states()
probabilities = result.get_probabilities()

for state, probability in zip(states, probabilities):
    print("State:\n{}\nwith probability {}".format(state, probability))
```

Output:

```
State:
Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = (8, 1), type = ket
Qobj data =
[[0.]
[1.]
[0.]
[0.]
[0.]
[0.]
[0.]
[0.]]
with probability 0.33333257054168813
State:
Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = (8, 1), type = ket
Qobj data =
[[0.]
[0.]
[1.]
[0.]
[0.]
[0.]
[0.]
[0.]]
with probability 0.33333257054168813
State:
Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = (8, 1), type = ket
Qobj data =
[[0.]
[0.]
[0.]
[0.]
[1.]
[0.]
[0.]
[0.]]
with probability 0.33333485891662384
```

The function returns a *CircuitResult* object which contains the output states. The method *run_statistics* can be used to obtain the possible states and probabilities. Since the state created by the circuit is the W-state, we observe the states $|001\rangle$, $|010\rangle$ and $|100\rangle$ with equal probability.

Circuit simulator

The *run* and *run_statistics* functions make use of the *CircuitSimulator* which enables exact simulation with more granular options. The simulator object takes a quantum circuit as an argument. It can optionally be supplied with an initial state. There are two modes in which the exact simulator can function. The default mode is the "state_vector_simulator" mode. In this mode, the state evolution proceeds maintaining the ket state throughout the computation. For each measurement gate, one of the possible outcomes is chosen probabilistically and computation proceeds. To demonstrate, we continue with our previous circuit:

```
from qutip.qip.circuit import CircuitSimulator

sim = CircuitSimulator(qc, state=zero_state)
```

This initializes the simulator object and carries out any pre-computation required. There are two ways to carry out state evolution with the simulator. The primary way is to use the *run* and *run_statistics* functions just like before (only now with the *CircuitSimulator* class).

The *CircuitSimulator* class also enables stepping through the circuit:

```
print(sim.step())
```

Output:

```
Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = (8, 1), type = ket
Qobj data =
[[0.57735159]
 [0.        ]
 [0.        ]
 [0.        ]
 [0.81649565]
 [0.        ]
 [0.        ]
 [0.        ]]
```

This only excutes one gate in the circuit and allows for a better understanding of how the state evolution takes place. The method steps through both the gates and the measurements.

Precomputing the unitary

By default, the *CircuitSimulator* class is initialized such that the circuit evolution is conducted by applying each unitary to the state interactively. However, by setting the argument `precompute_unitary=True`, *CircuitSimulator* precomputes the product of the unitaries (in between the measurements):

```
sim = CircuitSimulator(qc, precompute_unitary=True)

print(sim.ops)
```

```
[Quantum object: dims = [[2, 2, 2], [2, 2, 2]], shape = (8, 8), type = oper,␣
↪isherm = False
  Qobj data =
  [[ 0.          0.57734961  0.         -0.57734961  0.          0.40824922
     0.         -0.40824922]
   [ 0.57734961  0.         -0.57734961  0.          0.40824922  0.
    -0.40824922  0.        ]
   [ 0.57734961  0.          0.57734961  0.          0.40824922  0.
     0.40824922  0.        ]
   [ 0.          0.57734961  0.          0.57734961  0.          0.40824922
     0.          0.40824922]
   [ 0.57735159  0.          0.          0.         -0.81649565  0.
     0.          0.        ]
   [ 0.          0.57735159  0.          0.          0.         -0.81649565
     0.          0.        ]
   [ 0.          0.          0.57735159  0.          0.          0.
    -0.81649565  0.        ]
   [ 0.          0.          0.          0.57735159  0.          0.
     0.         -0.81649565]],
     Measurement(M0, target=[0], classical_store=0),
     Measurement(M1, target=[1], classical_store=1),
     Measurement(M2, target=[2], classical_store=2)]
```

Here, `sim.ops` stores all the circuit operations that are going to be applied during state evolution. As observed above, all the unitaries of the circuit are compressed into a single unitary product with the precompute optimization enabled. This is more efficient if one runs the same circuit one multiple initial states. However, as the number of qubits increases, this will consume more and more memory and become unfeasible.

Density Matrix Simulation

By default, the state evolution is carried out in the "state_vector_simulator" mode (specified by the **mode** argument) as described before. In the "density_matrix_simulator" mode, the input state can be either a ket or a density matrix. If it is a ket, it is converted into a density matrix before the evolution is carried out. Unlike the "state_vector_simulator" mode, upon measurement, the state does not collapse to one of the post-measurement states. Rather, the new state is now the density matrix representing the ensemble of post-measurement states. In this sense, we measure the qubits and forget all the results.

To demonstrate this consider the original W-state preparation circuit which is followed just by measurement on the first qubit:

```
qc = QubitCircuit(N=3, num_cbits=3)
qc.user_gates = {"cH": controlled_hadamard}
qc.add_gate("QASMU", targets=[0], arg_value=[1.91063, 0, 0])
qc.add_gate("cH", targets=[0,1])
qc.add_gate("TOFFOLI", targets=[2], controls=[0, 1])
qc.add_gate("X", targets=[0])
qc.add_gate("X", targets=[1])
qc.add_gate("CNOT", targets=[1], controls=0)
qc.add_measurement("M0", targets=[0], classical_store=0)
qc.add_measurement("M0", targets=[1], classical_store=0)
qc.add_measurement("M0", targets=[2], classical_store=0)
sim = CircuitSimulator(qc, mode="density_matrix_simulator")
print(sim.run(zero_state).get_final_states()[0])
```

```
  Quantum object: dims = [[2, 2, 2], [2, 2, 2]], shape = (8, 8), type = oper,␣
↪isherm = True
  Qobj data =
  [[0.         0.         0.         0.         0.         0.
    0.         0.        ]
   [0.         0.33333257 0.         0.         0.         0.
    0.         0.        ]
   [0.         0.         0.33333257 0.         0.         0.
    0.         0.        ]
   [0.         0.         0.         0.         0.         0.
    0.         0.        ]
   [0.         0.         0.         0.         0.33333486 0.
    0.         0.        ]
   [0.         0.         0.         0.         0.         0.
    0.         0.        ]
   [0.         0.         0.         0.         0.         0.
    0.         0.        ]
   [0.         0.         0.         0.         0.         0.
    0.         0.        ]]
```

We are left with a mixed state.

Import and export quantum circuits

QuTiP supports importation and exportation of quantum circuit in the OpenQASM 2 format through the functions *read_qasm* and *save_qasm*. We demonstrate this using the w-state generation circuit. The following code is in OpenQASM format:

```
// Name of Experiment: W-state v1

OPENQASM 2.0;
include "qelib1.inc";
```

(continues on next page)

(continued from previous page)

```
qreg q[4];
creg c[3];
gate cH a,b {
h b;
sdg b;
cx a,b;
h b;
t b;
cx a,b;
t b;
h b;
s b;
x b;
s a;
}

u3(1.91063,0,0) q[0];
cH q[0],q[1];
ccx q[0],q[1],q[2];
x q[0];
x q[1];
cx q[0],q[1];

measure q[0] -> c[0];
measure q[1] -> c[1];
measure q[2] -> c[2];
```

One can save it in a `.qasm` file and import it using the following code:

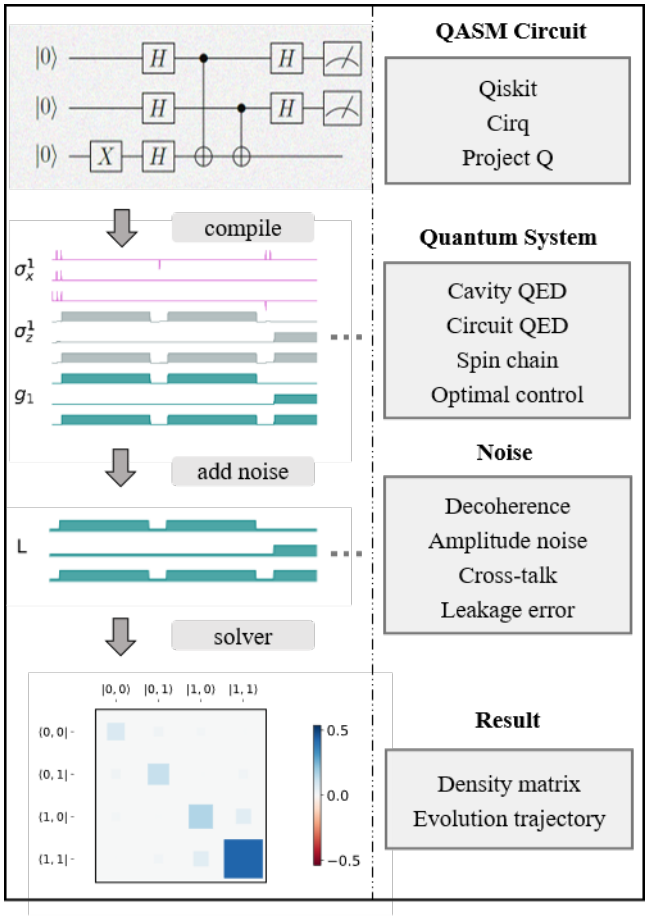
```
from qutip.qip.qasm import read_qasm
qc = read_qasm("guide/qip/w-state.qasm")
```

3.16.3 Pulse-level circuit simulation

Modelling quantum hardware with Processor

Based on the open system solver, *Processor* in the `qutip.qip` module simulates quantum circuits at the level of time evolution. One can consider the processor as a simulator of a quantum device, on which the quantum circuit is to be implemented.

The procedure is illustrated in the figure below. It first compiles circuit into a Hamiltonian model, adds noisy dynamics and then uses the QuTiP open time evolution solvers to simulation the evolution.



Like a real quantum device, the processor is determined by a list of Hamiltonians, i.e. the control pulses driving the evolution. Given the intensity of the control pulses and the corresponding time slices for each pulse, the evolution is then computed. A control pulse is characterized by *Pulse*, consisting of the control Hamiltonian, the targets qubit, the pulse coefficients and the time sequence. We can either use the coefficients as a step function or with cubic spline. For step function, `tlist` specifies the start and the end of each pulse and thus is one element longer the `coeffs`. One example of defining the control pulse coefficients and the time array is as follows:

```
import numpy as np
from qutip import sigmaz
from qutip.qip.device import Processor

processor = Processor(2)
processor.add_control(sigmaz(), cyclic_permutation=True)  # sigmaz for all qubits
processor.pulses[0].coeffs = np.array([[1.0, 1.5, 2.0], [1.8, 1.3, 0.8]])
processor.pulses[0].tlist = np.array([0.1, 0.2, 0.4, 0.5])
```

It defines a σ_z operator on both qubits and a pulse that acts on the first qubit. An equivalent approach is using the *add_pulse* method.

```
from qutip.qip.pulse import Pulse

processor = Processor(2)
coeff=np.array([0.1, 0.2, 0.4, 0.5])
tlist=np.array([[1.0, 1.5, 2.0], [1.8, 1.3, 0.8]])
pulse = Pulse(sigmaz(), targets=0, coeff=coeff, tlist=tlist)
processor.add_pulse(pulse)
```

One can also use choose the `pulse_mode` attribute of *Processor* between `"discrete"` and `"continuous"`.

Note: If the coefficients represent dicrete pulse, the length of each array is 1 element shorter than `tlist`. If it is supposed to be a continuous function, the length should be the same as `tlist`.

The above example shows the framework and the most essential part of the simulator's API. So far, it looks like just a wrapper for the open system solvers. However, based on this, we can implement different physical realizations. They differ mainly in how to find the control pulse for a quantum circuit, which gives birth to different sub-classes:

Processor
├── ModelProcessor
│ ├── DispersiveCavityQED
│ └── SpinChain
└── OptPulseProcessor

In general, there are two ways to find the control pulses. The first one, *ModelProcessor*, is more experiment-oriented and based on physical models. A universal set of gates is defined in the processor as well as the pulse implementing them in this particular physical model. This is usually the case where control pulses realizing those gates are well known and can be concatenated to realize the whole quantum circuits. Two realizations have already been implemented: the spin chain and the Cavity QED model for quantum computing. In those models, the driving Hamiltonians are predefined. Another approach, based on the optimal control module in QuTiP (see *Quantum Optimal Control*), is called *OptPulseProcessor*. In this subclass, one only defines the available Hamiltonians in their system. The processor then uses algorithms to find the optimal control pulses that realize the desired unitary evolution.

Despite this difference, the logic behind all processors is the same:

- One defines a processor by a list of available Hamiltonians and, as explained later, hardware-dependent noise. In model based processors, the Hamiltonians are predefined and one only needs to give the device parameters like frequency and interaction strength.
- The control pulse coefficients and time slices are either specified by the user or calculated by the method *load_circuit*, which takes a *QubitCircuit* and find the control pulse for this evolution.
- The processor calculates the evolution using the QuTiP solvers. Collapse operators can be added to simulate decoherence. The method *run_state* returns a object *qutip.solver.Result*.

It is also possible to calculate the evolution analytically with matrix exponentiation by setting `analytical=True`. A list of the matrices representing the gates is returned just like for *propagators*. However, this does not consider the collapse operators or other noise. As the system size gets larger, this approach will become very inefficient.

In the following we describe the predefined subclasses for *Processor*:

SpinChain

LinearSpinChain and *CircularSpinChain* are quantum computing models base on the spin chain realization. The control Hamiltonians are σ_x, σ_z and $\sigma_x\sigma_x + \sigma_y\sigma_y$. This processor will first decompose the gate into the universal gate set with ISWAP or SQRTISWAP as two-qubit gates, resolve them into quantum gates of adjacent qubits and then calculate the pulse coefficients.

An example of simulating a simple circuit is shown below:

```
from qutip import basis
from qutip.qip.circuit import QubitCircuit
from qutip.qip.device import LinearSpinChain

qc = QubitCircuit(2)
qc.add_gate("X", targets=0)
qc.add_gate("X", targets=1)
```

(continues on next page)

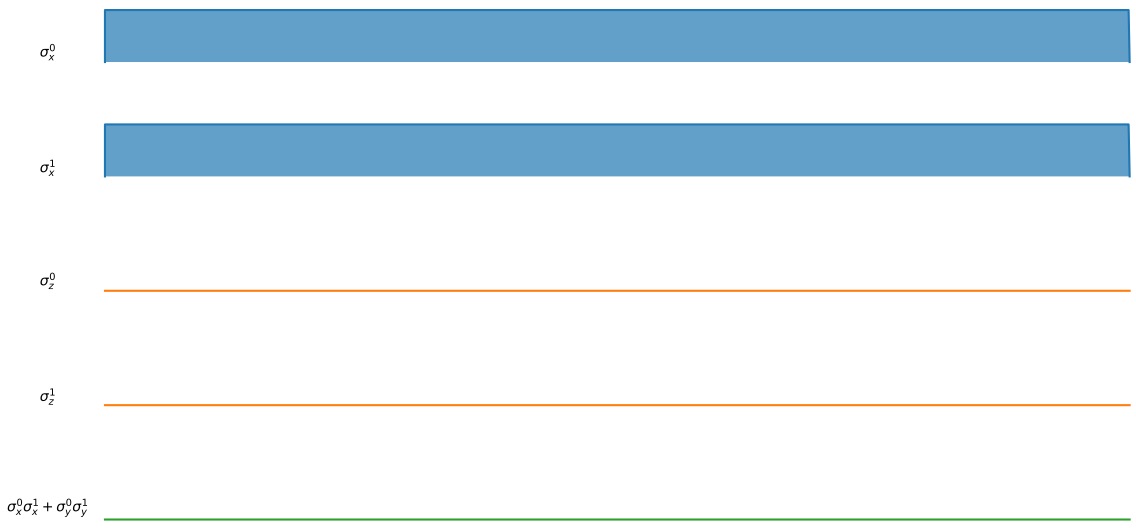
(continued from previous page)

```
processor = LinearSpinChain(2)
processor.load_circuit(qc)
result = processor.run_state(basis([2,2], [0,0]))
print(result.states[-1].tidyup(1.0e-6))
```

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

We can also visualize the pulses implementing this circuit:

```
from qutip import basis
from qutip.qip.circuit import QubitCircuit
from qutip.qip.device import LinearSpinChain

qc = QubitCircuit(2)
qc.add_gate("X", targets=0)
qc.add_gate("X", targets=1)
processor = LinearSpinChain(2)
processor.load_circuit(qc)
fig, axis = processor.plot_pulses()
fig.show()
```



DispersiveCavityQED

Same as above, *DispersiveCavityQED* is a simulator based on Cavity Quantum Electrodynamics. The workflow is similar to the one for the spin chain, except that the component systems are a multi-level cavity and a qubits system. The control Hamiltonians are the single-qubit rotation together with the qubits-cavity interaction $a^\dagger\sigma^- + a\sigma^+$. The device parameters including the cavity frequency, qubits frequency, detuning and interaction strength etc.

Note: The *run_state* method of *DispersiveCavityQED* returns the full simulation result of the solver, hence including the cavity. To obtain the circuit result, one needs to first trace out the cavity state.

OptPulseProcessor

The *OptPulseProcessor* uses the function in *optimize_pulse_unitary* in the optimal control module to find the control pulses. The Hamiltonian includes a drift part and a control part and only the control part will be optimized. The unitary evolution follows

$$U(\Delta t) = \exp\left(\mathrm{i}\cdot\Delta\mathrm{t}[\mathrm{H_d} + \sum_{\mathrm{j}} \mathrm{u_j H_j}]\right)$$

To let it find the optimal pulses, we need to give the parameters for *optimize_pulse_unitary* as keyword arguments to *load_circuit*. Usually, the minimal requirements are the evolution time `evo_time` and the number of time slices `num_tslots` for each gate. Other parameters can also be given in the keyword arguments. For available choices, see *optimize_pulse_unitary*. It is also possible to specify different parameters for different gates, as shown in the following example:

```
from qutip.qip.device import OptPulseProcessor
from qutip.operators import sigmaz, sigmax, sigmay
from qutip.tensor import tensor

# Same parameter for all the gates
qc = QubitCircuit(N=1)
qc.add_gate("SNOT", 0)

num_tslots = 10
evo_time = 10
processor = OptPulseProcessor(N=1, drift=sigmaz())
processor.add_control(sigmax())
# num_tslots and evo_time are two keyword arguments
tlist, coeffs = processor.load_circuit(
qc, num_tslots=num_tslots, evo_time=evo_time)

# Different parameters for different gates
qc = QubitCircuit(N=2)
qc.add_gate("SNOT", 0)
qc.add_gate("SWAP", targets=[0, 1])
qc.add_gate('CNOT', controls=1, targets=[0])

processor = OptPulseProcessor(N=2, drift=tensor([sigmaz()]*2))
processor.add_control(sigmax(), cyclic_permutation=True)
processor.add_control(sigmay(), cyclic_permutation=True)
processor.add_control(tensor([sigmay(), sigmay()]))

setting_args = {"SNOT": {"num_tslots": 10, "evo_time": 1},
                "SWAP": {"num_tslots": 30, "evo_time": 3},
                "CNOT": {"num_tslots": 30, "evo_time": 3}}

tlist, coeffs = processor.load_circuit(
                qc, setting_args=setting_args, merge_gates=False)
```

Compiler and scheduler

Note: New in QuTiP 4.6

In order to simulate quantum circuits at the level of time evolution. We need to first compile the circuit into the Hamiltonian model, i.e. the control pulses. Hence each *Processor* has a corresponding *GateCompiler* class. The compiler takes a *QubitCircuit* and returns the compiled `tlist` and `coeffs`. It is called implicitly when calling the method *run_state*.

```
from qutip.qip.compiler import SpinChainCompiler
qc = QubitCircuit(2)
qc.add_gate("X", targets=0)
qc.add_gate("X", targets=1)

processor = LinearSpinChain(2)
compiler = SpinChainCompiler(
    2, params=processor.params, pulse_dict=processor.pulse_dict)
resolved_qc = qc.resolve_gates(["RX", "RZ", "ISWAP"])
tlists, coeffs = compiler.compile(resolved_qc)
print(tlists)
print(coeffs)
```

Output

```
[array([0., 1.]), array([0., 1., 2.]), None, None, None]
[array([1.57079633]), array([0.        , 1.57079633]), None, None, None]
```

Here we first use *resolve_gates* to decompose the X gate to its natural gate on Spin Chain model, the rotation over X-axis. We pass the hardware parameters of the *SpinChain* model, `processor.params`, as well as a map between the pulse name and pulse index `pulse_dict` to the compiler. The later one allows one to address the pulse more conveniently in the compiler.

The compiler returns a list of `tlist` and `coeff`, corresponding to each pulse. The first pulse starts from `t=0` and ends at `t=1`, with the strengh $\pi/2$. The second one is turned on from `t=1` to `t=2` with the same strength. The compiled pulse here is different from what is shown in the plot in the previous subsection because the scheduler is turned off by default.

The scheduler is implemented in the class *Scheduler*, based on the idea of https://doi.org/10.1117/12.666419. It schedules the order of quantum gates and instructions for the shortest execution time. It works not only for quantum gates but also for pulse implementation of gates (*Instruction*) with varying pulse duration.

The scheduler first generates a quantum gates dependency graph, containing information about which gates have to be executed before some other gates. The graph preserves the mobility of the gates, i.e. commuting gates are not dependent on each other, even if they use the same qubits. Next, it computes the longest distance of each node to the start and end nodes. The distance for each dependency arrow is defined by the execution time of the instruction (By default, it is 1 for all gates). This is used as a priority measure in the next step. The gate with a longer distance to the end node and a shorter distance to the start node has higher priority. In the last step, it uses a list-schedule algorithm with hardware constraint and priority and returns a list of cycles for gates/instructions. Since the algorithm is heuristics, sometimes it does not find the optimal solution. Hence, we offer an option that randomly shuffles the commuting gates and repeats the scheduling a few times to get a better result.

```
from qutip.qip.circuit import QubitCircuit
from qutip.qip.compiler import Scheduler
circuit = QubitCircuit(7)
circuit.add_gate("SNOT", 3)  # gate0
circuit.add_gate("CZ", 5, 3)  # gate1
circuit.add_gate("CZ", 4, 3)  # gate2
circuit.add_gate("CZ", 2, 3)  # gate3
circuit.add_gate("CZ", 6, 5)  # gate4
circuit.add_gate("CZ", 2, 6)  # gate5
circuit.add_gate("ISWAP", [0, 2])  # gate6
scheduler = Scheduler("ASAP")
result = scheduler.schedule(circuit, gates_schedule=True)
print(result)
```

Output

```
[0, 1, 3, 2, 2, 3, 4]
```

The result shows the scheduling order of each gate in the original circuit.

For pulse schedule, or scheduling gates with different duration, one will need to wrap the *qutip.qip.Gate* object with *qutip.qip.compiler.Instruction* object, with a parameter *duration*. The result will then be the start time of each instruction.

Noise Simulation

In the common way of QIP simulation, where evolution is carried out by gate matrix product, the noise is usually simulated with bit flipping and sign flipping errors. The typical approaches are either applying bit/sign flipping gate probabilistically or applying Kraus operators representing different noisy channels (e.g. amplitude damping, dephasing) after each unitary gate evolution. In the case of a single qubit, they have the same effect and the parameters in the Kraus operators are exactly the probability of a flipping error happens during the gate operation time.

Since the processor simulates the state evolution at the level of the driving Hamiltonian, there is no way to apply an error operator to the continuous-time evolution. Instead, the error is added to the pulses (coherent control error) or the collapse operators (Lindblad error) contributing to the evolution. Mathematically, this is no different from adding error channel probabilistically (it is actually how *qutip.mcsolve* works internally). The collapse operator for single-qubit amplitude damping and dephasing are exactly the destroying operator and the sign-flipping operator. One just needs to choose the correct coefficients for them to simulate the noise, e.g. the relaxation time T1 and dephasing time T2. Because it is based on the open system evolution instead of abstract operators, this simulation is closer to the physical implementation and requires less pre-analysis of the system.

Compared to the approach of Kraus operators, this way of simulating noise is more computationally expensive. If you only want to simulate the decoherence of single-qubit relaxation and the relaxation time is much longer than the gate duration, there is no need to go through all the calculations. However, this simulator is closer to the real experiment and, therefore, more convenient in some cases, such as when coherent noise or correlated noise exist. For instance, a pulse on one qubit might affect the neighbouring qubits, the evolution is still unitary but the gate fidelity will decrease. It is not always easy or even possible to define a noisy gate matrix. In our simulator, it can be done by defining a *ControlAmpNoise* (Control Amplitude Noise).

In the simulation, noise can be added to the processor at different levels:

- The decoherence time T1 and T2 can be defined for the processor or for each qubit. When calculating the evolution, the corresponding collapse operators will be added automatically to the solver.
- The noise of the physical parameters (e.g. detuned frequency) can be simulated by changing the parameters in the model, e.g. laser frequency in cavity QED. (This can only be time-independent since QuTiP open system solver only allows varying coefficients, not varying Hamiltonian operators.)
- The noise of the pulse intensity can be simulated by modifying the coefficients of the Hamiltonian operators or even adding new Hamiltonians.

To add noise to a processor, one needs to first define a noise object *Noise*. The simplest relaxation noise can be defined directly in the processor with relaxation time. Other pre-defined noise can be found as subclasses of *Noise*. We can add noise to the simulator with the method *add_noise*.

Below, we show two examples.

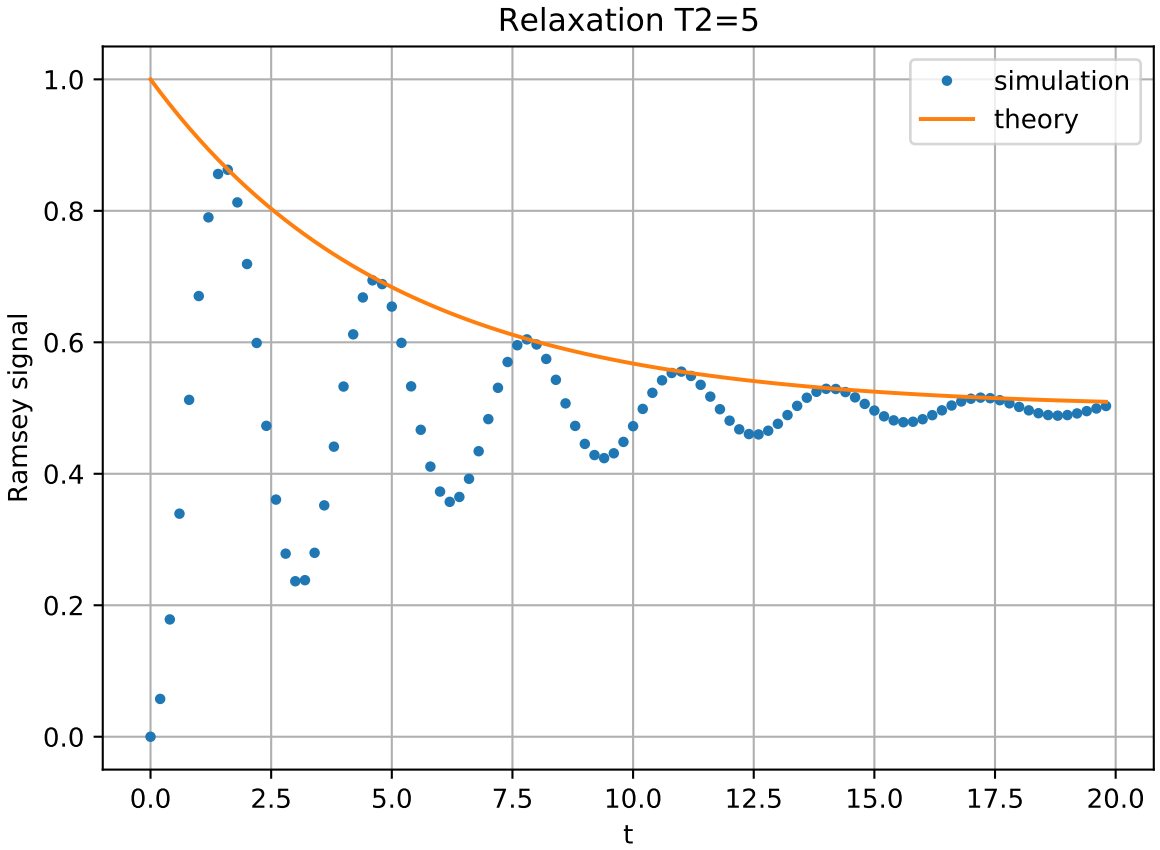
The first example is a processor with one qubit under rotation around the z-axis and relaxation time $T_2 = 5$. We measure the population of the $|+\rangle$ state and observe the Ramsey signal:

```
import numpy as np
import matplotlib.pyplot as plt
from qutip import sigmaz, destroy, basis
from qutip.qip.device import Processor
from qutip.qip.operations import snot

a = destroy(2)
Hadamard = snot()
plus_state = (basis(2,1) + basis(2,0)).unit()
tlist = np.arange(0.00, 20.2, 0.2)
```

(continues on next page)

(continued from previous page)

```
T2 = 5
processor = Processor(1, t2=T2)
processor.add_control(sigmaz())
processor.pulses[0].coeff = np.ones(len(tlist))
processor.pulses[0].tlist = tlist
result = processor.run_state(
    plus_state, e_ops=[a.dag()*a, Hadamard*a.dag()*a*Hadamard])

fig, ax = plt.subplots()
# detail about length of tlist needs to be fixed
ax.plot(tlist[:-1], result.expect[1][:-1], '.', label="simulation")
ax.plot(tlist[:-1], np.exp(-1./T2*tlist[:-1])*0.5 + 0.5, label="theory")
ax.set_xlabel("t")
ax.set_ylabel("Ramsey signal")
ax.legend()
ax.set_title("Relaxation T2=5")
ax.grid()
fig.tight_layout()
fig.show()
```



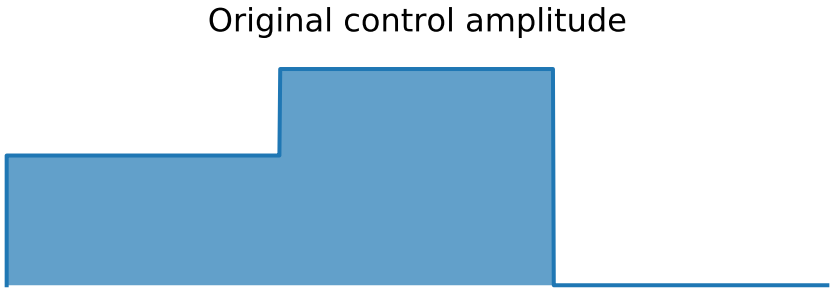
The second example demonstrates a biased Gaussian noise on the pulse amplitude. For visualization purposes, we plot the noisy pulse intensity instead of the state fidelity. The three pulses can, for example, be a zyz-decomposition of an arbitrary single-qubit gate:

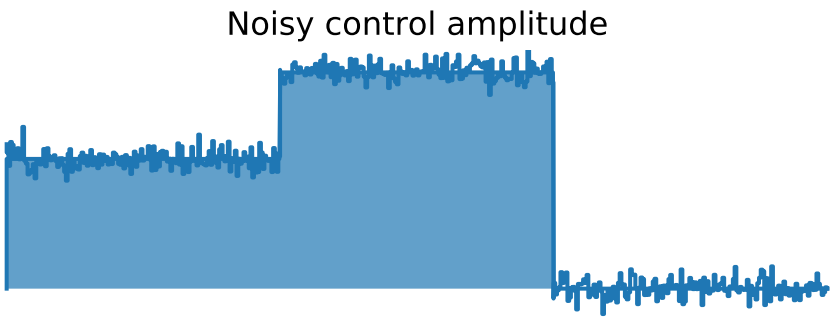
```
import numpy as np
import matplotlib.pyplot as plt
from qutip import sigmaz, sigmay
from qutip.qip.device import Processor
```

(continues on next page)

(continued from previous page)

```
from qutip.qip.noise import RandomNoise

# add control Hamiltonians
processor = Processor(N=1)
processor.add_control(sigmaz(), targets=0)

# define pulse coefficients and tlist for all pulses
processor.pulses[0].coeff = np.array([0.3, 0.5, 0. ])
processor.set_all_tlist(np.array([0., np.pi/2., 2*np.pi/2, 3*np.pi/2]))

# define noise, loc and scale are keyword arguments for np.random.normal
gaussnoise = RandomNoise(
            dt=0.01, rand_gen=np.random.normal, loc=0.00, scale=0.02)
processor.add_noise(gaussnoise)

# Plot the ideal pulse
fig1, axis1 = processor.plot_pulses(title="Original control amplitude", figsize=(5,
↪3))

# Plot the noisy pulse
qobjevo, _ = processor.get_qobjevo(noisy=True)
noisy_coeff = qobjevo.to_list()[1][1] + qobjevo.to_list()[2][1]
fig2, axis2 = processor.plot_pulses(title="Noisy control amplitude", figsize=(5,3))
axis2[0].step(qobjevo.tlist, noisy_coeff)
```



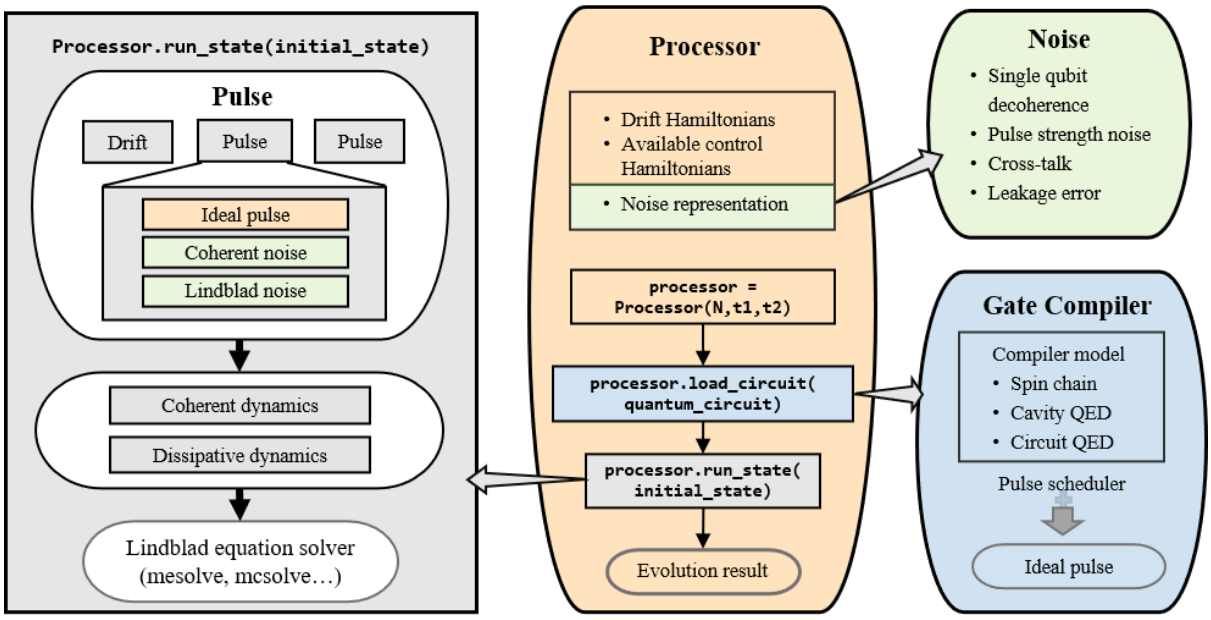


Customize the simulator

The number of predefined physical models and compilers are limited. However, it is designed for easy customization and one can easily build customized model and compiling routines. For guide and examples, please refer to the tutorial notebooks at https://qutip.org/tutorials.html

The workflow of the simulator

The following plot demonstrates the workflow of the simulator.



The core of the simulator is `Processor`, which characterizes the quantum hardware of interest, containing the information such as the non-controllable drift Hamiltonian and the control Hamiltonian. Apart from the ideal system representing the qubits, one can also define hardware-dependent or pulse-dependent noise in `Noise`. It describes how noisy terms such as imperfect control and decoherence can be added once the ideal control pulse is defined. When loading a quantum circuit, a `GateCompiler` compiles the circuit into a sequence of control pulse signals and schedule the pulse for parallel execution. For each control Hamiltonian, a `Pulse` instance is created that including the ideal evolution and associated noise. They will then be sent to the QuTiP solvers for the computation.

3.17 Measurement of Quantum Objects

Note: New in QuTiP 4.6

3.17.1 Introduction

Measurement is a fundamental part of the standard formulation of quantum mechanics and is the process by which classical readings are obtained from a quantum object. Although the interpretation of the procedure is at times contentious, the procedure itself is mathematically straightforward and is described in many good introductory texts.

Here we will show you how to perform simple measurement operations on QuTiP objects. The same functions *measure* and *measurement_statistics* can be used to handle both observable-style measurements and projective style measurements.

3.17.2 Performing a basic measurement (Observable)

First we need to select some states to measure. For now, let us create an *up* state and a *down* state:

```
up = basis(2, 0)

down = basis(2, 1)
```

which represent spin-1/2 particles with their spin pointing either up or down along the z-axis.

We choose what to measure (in this case) by selecting a **measurement operator**. For example, we could select *sigmaz* which measures the z-component of the spin of a spin-1/2 particle, or *sigmax* which measures the x-component:

```
spin_z = sigmaz()

spin_x = sigmax()
```

How do we know what these operators measure? The answer lies in the measurement procedure itself:

- A quantum measurement transforms the state being measured by projecting it into one of the eigenvectors of the measurement operator.
- Which eigenvector to project onto is chosen probabilistically according to the square of the amplitude of the state in the direction of the eigenvector.
- The value returned by the measurement is the eigenvalue corresponding to the chosen eigenvector.

Note: How to interpret this "random choosing" is the famous "quantum measurement problem".

The eigenvectors of *spin_z* are the states with their spin pointing either up or down, so it measures the component of the spin along the z-axis.

The eigenvectors of *spin_x* are the states with their spin pointing either left or right, so it measures the component of the spin along the x-axis.

When we measure our *up* and *down* states using the operator *spin_z*, we always obtain:

```
from qutip.measurement import measure, measurement_statistics

measure(up, spin_z) == (1.0, up)

measure(down, spin_z) == (-1.0, down)
```

because *up* is the eigenvector of *spin_z* with eigenvalue *1.0* and *down* is the eigenvector with eigenvalue *-1.0*. The minus signs are just an arbitrary global phase – *up* and *-up* represent the same quantum state.

Neither eigenvector has any component in the direction of the other (they are orthogonal), so *measure(spin_z, up)* returns the state *up* 100% percent of the time and *measure(spin_z, down)* returns the state *down* 100% of the time.

Note how `measure` returns a pair of values. The first is the measured value, i.e. an eigenvalue of the operator (e.g. *1.0*), and the second is the state of the quantum system after the measurement, i.e. an eigenvector of the operator (e.g. *up*).

Now let us consider what happens if we measure the x-component of the spin of *up*:

```
measure(up, spin_x)
```

The *up* state is not an eigenvector of *spin_x*. *spin_x* has two eigenvectors which we will call *left* and *right*. The *up* state has equal components in the direction of these two vectors, so measurement will select each of them 50% of the time.

These *left* and *right* states are:

```
left = (up - down).unit()

right = (up + down).unit()
```

When *left* is chosen, the result of the measurement will be *(-1.0, -left)*.

When *right* is chosen, the result of measurement with be *(1.0, right)*.

Note: When `measure` is invoked with the second argument being an observable, it acts as an alias to `measure_observable`.

3.17.3 Performing a basic measurement (Projective)

We can also choose what to measure by specifying a *list of projection operators*. For example, we could select the projection operators $|0\rangle\langle 0|$ and $|1\rangle\langle 1|$ which measure the state in the $|0\rangle, |1\rangle$ basis. Note that these projection operators are simply the projectors determined by the eigenstates of the `sigmaz` operator.

```
Z0, Z1 = ket2dm(basis(2, 0)), ket2dm(basis(2, 1))
```

The probabilities and respective output state are calculated for each projection operator.

```
measure(up, [Z0, Z1]) == (0, up)

measure(down, [Z0, Z1]) == (1, down)
```

In this case, the projection operators are conveniently eigenstates corresponding to subspaces of dimension 1. However, this might not be the case, in which case it is not possible to have unique eigenvalues for each eigenstate. Suppose we want to measure only the first qubit in a two-qubit system. Consider the two qubit state $|0+\rangle$

```
state_0 = basis(2, 0)

state_plus = (basis(2, 0) + basis(2, 1)).unit()

state_0plus = tensor(state_0, state_plus)
```

Now, suppose we want to measure only the first qubit in the computational basis. We can do that by measuring with the projection operators $|0\rangle\langle 0| \otimes I$ and $|1\rangle\langle 1| \otimes I$.

```
PZ1 = [tensor(Z0, identity(2)), tensor(Z1, identity(2))]

PZ2 = [tensor(identity(2), Z0), tensor(identity(2), Z1)]
```

Now, as in the previous example, we can measure by supplying a list of projection operators and the state.

```
measure(state_0plus, PZ1) == (0, state_0plus)
```

The output of the measurement is the index of the measurement outcome as well as the output state on the full Hilbert space of the input state. It is crucial to note that we do not discard the measured qubit after measurement (as opposed to when measuring on quantum hardware).

Note: When *measure* is invoked with the second argument being a list of projectors, it acts as an alias to *measure_povm*.

The *measure* function can perform measurements on density matrices too. You can read about these and other details at *measure_povm* and *measure_observable*.

Now you know how to measure quantum states in QuTiP!

3.17.4 Obtaining measurement statistics(Observable)

You've just learned how to perform measurements in QuTiP, but you've also learned that measurements are probabilistic. What if instead of just making a single measurement, we want to determine the probability distribution of a large number of measurements?

One way would be to repeat the measurement many times – and this is what happens in many quantum experiments. In QuTiP one could simulate this using:

```
results = {1.0: 0, -1.0: 0}  # 1 and -1 are the possible outcomes
for _ in range(1000):
   value, new_state = measure(up, spin_x)
   results[round(value)] += 1
print(results)
```

Output:

```
{1.0: 497, -1.0: 503}
```

which measures the x-component of the spin of the *up* state *1000* times and stores the results in a dictionary. Afterwards we expect to have seen the result *1.0* (i.e. left) roughly 500 times and the result *-1.0* (i.e. right) roughly 500 times, but, of course, the number of each will vary slightly each time we run it.

But what if we want to know the distribution of results precisely? In a physical system, we would have to perform the measurement many many times, but in QuTiP we can peak at the state itself and determine the probability distribution of the outcomes exactly in a single line:

```
>>> eigenvalues, eigenstates, probabilities = measurement_statistics(up, spin_x)

>>> eigenvalues
array([-1., 1.])

>>> eigenstates
array([Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[ 0.70710678]
 [-0.70710678]],
       Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[0.70710678]
 [0.70710678]]], dtype=object)

>>> probabilities
[0.5000000000000001, 0.4999999999999999]
```

The *measurement_statistics* function then returns three values when called with a single observable:

- *eigenvalues* is an array of eigenvalues of the measurement operator, i.e. a list of the possible measurement results. In our example the value is *array([-1., -1.])*.
- *eigenstates* is an array of the eigenstates of the measurement operator, i.e. a list of the possible final states after the measurement is complete. Each element of the array is a *Qobj*.
- *probabilities* is a list of the probabilities of each measurement result. In our example the value is *[0.5, 0.5]* since the *up* state has equal probability of being measured to be in the left (*-1.0*) or right (*1.0*) eigenstates.

All three lists are in the same order – i.e. the first eigenvalue is *eigenvalues[0]*, its corresponding eigenstate is *eigenstates[0]*, and its probability is *probabilities[0]*, and so on.

Note: When *measurement_statistics* is invoked with the second argument being an observable, it acts as an alias to *measurement_statistics_observable*.

3.17.5 Obtaining measurement statistics(Projective)

Similarly, when we want to obtain measurement statistics for projection operators, we can use the *measurement_statistics* function with the second argument being a list of projectors. Consider again, the state $|0+\rangle$. Suppose, now we want to obtain the measurement outcomes for the second qubit. We must use the projectors specified earlier by *PZ2* which allow us to measure only on the second qubit. Since the second qubit has the state $|+\rangle$, we get the following result.

```
collapsed_states, probabilities = measurement_statistics(state_0plus, PZ2)

print(collapsed_states)
```

Output:

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

```
print(probabilities)
```

Output:

```
[0.4999999999999999, 0.4999999999999999]
```

The function *measurement_statistics* then returns two values:

- *collapsed_states* is an array of the possible final states after the measurement is complete. Each element of the array is a *Qobj*.
- *probabilities* is a list of the probabilities of each measurement outcome.

Note that the collapsed_states are exactly $|00\rangle$ and $|01\rangle$ with equal probability, as expected. The two lists are in the same order.

Note: When *measurement_statistics* is invoked with the second argument being a list of projectors, it acts as an alias to *measurement_statistics_povm*.

The *measurement_statistics* function can provide statistics for measurements of density matrices too. You can read about these and other details at *measurement_statistics_observable* and *measurement_statistics_povm*.

Furthermore, the *measure_povm* and *measurement_statistics_povm* functions can handle POVM measurements which are more general than projective measurements.

Chapter 4

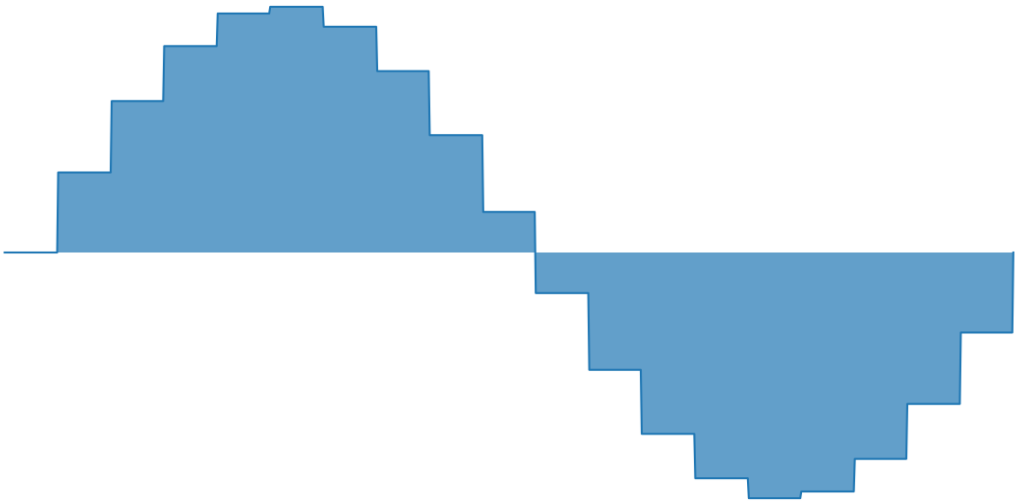
Gallery

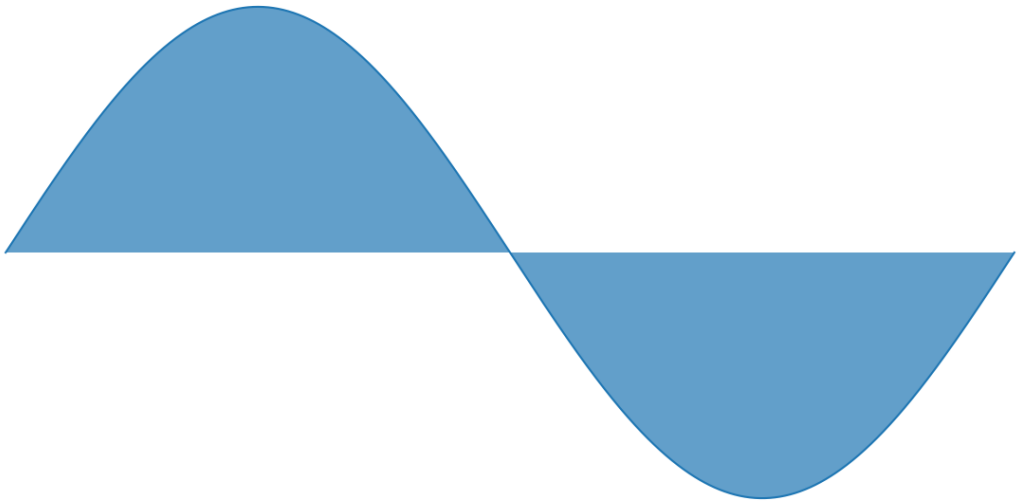
This is the gallery for QuTiP examples, you can click on the image to see the source code.

4.1 Quantum Information Processing

4.1.1 Basic use of Processor

This example contains the basic functions of *qutip.qip.device.Processor*. We define a simulator with control Hamiltonian, pulse amplitude and time slice for each pulse. The two figures illustrate the pulse shape for two different setup: step function or continuous pulse.





Out:

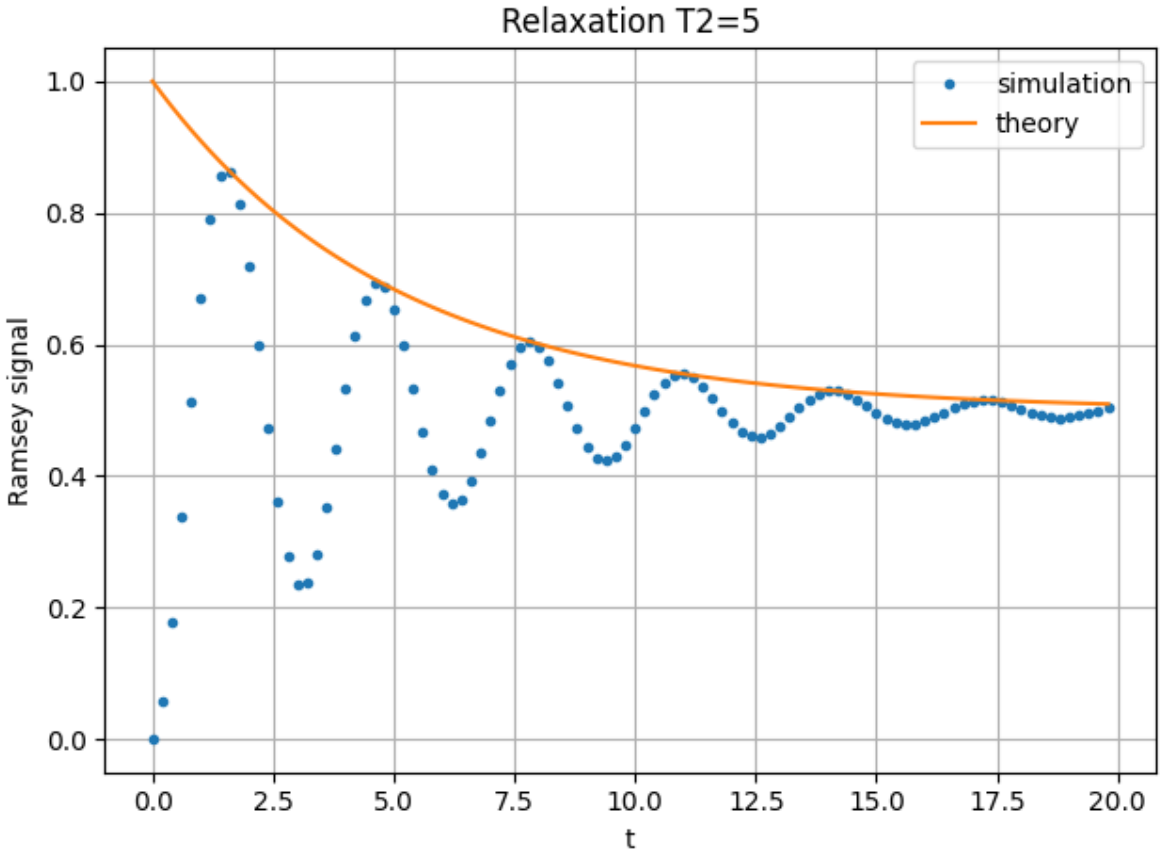
```
(<Figure size 1200x600 with 1 Axes>, [<AxesSubplot:>])
```

```
import copy
import numpy as np
import matplotlib.pyplot as plt
pi = np.pi
from qutip.qip.device import Processor
from qutip.operators import sigmaz
from qutip.states import basis

processor = Processor(N=1)
processor.add_control(sigmaz(), targets=0)

tlist = np.linspace(0., 2*np.pi, 20)
processor = Processor(N=1, spline_kind="step_func")
processor.add_control(sigmaz(), 0)
processor.pulses[0].tlist = tlist
processor.pulses[0].coeff = np.array([np.sin(t) for t in tlist])
processor.plot_pulses()

tlist = np.linspace(0., 2*np.pi, 20)
processor = Processor(N=1, spline_kind="cubic")
processor.add_control(sigmaz())
processor.pulses[0].tlist = tlist
processor.pulses[0].coeff = np.array([np.sin(t) for t in tlist])
processor.plot_pulses()
```

Total running time of the script: (0 minutes 0.877 seconds)

4.1.2 T2 Relaxation

Simulating the T2 relaxation of a single qubit with *qutip.qip.device.Processor*. The single qubit is driven by a rotation around z axis. We measure the population of the plus state as a function of time to see the Ramsey signal.




```
import numpy as np
import matplotlib.pyplot as plt
from qutip.qip.device import Processor
from qutip.operators import sigmaz, destroy
from qutip.qip.operations import snot
from qutip.states import basis

a = destroy(2)
Hadamard = snot()
plus_state = (basis(2,1) + basis(2,0)).unit()
tlist = np.arange(0.00, 20.2, 0.2)

T2 = 5
processor = Processor(1, t2=T2)
processor.add_control(sigmaz())
processor.pulses[0].coeff = np.ones(len(tlist))
processor.pulses[0].tlist = tlist
result = processor.run_state(
    plus_state, e_ops=[a.dag()*a, Hadamard*a.dag()*a*Hadamard])

fig, ax = plt.subplots()
# detail about length of tlist needs to be fixed
ax.plot(tlist[:-1], result.expect[1][:-1], '.', label="simulation")
ax.plot(tlist[:-1], np.exp(-1./T2*tlist[:-1])*0.5 + 0.5, label="theory")
```

(continues on next page)

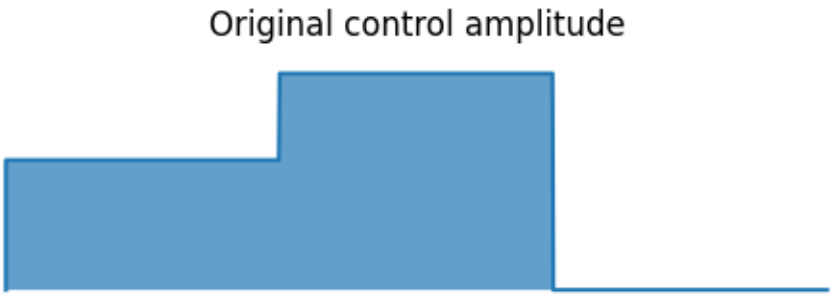
(continued from previous page)

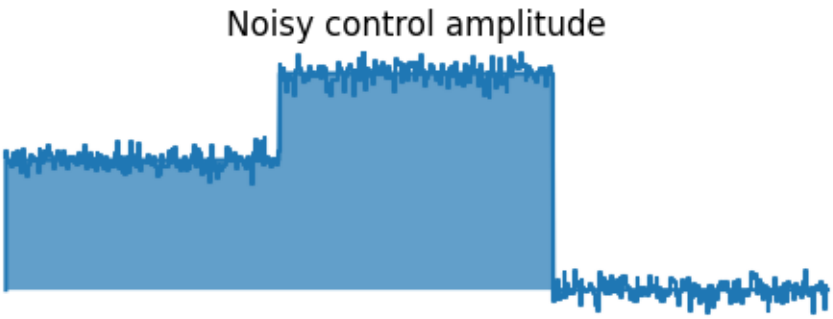
```
ax.set_xlabel("t")
ax.set_ylabel("Ramsey signal")
ax.legend()
ax.set_title("Relaxation T2=5")
ax.grid()
fig.tight_layout()
fig.show()
```

Total running time of the script: (0 minutes 0.138 seconds)

4.1.3 Control Amplitude Noise

This example demonstrates how to add Gaussian noise to the control pulse.



- 

Out:

```
[<matplotlib.lines.Line2D object at 0x7fe6c60e2370>]
```

```
import numpy as np
import matplotlib.pyplot as plt
from qutip.qip.device import Processor
from qutip.qip.noise import RandomNoise
from qutip.operators import sigmaz, sigmay

# add control Hamiltonians
processor = Processor(N=1)
processor.add_control(sigmaz(), targets=0)

# define pulse coefficients and tlist for all pulses
processor.pulses[0].coeff = np.array([0.3, 0.5, 0. ])
processor.set_all_tlist(np.array([0., np.pi/2., 2*np.pi/2, 3*np.pi/2]))

# define noise, loc and scale are keyword arguments for np.random.normal
gaussnoise = RandomNoise(
            dt=0.01, rand_gen=np.random.normal, loc=0.00, scale=0.02)
processor.add_noise(gaussnoise)

# Plot the ideal pulse
processor.plot_pulses(title="Original control amplitude", figsize=(5,3))

# Plot the noisy pulse
qobjevo, _ = processor.get_qobjevo(noisy=True)
noisy_coeff = qobjevo.to_list()[1][1] + qobjevo.to_list()[2][1]
fig2, ax2 = processor.plot_pulses(title="Noisy control amplitude", figsize=(5,3))
ax2[0].step(qobjevo.tlist, noisy_coeff)
```

Total running time of the script: (0 minutes 0.072 seconds)

Chapter 5

API documentation

This chapter contains automatically generated API documentation, including a complete list of QuTiP's public classes and functions.

5.1 Classes

5.1.1 Qobj

class Qobj(*inpt=None*, *dims=None*, *shape=None*, *type=None*, *isherm=None*, *copy=True*, *fast=False*, *superrep=None*, *isunitary=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'.

superrep [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.

isunitary [bool] Indictaes if quantum object represents unitary 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.

isoperket [bool] Indicates if the quantum object represents an operator in column vector form.

isoperbra [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.
inv()	Return a Qobj corresponding to the matrix inverse of the operator.
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.
proj()	Computes the projector for a ket or bra vector.
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.

check_isunitary()
Checks whether qobj is a unitary matrix

conj()
Conjugate operator of quantum object.

copy()
Create identical copy

cosm()
Cosine of a quantum operator.

Operator must be square.

Returns

oper [*qutip.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 [*qutip.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*, *phase_fix=None*)
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).

phase_fix [int, None] If not None, set the phase of each kets so that ket[phase_fix,0] is real positive.

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 [*qutip.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 [*qutip.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 [*qutip.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 [*qutip.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 [*qutip.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.

inv(*sparse=False*)
Matrix inverse of a quantum operator

Operator must be square.

Returns

oper [*qutip.Qobj*] Matrix inverse of operator.

Raises

TypeError Quantum object is not square.

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 [*qutip.Qobj*] Quantum object of type 'bra' or 'ket'

ket [*qutip.Qobj*] Quantum object of type 'ket'.

Returns

elem [complex] Complex valued matrix element.

Notes

It is slightly more computationally efficient to use a ket vector for the 'bra' input.

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(*other*)
Overlap between two state vectors or two operators.

Gives the overlap (inner product) between the current bra or ket Qobj and and another bra or ket Qobj. It gives the Hilbert-Schmidt overlap when one of the Qobj is an operator/density matrix.

Parameters

other [*qutip.Qobj*] Quantum object for a state vector of type 'ket', 'bra' or density matrix.

Returns

overlap [complex] Complex valued overlap.

Raises

TypeError Can only calculate overlap between a bra, ket and density matrix quantum objects.

Notes

Since QuTiP mainly deals with ket vectors, the most efficient inner product call is the ket-ket version that computes the product <selflother> with both vectors expressed as kets.

permute(*order*)
Permutes a composite quantum object.

Parameters

order [list/array] List specifying new tensor order.

Returns

P [*qutip.Qobj*] Permuted quantum object.

proj()
Form the projector from a given ket or bra vector.

Parameters

Q [*qutip.Qobj*] Input bra or ket vector

Returns

P [*qutip.Qobj*] Projection operator.

ptrace(*sel*, *sparse=None*)
Partial trace of the quantum object.

Parameters

sel [int/list] An `int` or `list` of components to keep after partial trace. The order is unimportant; no transposition will be done and the spaces will remain in the same order in the output.

Returns

oper [*qutip.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.

purity()
Calculate purity of a quantum object.

Returns

state_purity [float] Returns the purity of a quantum object. For a pure state, the purity is 1. For a mixed state of dimension *d*, 1/d<=purity<1.

sinm()
Sine of a quantum operator.

Operator must be square.

Returns

oper [*qutip.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 [*qutip.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 [*qutip.Qobj*] Quantum object with small elements removed.

tr()
Trace of a quantum object.

Returns

trace [float] Returns the trace of the quantum object.

trans()
Transposed operator.

Returns

oper [*qutip.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 [*qutip.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 [*qutip.Qobj*] A valid density operator.

unit(*inplace=False*, *norm=None*, *sparse=False*, *tol=0*, *maxiter=100000*)
Operator or state normalized to unity.

Uses norm from Qobj.norm().

Parameters

inplace [bool] Do an in-place normalization

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 [*qutip.Qobj*] Normalized quantum object if not in-place, else None.

5.1.2 QobjEvo

class QobjEvo(*Q_object=[]*, *args={}*, *copy=True*, *tlist=None*, *state0=None*, *e_ops=[]*)
A class for representing time-dependent quantum objects, such as quantum operators and states.

Basic math operations are defined:

- +, − : *QobjEvo*, *Qobj*, scalars.
- ⋆: *Qobj*, C number
- / : C number

This object is constructed by passing a list of *Qobj* instances, each of which *may* have an associated scalar time dependence. The list is summed to produce the final result. In other words, if an instance of this class is $Q(t)$, then it is constructed from a set of constant:obj:~*qutip.Qobj* $\{Q_k\}$ and time-dependent scalars $f_k(t)$ by

$$Q(t) = \sum_k f_k(t) Q_k$$

If a scalar $f_k(t)$ is not passed with a given *Qobj*, then that term is assumed to be constant. The next section contains more detail on the allowed forms of the constants, and gives several examples for how to build instances of this class.

Time-dependence formats

There are three major formats for specifying a time-dependent scalar:

- Python function
- string
- array

For function format, the function signature must be `f(t: float, args: dict) -> complex`, for example

```
def f1_t(t, args):
    return np.exp(-1j * t * args["w1"])

def f2_t(t, args):
    return np.cos(t * args["w2"])

H = QobjEvo([H0, [H1, f1_t], [H2, f2_t]], args={"w1":1., "w2":2.})
```

For string-based coeffients, the string must be a compilable python code resulting in a complex. The following symbols are defined:

```
pi   exp   log   log10
erf  zerf  norm  proj
real imag conj abs arg
sin  sinh  asin  asinh
cos  cosh  acos  acosh
tan  tanh  atan  atanh
numpy as np
scipy.special as spe
```

A couple more simple examples:

```
H = QobjEvo([H0, [H1, 'exp(-1j*w1*t)'], [H2, 'cos(w2*t)']],
            args={"w1":1.,"w2":2.})
```

For numpy array format, the array must be an 1d of dtype `np.float64` or `np.complex128`. A list of times (`np.float64`) at which the coeffients must be given as `tlist`. The coeffients array must have the same length as the tlist. The times of the tlist do not need to be equidistant, but must be sorted. By default, a cubic spline interpolation will be used for the coefficient at time t. If the coefficients are to be treated as step functions, use the arguments `args = {"_step_func_coeff": True}`. Examples of array-format usage are:

```
tlist = np.logspace(-5,0,100)
H = QobjEvo([H0, [H1, np.exp(-1j*tlist)], [H2, np.cos(2.*tlist)]],
            tlist=tlist)
```

Mixing time formats is allowed. It is not possible to create a single `QobjEvo` that contains different `tlist` values, however.

Passing arguments

`args` is a dict of (name: object). The name must be a valid Python identifier string, and in general the object can be any type that is supported by the code to be compiled in the string.

There are some "magic" names that can be specified, whose objects will be overwritten when used within *`sesolve`*, *`mesolve`* and *`mcsolve`*. This allows access to the solvers' internal states, and they are updated at every call. The initial values of these dictionary elements is unimportant. The magic names available are:

- `"state"`: the current state as a *`Qobj`*
- `"state_vec"`: the current state as a column-stacked 1D `np.ndarray`
- `"state_mat"`: the current state as a 2D `np.ndarray`
- `"expect_op_<n>"`: the current expectation value of the element `e_ops[n]`, which is an argument to the solvers. Replace `<n>` with an integer literal, e.g. `"expect_op_0"`. This will be either real- or complex-valued, depending on whether the state and operator are both Hermitian or not.
- `"collapse"`: (*`mcsolve`* only) a list of the collapses that have occurred during the evolution. Each element of the list is a 2-tuple `(time: float, which: int)`, where `time` is the time this collapse happened, and `which` is an integer indexing the `c_ops` argument to *`mcsolve`*.

Parameters

Q_object [list, *`Qobj`* or *`QobjEvo`*] The time-dependent description of the quantum object. This is of the same format as the first parameter to the general ODE solvers; in general, it is a list of `[Qobj, time_dependence]` pairs that are summed to make the whole object. The `time_dependence` can be any of the formats discussed in the previous section. If a particular term has no time-dependence, then you should just give the `Qobj` instead of the 2-element list.

args [dict, optional] Mapping of `{str: object}`, discussed in greater detail above. The strings can be any valid Python identifier, and the objects are of the consumable

types. See the previous section for details on the "magic" names used to access solver internals.

tlist [array_like, optional] List of the times any numpy-array coefficients describe. This is used only in at least one of the time dependences in `Q_object` is given in Numpy-array format. The times must be sorted, but need not be equidistant. Values inbetween will be interpolated.

Attributes

cte [*Qobj*] Constant part of the QobjEvo.

ops [list of `EvoElement`] Internal representation of the time-dependence structure of the elements.

args [dict] The current value of the `args` dictionary passed into the constructor.

dynamics_args [list] Names of the dynamic arguments that the solvers will generate. These are the magic names that were found in the `args` parameter.

tlist [array_like] List of times at which the numpy-array coefficients are applied.

compiled [str] A string representing the properties of the low-level Cython class backing this object (may be empty).

compiled_qobjevo [`CQobjCte` or `CQobjEvoTd`] Cython version of the QobjEvo.

coeff_get [callable] Object called to obtain a list of all the coefficients at a particular time.

coeff_files [list] Runtime created files to delete with the instance.

dummy_cte [bool] Is self.cte an empty Qobj

const [bool] Indicates if quantum object is constant

type [{"cte", "string", "func", "array", "spline", "mixed_callable", "mixed_compilable"}] Information about the type of coefficients used in the entire object.

num_obj [int] Number of *Qobj* in the QobjEvo.

use_cython [bool] Flag to compile string to Cython or Python

safePickle [bool] Flag to not share pointers between thread.

apply (*function*, **args*, ***kw_args*)

Apply the linear function `function` to every `Qobj` included in this time-dependent object, and return a new `QobjEvo` with the result.

Any additional arguments or keyword arguments will be appended to every function call.

apply_decorator (*function*, **args*, *str_mod=None*, *inplace_np=False*, ***kw_args*)

Apply the given function to every time-dependent coefficient in the quantum object, and return a new object with the result.

Any additional arguments and keyword arguments will be appended to the function calls.

Parameters

function [callable] `(time_dependence, *args, **kwargs) -> time_dependence`. Called on each time-dependent coefficient to produce a new coefficient. The additional arguments and keyword arguments are the ones given to this function.

str_mod [list] A 2-element list of strings, that will additionally wrap any string time-dependences. An existing time-dependence string `x` will become `str_mod[0] + x + str_mod[1]`.

inplace_np [bool, default False] Whether this function should modify Numpy arrays inplace, or be used like a regular decorator. Some decorators create incorrect arrays

as some transformations `f'(t) = f(g(t))` create a mismatch between the array and the associated time list.

arguments(*new_args*)
Update the scoped variables that were passed as `args` to new values.

compile(*code=False*, *matched=False*, *dense=False*, *omp=0*)
Create an associated Cython object for faster usage. This function is called automatically by the solvers.

Parameters

code [bool, default False] Return the code string generated by compilation of any strings.

matched [bool, default False] If True, the underlying sparse matrices used to represent each element of the type will have their structures unified. This may include adding explicit zeros to sparse matrices, but can be faster in some cases due to not having to deal with repeated structural mismatches.

dense [bool, default False] Whether to swap to using dense matrices to back the data.

omp [int, optional] The number of OpenMP threads to use when doing matrix multiplications, if QuTiP was compiled with OpenMP.

Returns

compiled_str [str] (Only if *code* was set to True). The code-generated string of compiled calling code.

compress()
Merge together elements that share the same time-dependence, to reduce the number of matrix multiplications and additions that need to be done to evaluate this object.

Modifies the object inplace.

conj()
Return the matrix elementwise conjugation.

copy()
Return a copy of this object.

dag()
Return the matrix conjugate-transpose (dagger).

expect(*t*, *state*, *herm=False*)
Calculate the expectation value of this operator on the given (time-independent) state at a particular time.

This is more efficient than `expect(QobjEvo(t), state)`.

Parameters

t [float] The time to evaluate this operator at.

state [Qobj or np.ndarray] The state to take the expectation value around.

herm [bool, default False] Whether this operator and the state are both Hermitian. If True, only the real part of the result will be returned.

See also:

expect General-purpose expectation values.

mul_mat(*t*, *mat*)
Multiply this object evaluated at time *t* by a matrix (from the right).

Parameters

t [float] The time to evaluate this object at.

mat [Qobj or np.ndarray] The matrix that is multiplied by this object.

Returns

mat: Qobj or np.ndarray The matrix result in the same type as the input.

mul_vec(*t*, *vec*)
Multiply this object evaluated at time *t* by a vector.

Parameters

t [float] The time to evaluate this object at.

vec [Qobj or np.ndarray] The state-vector to multiply this object by.

Returns

vec: Qobj or np.ndarray The vector result in the same type as the input.

permute(*order*)
Permute the tensor structure of the underlying matrices into a new format.

See also:

Qobj.permute the same operation on constant quantum objects.

tidyup(*atol=None*)
Removes small elements from this quantum object inplace.

to_list()
Return this operator in the list-like form used to initialised it, like can be passed to *mesolve*.

trans()
Return the matrix transpose.

5.1.3 eseries

class eseries(*q=None*, *s=array([], dtype=float64)*)
Class representation of an exponential-series expansion of time-dependent quantum objects.

Deprecated since version 4.6.0: *eseries* will be removed in QuTiP 5. Please use *QobjEvo* for general time-dependence.

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`.

5.1.4 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 [matplotlib.axes.Axes] User supplied Matplotlib axes for Bloch sphere animation.

fig [matplotlib.figure.Figure] User supplied Matplotlib Figure instance for plotting Bloch sphere.

font_color [str, default 'black'] Color of font used for Bloch sphere labels.

font_size [int, default 20] Size of font used for Bloch sphere labels.

frame_alpha [float, default 0.1] Sets transparency of Bloch sphere frame.

frame_color [str, default 'gray'] Color of sphere wireframe.

frame_width [int, default 1] Width of wireframe.

point_color [list, default ["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, default ["o", "s", "d", "^"]] List of point marker shapes to cycle through.

point_size [list, default [25, 32, 35, 45]] List of point marker sizes. Note, not all point markers look the same size when plotted!

sphere_alpha [float, default 0.2] Transparency of Bloch sphere itself.

sphere_color [str, default '#FFDDDD'] Color of Bloch sphere.

figsize [list, default [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, default 5] Width of displayed vectors.

vector_style [str, default '-|>'] Vector arrowhead style (from matplotlib's arrow style).

vector_mutation [int, default 20] Width of vectors arrowhead.

view [list, default [-60, 30]] Azimuthal and Elevation viewing angles.

xlabel [list, default ["x", ""]] List of strings corresponding to +x and -x axes labels, respectively.

xlpos [list, default [1.1, -1.1]] Positions of +x and -x labels respectively.

ylabel [list, default ["y", ""]] List of strings corresponding to +y and -y axes labels, respectively.

ylpos [list, default [1.2, -1.2]] Positions of +y and -y labels respectively.

zlabel [list, default ['$\left|0\right>$', '$\left|1\right>$']] List of strings corresponding to +z and -z axes labels, respectively.

zlpos [list, default [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] 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_arc(*start*, *end*, *fmt='b'*, *steps=None*, ***kwargs*)
Adds an arc between two points on a sphere. The arc is set to be blue solid curve by default.

The start and end points must be on the same sphere (i.e. have the same radius) but need not be on the unit sphere.

Parameters

start [Qobj or array-like] Array with cartesian coordinates of the first point, or a state vector or density matrix that can be mapped to a point on or within the Bloch sphere.

end [Qobj or array-like] Array with cartesian coordinates of the second point, or a state vector or density matrix that can be mapped to a point on or within the Bloch sphere.

fmt [str, default: "b"] A matplotlib format string for rendering the arc.

steps [int, default: None] The number of segments to use when rendering the arc. The default uses 100 steps times the distance between the start and end points, with a minimum of 2 steps.

****kwargs** [dict] Additional parameters to pass to the matplotlib .plot function when rendering this arc.

add_line(*start*, *end*, *fmt='k'*, ***kwargs*)
Adds a line segment connecting two points on the bloch sphere.

The line segment is set to be a black solid line by default.

Parameters

start [Qobj or array-like] Array with cartesian coordinates of the first point, or a state vector or density matrix that can be mapped to a point on or within the Bloch sphere.

end [Qobj or array-like] Array with cartesian coordinates of the second point, or a state vector or density matrix that can be mapped to a point on or within the Bloch sphere.

fmt [str, default: "k"] A matplotlib format string for rendering the line.

****kwargs** [dict] Additional parameters to pass to the matplotlib .plot function when rendering this line.

add_points(*points*, *meth='s'*, *alpha=1.0*)
Add a list of data points to bloch sphere.

Parameters

points [array_like] Collection of data points.

meth [{'s', 'm', 'l'}] Type of points to plot, use 'm' for multicolored, 'l' for points connected with a line.

alpha [float, default=1.] Transparency value for the vectors. Values between 0 and 1.

.. note:: When using `meth=l` in QuTiP 4.6, the line transparency defaulted to `0.75` and there was no way to alter it. When the `alpha` parameter was added in QuTiP 4.7, the default became `alpha=1.0` for values of `meth`.

add_states(*state*, *kind='vector'*, *alpha=1.0*)
Add a state vector Qobj to Bloch sphere.

Parameters

state [Qobj] Input state vector.

kind [{'vector', 'point'}] Type of object to plot.

alpha [float, default=1.] Transparency value for the vectors. Values between 0 and 1.

add_vectors(*vectors*, *alpha=1.0*)
Add a list of vectors to Bloch sphere.

Parameters

vectors [array_like] Array with vectors of unit length or smaller.

alpha [float, default=1.] Transparency value for the vectors. Values between 0 and 1.

clear()
Resets Bloch sphere data sets to empty.

make_sphere()
Plots Bloch sphere and data sets.

render()
Render the Bloch sphere and its data sets in on given figure and axes.

save(*name=None*, *format='png'*, *dirc=None*, *dpin=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.

dpin [int] Resolution in dots per inch.

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: https://en.wikipedia.org/wiki/Jones_calculus

- "polarization stokes" see also: https://en.wikipedia.org/wiki/Stokes_parameters

show()
Display Bloch sphere and corresponding data sets.

Notes

When using inline plotting in Jupyter notebooks, any figure created in a notebook cell is displayed after the cell executes. Thus if you create a figure yourself and use it create a Bloch sphere with `b = Bloch(..., fig=fig)` and then call `b.show()` in the same cell, then the figure will be displayed twice. If you do create your own figure, the simplest solution to this is to not call `.show()` in the cell you create the figure in.

vector_mutation
Sets the width of the vectors arrowhead

vector_style
Style of Bloch vectors, default = '-|>' (or 'simple')

vector_width
Width of Bloch vectors, default = 5

class Bloch3d(*fig=None*)
Class for plotting data on a 3D Bloch sphere using mayavi. Valid data can be either points, vectors, or qobj objects corresponding to state vectors or density matrices. for a two-state system (or subsystem).

Notes

The use of mayavi for 3D rendering of the Bloch sphere comes with a few limitations: I) You can not embed a Bloch3d figure into a matplotlib window. II) The use of LaTex is not supported by the mayavi rendering engine. Therefore all labels must be defined using standard text. Of course you can post-process the generated figures later to add LaTeX using other software if needed.

Attributes

fig [instance {None}] User supplied Matplotlib Figure instance for plotting Bloch sphere.

font_color [str {'black'}] Color of font used for Bloch sphere labels.

font_scale [float {0.08}] Scale for font used for Bloch sphere labels.

frame [bool {True}] Draw frame for Bloch sphere

frame_alpha [float {0.05}] Sets transparency of Bloch sphere frame.

frame_color [str {'gray'}] Color of sphere wireframe.

frame_num [int {8}] Number of frame elements to draw.

frame_radius [floats {0.005}] Width of wireframe.

point_color [list {['r', 'g', 'b', 'y']}] List of colors for Bloch sphere point markers to cycle through. i.e. By default, points 0 and 4 will both be blue ('r').

point_mode [string {'sphere','cone','cube','cylinder','point'}] Point marker shapes.

point_size [float {0.075}] Size of points on Bloch sphere.

sphere_alpha [float {0.1}] Transparency of Bloch sphere itself.

sphere_color [str {'#808080'}] Color of Bloch sphere.

size [list {[500,500]}] Size of Bloch sphere plot in pixels. Best to have both numbers the same otherwise you will have a Bloch sphere that looks like a football.

vector_color [list {['r', 'g', 'b', 'y']}] List of vector colors to cycle through.

vector_width [int {3}] Width of displayed vectors.

view [list {[45,65]}] Azimuthal and Elevation viewing angles.

xlabel [list {`['|x>', '']`}] List of strings corresponding to +x and -x axes labels, respectively.

xlpos [list {[1.07,-1.07]}] Positions of +x and -x labels respectively.

ylabel [list {`['|y>', '']`}] List of strings corresponding to +y and -y axes labels, respectively.

ylpos [list {[1.07,-1.07]}] Positions of +y and -y labels respectively.

zlabel [list {`['|0>', '|1>']`}] List of strings corresponding to +z and -z axes labels, respectively.

zlpos [list {[1.07,-1.07]}] Positions of +z and -z labels respectively.

add_points(*points*, *meth='s'*, *alpha=1.0*)
Add a list of data points to bloch sphere.

Parameters

points [array/list] Collection of data points.

meth [str {'s','m'}] Type of points to plot, use 'm' for multicolored.

alpha [float, default=1.] Transparency value for the vectors. Values between 0 and 1.

add_states(*state*, *kind='vector'*, *alpha=1.0*)
Add a state vector Qobj to Bloch sphere.

Parameters

state [qobj] Input state vector.

kind [str {'vector','point'}] Type of object to plot.

alpha [float, default=1.] Transparency value for the vectors. Values between 0 and 1.

add_vectors(*vectors*, *alpha=1.0*)
Add a list of vectors to Bloch sphere.

Parameters

vectors [array/list] Array with vectors of unit length or smaller.

alpha [float, default=1.] Transparency value for the vectors. Values between 0 and 1.

clear()
Resets the Bloch sphere data sets to empty.

make_sphere()
Plots Bloch sphere and data sets.

plot_points()
Plots points on the Bloch sphere.

plot_vectors()
Plots vectors on the Bloch sphere.

save(*name=None*, *format='png'*, *dirc=None*)
Saves Bloch sphere to file of type `format` in directory `dirc`.

Parameters

name [str] Name of saved image. Must include path and format as well. i.e. '/Users/Paul/Desktop/bloch.png' This overrides the 'format' and 'dirc' arguments.

format [str] Format of output image. Default is 'png'.

dirc [str] Directory for output images. Defaults to current working directory.

Returns

File containing plot of Bloch sphere.

show()
Display the Bloch sphere and corresponding data sets.

5.1.5 Distributions

class QFunc(*xvec*, *yvec*, *g: float = 1.4142135623730951*, *memory: float = 1024*)
Class-based method of calculating the Husimi-Q function of many different quantum states at fixed phase-space points `0.5*g* (xvec + i*yvec)`. This class has slightly higher first-usage costs than *qfunc*, but subsequent operations will be several times faster. However, it can require quite a lot of memory. Call the created object as a function to retrieve the Husimi-Q function.

Parameters

xvec, yvec [array_like] x- and y-coordinates at which to calculate the Husimi-Q function.

g [float, default sqrt(2)] Scaling factor for `a = 0.5 * g * (x + iy)`. The value of g is related to the value of *hbar* in the commutation relation $[x,\, y] = i\hbar$ via $\hbar = 2/g^2$, so the default corresponds to $\hbar = 1$.

memory [real, default 1024] Size in MB that may be used internally as workspace. This class will raise `MemoryError` if subsequently passed a state of sufficiently large dimension that this bound would be exceeded. In those cases, use *qfunc* with `precompute_memory=None` instead to force using the slower, more memory-efficient algorithm.

See also:

qfunc a single function version, which will involve computing several quantities multiple times in order to use less memory.

Examples

Initialise the class for a square set of coordinates, with some states we want to investigate.

```
>>> xvec = np.linspace(-2, 2, 101)
>>> states = [qutip.rand_dm(10) for _ in [None]*10]
>>> qfunc = qutip.QFunc(xvec, xvec)
```

Now we can calculate the Husimi-Q function over each of the states more efficiently with:

```
>>> husimiq = np.array([qfunc(state) for state in states])
```

5.1.6 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.

5.1.7 Non-Markovian Solvers

class HEOMSolver(*H_sys*, *bath*, *max_depth*, *options=None*, *progress_bar=None*)
HEOM solver that supports multiple baths.

The baths must be all either bosonic or fermionic baths.

Parameters

H_sys [QObj, QobjEvo or a list] The system Hamiltonian or Liouvillian specified as either a *Qobj*, a *QobjEvo*, or a list of elements that may be converted to a *QobjEvo*.

bath [Bath or list of Bath] A *Bath* containing the exponents of the expansion of the bath correlation funcion and their associated coefficients and coupling operators, or a list of baths.

If multiple baths are given, they must all be either fermionic or bosonic baths.

max_depth [int] The maximum depth of the heirarchy (i.e. the maximum number of bath exponent "excitations" to retain).

options [*qutip.solver.Options*] Generic solver options. If set to None the default options will be used.

progress_bar [None, True or `BaseProgressBar`] Optional instance of BaseProgressBar, or a subclass thereof, for showing the progress of the solver. If True, an instance of `TextProgressBar` is used instead.

Attributes

ados [*HierarchyADOs*] The description of the hierarchy constructed from the given bath and maximum depth.

run(*rho0*, *tlist*, *e_ops=None*, *ado_init=False*, *ado_return=False*)
Solve for the time evolution of the system.

Parameters

rho0 [Qobj or HierarchyADOsState or numpy.array] Initial state (*Qobj* density matrix) of the system if `ado_init` is `False`.

If `ado_init` is `True`, then `rho0` should be an instance of *HierarchyADOsState* or a numpy array giving the initial state of all ADOs. Usually the state of the ADOs would be determine from a previous call to `.run(..., ado_return=True)`. For example, `result = solver.run(..., ado_return=True)` could be followed by `solver.run(result.ado_states[-1], tlist, ado_init=True)`.

If a numpy array is passed its shape must be `(number_of_ados, n, n)` where `(n, n)` is the system shape (i.e. shape of the system density matrix) and the ADOs must be in the same order as in `.ados.labels`.

tlist [list] An ordered list of times at which to return the value of the state.

e_ops [Qobj / callable / list / dict / None, optional] A list or dictionary of operators as *Qobj* and/or callable functions (they can be mixed) or a single operator or callable function. For an operator `op`, the result will be computed using `(state * op).tr()` and the state at each time `t`. For callable functions, `f`, the result is computed using `f(t, ado_state)`. The values are stored in `expect` on (see the return section below).

ado_init: bool, default False Indicates if initial condition is just the system state, or a numpy array including all ADOs.

ado_return: bool, default True Whether to also return as output the full state of all ADOs.

Returns

qutip.solver.Result The results of the simulation run, with the following attributes:

- `times`: the times `t` (i.e. the `tlist`).
- `states`: the system state at each time `t` (only available if `e_ops` was `None` or if the solver option `store_states` was set to `True`).
- `ado_states`: the full ADO state at each time (only available if `ado_return` was set to `True`). Each element is an instance of *HierarchyADOsState*. . The state of a particular ADO may be extracted from `result.ado_states[i]` by calling *extract*.
- `expect`: the value of each `e_ops` at time `t` (only available if `e_ops` were given). If `e_ops` was passed as a dictionary, then `expect` will be a dictionary with the same keys as `e_ops` and values giving the list of outcomes for the corresponding key.

steady_state(*use_mkl=True*, *mkl_max_iter_refine=100*, *mkl_weighted_matching=False*)

Compute the steady state of the system.

Parameters

use_mkl [bool, default=False] Whether to use mkl or not. If mkl is not installed or if this is false, use the scipy splu solver instead.

mkl_max_iter_refine [int] Specifies the the maximum number of iterative refinement steps that the MKL PARDISO solver performs.

For a complete description, see iparm(8) in http://cali2.unilim.fr/intel-xe/mkl/mklman/GUID-264E311E-ACED-4D56-AC31-E9D3B11D1CBF.htm.

mkl_weighted_matching [bool] MKL PARDISO can use a maximum weighted matching algorithm to permute large elements close the diagonal. This strategy adds an additional level of reliability to the factorization methods.

For a complete description, see iparm(13) in http://cali2.unilim.fr/intel-xe/mkl/mklman/GUID-264E311E-ACED-4D56-AC31-E9D3B11D1CBF.htm.

Returns

steady_state [Qobj] The steady state density matrix of the system.

steady_ados [*HierarchyADOsState*] The steady state of the full ADO hierarchy. A particular ADO may be extracted from the full state by calling *extract*.

class HSolverDL(*H_sys*, *coup_op*, *coup_strength*, *temperature*, *N_cut*, *N_exp*, *cut_freq*, *bnd_cut_approx=False*, *options=None*, *progress_bar=None*, *combine=True*)

A helper class for creating an *HEOMSolver* that is backwards compatible with the `HSolverDL` provided in `qutip.nonmarkov.heom` in QuTiP 4.6 and below.

See *HEOMSolver* and *DrudeLorentzBath* for more descriptions of the underlying solver and bath construction.

An exact copy of the QuTiP 4.6 HSolverDL is provided in `qutip.nonmarkov.dlheom_solver` for cases where the functionality of the older solver is required. The older solver will be completely removed in QuTiP 5.

Note: Unlike the version of `HSolverDL` in QuTiP 4.6, this solver supports supplying a time-dependent or Liouvillian `H_sys`.

Note: For compatibility with `HSolverDL` in QuTiP 4.6 and below, the parameter `N_exp` specifying the number of exponents to keep in the expansion of the bath correlation function is one more than the equivalent `Nk` used in the *DrudeLorentzBath*. I.e., `Nk = N_exp - 1`. The `Nk` parameter in the *DrudeLorentzBath* does not count the zeroeth exponent in order to better match common usage in the literature.

Note: The `stats` and `renorm` arguments accepted in QuTiP 4.6 and below are no longer supported.

Parameters

H_sys [Qobj or QobjEvo or list] The system Hamiltonian or Liouvillian. See *HEOMSolver* for a complete description.

coup_op [Qobj] Operator describing the coupling between system and bath. See parameter `Q` in *BosonicBath* for a complete description.

coup_strength [float] Coupling strength. Referred to as `lam` in *DrudeLorentzBath*.

temperature [float] Bath temperature. Referred to as `T` in *DrudeLorentzBath*.

N_cut [int] The maximum depth of the hierarchy. See `max_depth` in *HEOMSolver* for a full description.

N_exp [int] Number of exponential terms used to approximate the bath correlation functions. The equivalent `Nk` in *DrudeLorentzBath* is one less than `N_exp` (see note above).

cut_freq [float] Bath spectral density cutoff frequency. Referred to as `gamma` in *DrudeLorentzBath*.

bnd_cut_approx [bool] Use boundary cut off approximation. If true, the Matsubara terminator is added to the system Liouvillian (and H_sys is promoted to a Liouvillian if it was a Hamiltonian).

options [*qutip.solver.Options*] Generic solver options. If set to None the default options will be used.

progress_bar [None, True or `BaseProgressBar`] Optional instance of BaseProgressBar, or a subclass thereof, for showing the progress of the solver. If True, an instance of `TextProgressBar` is used instead.

combine [bool, default True] Whether to combine exponents with the same frequency (and coupling operator). See *BosonicBath.combine* for details.

class BathExponent(*type*, *dim*, *Q*, *ck*, *vk*, *ck2=None*, *sigma_bar_k_offset=None*, *tag=None*)

Represents a single exponent (naively, an excitation mode) within the decomposition of the correlation functions of a bath.

Parameters

type [{"R", "I", "RI", "+", "-"} or BathExponent.ExponentType] The type of bath exponent.

"R" and "I" are bosonic bath exponents that appear in the real and imaginary parts of the correlation expansion.

"RI" is combined bosonic bath exponent that appears in both the real and imaginary parts of the correlation expansion. The combined exponent has a single `vk`. The `ck` is the coefficient in the real expansion and `ck2` is the coefficient in the imaginary expansion.

"+" and "-" are fermionic bath exponents. These fermionic bath exponents must specify `sigma_bar_k_offset` which specifies the amount to add to `k` (the exponent index within the bath of this exponent) to determine the `k` of the corresponding exponent with the opposite sign (i.e. "-" or "+").

dim [int or None] The dimension (i.e. maximum number of excitations for this exponent). Usually `2` for fermionic exponents or `None` (i.e. unlimited) for bosonic exponents.

Q [Qobj] The coupling operator for this excitation mode.

vk [complex] The frequency of the exponent of the excitation term.

ck [complex] The coefficient of the excitation term.

ck2 [optional, complex] For exponents of type "RI" this is the coefficient of the term in the imaginary expansion (and `ck` is the coefficient in the real expansion).

sigma_bar_k_offset [optional, int] For exponents of type "+" this gives the offset (within the list of exponents within the bath) of the corresponding "-" bath exponent. For exponents of type "-" it gives the offset of the corresponding "+" exponent.

tag [optional, str, tuple or any other object] A label for the exponent (often the name of the bath). It defaults to None.

Attributes

All of the parameters are available as attributes.

types
alias of `qutip.nonmarkov.bofin_baths.ExponentType`

class Bath(*exponents*)
Represents a list of bath expansion exponents.

Parameters

exponents [list of BathExponent] The exponents of the correlation function describing the bath.

Attributes

All of the parameters are available as attributes.

class BosonicBath(*Q*, *ck_real*, *vk_real*, *ck_imag*, *vk_imag*, *combine=True*, *tag=None*)
A helper class for constructing a bosonic bath from the expansion coefficients and frequencies for the real and imaginary parts of the bath correlation function.

If the correlation functions `C(t)` is split into real and imaginary parts:

```
C(t) = C_real(t) + i * C_imag(t)
```

then:

```
C_real(t) = sum(ck_real * exp(- vk_real * t))
C_imag(t) = sum(ck_imag * exp(- vk_imag * t))
```

Defines the coefficients `ck` and the frequencies `vk`.

Note that the `ck` and `vk` may be complex, even through `C_real(t)` and `C_imag(t)` (i.e. the sum) is real.

Parameters

Q [Qobj] The coupling operator for the bath.

ck_real [list of complex] The coefficients of the expansion terms for the real part of the correlation function. The corresponding frequencies are passed as vk_real.

vk_real [list of complex] The frequencies (exponents) of the expansion terms for the real part of the correlation function. The corresponding ceofficients are passed as ck_real.

ck_imag [list of complex] The coefficients of the expansion terms in the imaginary part of the correlation function. The corresponding frequencies are passed as vk_imag.

vk_imag [list of complex] The frequencies (exponents) of the expansion terms for the imaginary part of the correlation function. The corresponding ceofficients are passed as ck_imag.

combine [bool, default True] Whether to combine exponents with the same frequency (and coupling operator). See *combine* for details.

tag [optional, str, tuple or any other object] A label for the bath exponents (for example, the name of the bath). It defaults to None but can be set to help identify which bath an exponent is from.

classmethod combine(*exponents*, *rtol=1e-05*, *atol=1e-07*)

Group bosonic exponents with the same frequency and return a single exponent for each frequency present.

Exponents with the same frequency are only combined if they share the same coupling operator `.Q`.

Note that combined exponents take their tag from the first exponent in the group being combined (i.e. the one that occurs first in the given exponents list).

Parameters

exponents [list of BathExponent] The list of exponents to combine.

rtol [float, default 1e-5] The relative tolerance to use to when comparing frequencies and coupling operators.

atol [float, default 1e-7] The absolute tolerance to use to when comparing frequencies and coupling operators.

Returns

list of BathExponent The new reduced list of exponents.

class DrudeLorentzBath(*Q*, *lam*, *gamma*, *T*, *Nk*, *combine=True*, *tag=None*)

A helper class for constructing a Drude-Lorentz bosonic bath from the bath parameters (see parameters below).

Parameters

Q [Qobj] Operator describing the coupling between system and bath.

lam [float] Coupling strength.

gamma [float] Bath spectral density cutoff frequency.

T [float] Bath temperature.

Nk [int] Number of exponential terms used to approximate the bath correlation functions.

combine [bool, default True] Whether to combine exponents with the same frequency (and coupling operator). See *BosonicBath.combine* for details.

tag [optional, str, tuple or any other object] A label for the bath exponents (for example, the name of the bath). It defaults to None but can be set to help identify which bath an exponent is from.

terminator()
Return the Matsubara terminator for the bath and the calculated approximation discrepancy.

Returns

delta: float The approximation discrepancy. That is, the difference between the true correlation function of the Drude-Lorentz bath and the sum of the `Nk` exponential terms is approximately `2 * delta * dirac(t)`, where `dirac(t)` denotes the Dirac delta function.

terminator [Qobj] The Matsubara terminator – i.e. a liouvillian term representing the contribution to the system-bath dynamics of all exponential expansion terms beyond `Nk`. It should be used by adding it to the system liouvillian (i.e. `liouvillian(H_sys)`).

class DrudeLorentzPadeBath(*Q*, *lam*, *gamma*, *T*, *Nk*, *combine=True*, *tag=None*)
A helper class for constructing a Padé expansion for a Drude-Lorentz bosonic bath from the bath parameters (see parameters below).

A Padé approximant is a sum-over-poles expansion (see https://en.wikipedia.org/wiki/Pad%C3%A9_approximant).

The application of the Padé method to spectrum decompoisitions is described in "Padé spectrum decompositions of quantum distribution functions and optimal hierarchical equations of motion construction for quantum open systems" [1].

The implementation here follows the approach in the paper.

[1] J. Chem. Phys. 134, 244106 (2011); https://doi.org/10.1063/1.3602466

This is an alternative to the *DrudeLorentzBath* which constructs a simpler exponential expansion.

Parameters

Q [Qobj] Operator describing the coupling between system and bath.

lam [float] Coupling strength.

gamma [float] Bath spectral density cutoff frequency.

T [float] Bath temperature.

Nk [int] Number of Padé exponentials terms used to approximate the bath correlation functions.

combine [bool, default True] Whether to combine exponents with the same frequency (and coupling operator). See *BosonicBath.combine* for details.

tag [optional, str, tuple or any other object] A label for the bath exponents (for example, the name of the bath). It defaults to None but can be set to help identify which bath an exponent is from.

terminator()
Return the Padé terminator for the bath and the calculated approximation discrepancy.

Returns

delta: float The approximation discrepancy. That is, the difference between the true correlation function of the Drude-Lorentz bath and the sum of the `Nk` exponential terms is approximately `2 * delta * dirac(t)`, where `dirac(t)` denotes the Dirac delta function.

terminator [Qobj] The Padé terminator – i.e. a liouvillian term representing the contribution to the system-bath dynamics of all exponential expansion terms beyond `Nk`. It should be used by adding it to the system liouvillian (i.e. `liouvillian(H_sys)`).

class UnderDampedBath(*Q*, *lam*, *gamma*, *w0*, *T*, *Nk*, *combine=True*, *tag=None*)

A helper class for constructing an under-damped bosonic bath from the bath parameters (see parameters below).

Parameters

Q [Qobj] Operator describing the coupling between system and bath.

lam [float] Coupling strength.

gamma [float] Bath spectral density cutoff frequency.

w0 [float] Bath spectral density resonance frequency.

T [float] Bath temperature.

Nk [int] Number of exponential terms used to approximate the bath correlation functions.

combine [bool, default True] Whether to combine exponents with the same frequency (and coupling operator). See *BosonicBath.combine* for details.

tag [optional, str, tuple or any other object] A label for the bath exponents (for example, the name of the bath). It defaults to None but can be set to help identify which bath an exponent is from.

class FermionicBath(*Q*, *ck_plus*, *vk_plus*, *ck_minus*, *vk_minus*, *tag=None*)

A helper class for constructing a fermionic bath from the expansion coefficients and frequencies for the + and − modes of the bath correlation function.

There must be the same number of + and − modes and their coefficients must be specified in the same order so that `ck_plus[i], vk_plus[i]` are the plus coefficient and frequency corresponding to the minus mode `ck_minus[i], vk_minus[i]`.

In the fermionic case the order in which excitations are created or destroyed is important, resulting in two different correlation functions labelled `C_plus(t)` and `C_plus(t)`:

```
C_plus(t) = sum(ck_plus * exp(- vk_plus * t))
C_minus(t) = sum(ck_minus * exp(- vk_minus * t))
```

where the expansions above define the coeffiients `ck` and the frequencies `vk`.

Parameters

Q [Qobj] The coupling operator for the bath.

ck_plus [list of complex] The coefficients of the expansion terms for the + part of the correlation function. The corresponding frequencies are passed as vk_plus.

vk_plus [list of complex] The frequencies (exponents) of the expansion terms for the + part of the correlation function. The corresponding ceofficients are passed as ck_plus.

ck_minus [list of complex] The coefficients of the expansion terms for the − part of the correlation function. The corresponding frequencies are passed as vk_minus.

vk_minus [list of complex] The frequencies (exponents) of the expansion terms for the − part of the correlation function. The corresponding ceofficients are passed as ck_minus.

tag [optional, str, tuple or any other object] A label for the bath exponents (for example, the name of the bath). It defaults to None but can be set to help identify which bath an exponent is from.

class LorentzianBath(*Q*, *gamma*, *w*, *mu*, *T*, *Nk*, *tag=None*)

A helper class for constructing a Lorentzian fermionic bath from the bath parameters (see parameters below).

Note: This Matsubara expansion used in this bath converges very slowly and `Nk > 20` may be required to get good convergence. The Padé expansion used by *LorentzianPadeBath* converges much more quickly.

Parameters

Q [Qobj] Operator describing the coupling between system and bath.

gamma [float] The coupling strength between the system and the bath.

w [float] The width of the environment.

mu [float] The chemical potential of the bath.

T [float] Bath temperature.

Nk [int] Number of exponential terms used to approximate the bath correlation functions.

tag [optional, str, tuple or any other object] A label for the bath exponents (for example, the name of the bath). It defaults to None but can be set to help identify which bath an exponent is from.

class LorentzianPadeBath(*Q*, *gamma*, *w*, *mu*, *T*, *Nk*, *tag=None*)

A helper class for constructing a Padé expansion for Lorentzian fermionic bath from the bath parameters (see parameters below).

A Padé approximant is a sum-over-poles expansion (see https://en.wikipedia.org/wiki/Pad%C3%A9_approximant).

The application of the Padé method to spectrum decompoisitions is described in "Padé spectrum decompositions of quantum distribution functions and optimal hierarchical equations of motion construction for quantum open systems" [1].

The implementation here follows the approach in the paper.

[1] J. Chem. Phys. 134, 244106 (2011); https://doi.org/10.1063/1.3602466

This is an alternative to the *LorentzianBath* which constructs a simpler exponential expansion that converges much more slowly in this particular case.

Parameters

Q [Qobj] Operator describing the coupling between system and bath.

gamma [float] The coupling strength between the system and the bath.

w [float] The width of the environment.

mu [float] The chemical potential of the bath.

T [float] Bath temperature.

Nk [int] Number of exponential terms used to approximate the bath correlation functions.

tag [optional, str, tuple or any other object] A label for the bath exponents (for example, the name of the bath). It defaults to None but can be set to help identify which bath an exponent is from.

class HierarchyADOs(*exponents*, *max_depth*)

A description of ADOs (auxilliary density operators) with the hierarchical equations of motion.

The list of ADOs is constructed from a list of bath exponents (corresponding to one or more baths). Each ADO is referred to by a label that lists the number of "excitations" of each bath exponent. The level of a label within the hierarchy is the sum of the "excitations" within the label.

For example the label `(0, 0, ..., 0)` represents the density matrix of the system being solved and is the only 0th level label.

The labels with a single 1, i.e. `(1, 0, ..., 0)`, `(0, 1, 0, ... 0)`, etc. are the 1st level labels.

The second level labels all have either two 1s or a single 2, and so on for the third and higher levels of the hierarchy.

Parameters

exponents [list of BathExponent] The exponents of the correlation function describing the bath or baths.

max_depth [int] The maximum depth of the hierarchy (i.e. the maximum sum of "excitations" in the hierarchy ADO labels or maximum ADO level).

Attributes

exponents [list of BathExponent] The exponents of the correlation function describing the bath or baths.

max_depth [int] The maximum depth of the hierarchy (i.e. the maximum sum of "excitations" in the hierarchy ADO labels).

dims [list of int] The dimensions of each exponent within the bath(s).

vk [list of complex] The frequency of each exponent within the bath(s).

ck [list of complex] The coefficient of each exponent within the bath(s).

ck2: list of complex For exponents of type "RI", the coefficient of the exponent within the imaginary expansion. For other exponent types, the entry is None.

sigma_bar_k_offset: list of int For exponents of type "+" or "-" the offset within the list of modes of the corresponding "-" or "+" exponent. For other exponent types, the entry is None.

labels: list of tuples A list of the ADO labels within the hierarchy.

exps(*label*)

Converts an ADO label into a tuple of exponents, with one exponent for each "excitation" within the label.

The number of exponents returned is always equal to the level of the label within the hierarchy (i.e. the sum of the indices within the label).

Parameters

label [tuple] The ADO label to convert to a list of exponents.

Returns

tuple of BathExponent A tuple of BathExponents.

Examples

`ados.exps((1, 0, 0))` would return `[ados.exponents[0]]`

`ados.exps((2, 0, 0))` would return `[ados.exponents[0], ados.exponents[0]]`.

`ados.exps((1, 2, 1))` would return `[ados.exponents[0], ados.exponents[1], ados.exponents[1], ados.exponents[2]]`.

filter(*level=None*, *tags=None*, *dims=None*, *types=None*)

Return a list of ADO labels for ADOs whose "excitations" match the given patterns.

Each of the filter parameters (tags, dims, types) may be either unspecified (None) or a list. Unspecified parameters are excluded from the filtering.

All specified filter parameters must be lists of the same length. Each position in the lists describes a particular excitation and any exponent that matches the filters may supply that excitation. The level of all labels returned is thus equal to the length of the filter parameter lists.

Within a filter parameter list, items that are None represent wildcards and match any value of that exponent attribute

Parameters

level [int] The hierarchy depth to return ADOs from.

tags [list of object or None] Filter parameter that matches the `.tag` attribute of exponents.

dims [list of int] Filter parameter that matches the `.dim` attribute of exponents.

types [list of BathExponent types or list of str] Filter parameter that matches the `.type` attribute of exponents. Types may be supplied by name (e.g. "R", "I", "+") instead of by the actual type (e.g. `BathExponent.types.R`).

Returns

list of tuple The ADO label for each ADO whose exponent excitations (i.e. label) match the given filters or level.

idx(*label*)
Return the index of the ADO label within the list of labels, i.e. within `self.labels`.

Parameters

label [tuple] The label to look up.

Returns

int The index of the label within the list of ADO labels.

next(*label*, *k*)
Return the ADO label with one more excitation in the k'th exponent dimension or `None` if adding the excitation would exceed the dimension or maximum depth of the hierarchy.

Parameters

label [tuple] The ADO label to add an excitation to.

k [int] The exponent to add the excitation to.

Returns

tuple or None The next label.

prev(*label*, *k*)
Return the ADO label with one fewer excitation in the k'th exponent dimension or `None` if the label has no exciations in the k'th exponent.

Parameters

label [tuple] The ADO label to remove the excitation from.

k [int] The exponent to remove the excitation from.

Returns

tuple or None The previous label.

class HierarchyADOsState(*rho*, *ados*, *ado_state*)
Provides convenient access to the full hierarchy ADO state at a particular point in time, `t`.

Parameters

rho [*Qobj*] The current state of the system (i.e. the 0th component of the hierarchy).

ados [*HierarchyADOs*] The description of the hierarchy.

ado_state [numpy.array] The full state of the hierarchy.

Attributes

rho [Qobj] The system state.

In addition, all of the attributes of the hierarchy description,

i.e. ``HierarchyADOs``, are provided directly on this class for

convenience. E.g. one can access ``.labels``, or ``.exponents`` or

call ``.idx(label)`` directly.

See :class:`HierarchyADOs` for a full list of the available attributes

and methods.

extract(*idx_or_label*)

Extract a Qobj representing specified ADO from a full representation of the ADO states.

Parameters

idx [int or label] The index of the ADO to extract. If an ADO label, e.g. `(0, 1, 0, ...)` is supplied instead, then the ADO is extracted by label instead.

Returns

Qobj A *Qobj* representing the state of the specified ADO.

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 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: BaseProgressBar 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 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.

integrator [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

dynmaps [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 $\rho(t_n)$

learningtimes [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.

5.1.8 Solver Options and Results

class ExpectOps(*e_ops=[]*, *super_=False*)
Contain and compute expectation values

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_t_tol=1e-06*, *norm_steps=5*, *rhs_reuse=False*, *rhs_filename=None*, *ntraj=500*, *gui=False*, *rhs_with_state=False*, *store_final_state=False*, *store_states=False*, *steady_state_average=False*, *seeds=None*, *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.

average_states [bool {False}] Average states values over trajectories in stochastic solvers.

average_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 SolverConfiguration

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

sections [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

total_time [float] Time in seconds for the solver to complete processing Can be None, meaning that total timing percentages will be reported

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 `_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 [`_StatsSection`] 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(*me*, *H=None*, *c_ops=[]*, *sc_ops=[]*, *state0=None*, *e_ops=[]*, *m_ops=None*, *store_all_expect=False*, *store_measurement=False*, *dW_factors=None*, *solver=None*, *method='homodyne'*, *normalize=None*, *times=None*, *nsubsteps=1*, *ntraj=1*, *tol=None*, *generate_noise=None*, *noise=None*, *progress_bar=None*, *map_func=None*, *map_kwargs=None*, *args={}*, *options=None*, *noiseDepth=20*)

Class of options for stochastic solvers such as *qutip.stochastic.ssesolve*, *qutip.stochastic.smesolve*, etc.

The stochastic solvers *qutip.stochastic.general_stochastic*, *qutip.stochastic.ssesolve*, *qutip.stochastic.smesolve*, *qutip.stochastic.photocurrent_sesolve* and *qutip.stochastic.photocurrent_mesolve* 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.

Within the attribute list, a `time_dependent_object` is either

- *Qobj*: a constant term
- 2-element list of `[Qobj, time_dependence]`: a time-dependent term where the `Qobj` will be multiplied by the time-dependent scalar.

For more details on all allowed time-dependent objects, see the documentation for *QobjEvo*.

Attributes

H [time_dependent_object or list of time_dependent_object] System Hamiltonian in standard time-dependent list format. This is the same as the argument that (e.g.) *mesolve* takes. If this is a list of elements, they are summed.

state0 [*qutip.Qobj*] Initial state vector (ket) or density matrix.

times [array_like of float] List of times for t. Must be uniformly spaced.

c_ops [list of time_dependent_object] List of deterministic collapse operators. Each element of the list is a separate operator; unlike the Hamiltonian, there is no implicit summation over the terms.

sc_ops [list of time_dependent_object] 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. Each element of the list is a separate operator, like `c_ops`.

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] Dictionary of parameters for time dependent systems.

tol [float] Tolerance of the solver for implicit methods.

ntraj [int] Number of trajectors.

nsubsteps [int] Number of sub steps between each time-spep given in *times*.

dW_factors [array] Array of length len(sc_ops), containing scaling factors for each measurement operator in m_ops.

solver [string] Name of the solver method to use for solving the stochastic equations. Valid values are:

- order 1/2 algorithms: ‘euler-maruyama’, ‘pc-euler’, ‘pc-euler-imp’
- order 1 algorithms: ‘milstein’, ‘platen’, ‘milstein-imp’, ‘rouchon’

- order 3/2 algorithms: 'taylor1.5', 'taylor1.5-imp', 'explicit1.5'
- order 2 algorithms: 'taylor2.0'

See the documentation of *stochastic_solvers* for a description of the solvers. Implicit methods can adjust tolerance via the kw 'tol'. Default is {'tol': 1e-6}

method [string ('homodyne', 'heterodyne')] The name of the type of measurement process that give rise to the stochastic equation to solve.

store_all_expect [bool (default False)] Whether or not to store the e_ops expect values for all paths.

store_measurement [bool (default False)] Whether or not to store the measurement results in the *qutip.solver.Result* instance returned by the solver.

noise [int, or 1D array of int, or 4D array of float]

- int : seed of the noise
- 1D array : length = ntraj, seeds for each trajectories.
- 4D array : `(ntraj, len(times), nsubsteps, len(sc_ops)*[1|2])`. Vector for the noise, the len of the last dimensions is doubled for solvers of order 1.5. This corresponds to results.noise.

noiseDepth [int] Number of terms kept of the truncated series used to create the noise used by taylor2.0 solver.

normalize [bool] (default True for (photo)ssesolve, False for (photo)smesolve) Whether or not to normalize the wave function during the evolution. Normalizing density matrices introduce numerical errors.

options [*qutip.solver.Options*] Generic solver options. Only options.average_states and options.store_states are used.

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.

5.1.9 Permutational Invariance

class Dicke(*N*, *hamiltonian=None*, *emission=0.0*, *dephasing=0.0*, *pumping=0.0*, *collective_emission=0.0*, *collective_dephasing=0.0*, *collective_pumping=0.0*)

The Dicke class which builds the Lindbladian and Liouvillian matrix.

Parameters

N: int The number of two-level systems.

hamiltonian [*qutip.Qobj*] A Hamiltonian in the Dicke basis.

The matrix dimensions are (nds, nds), with nds being the number of Dicke states. The Hamiltonian can be built with the operators given by the *jspin* functions.

emission: float Incoherent emission coefficient (also nonradiative emission). default: 0.0

dephasing: float Local dephasing coefficient. default: 0.0

pumping: float Incoherent pumping coefficient. default: 0.0

collective_emission: float Collective (superradiant) emmission coefficient. default: 0.0

collective_pumping: float Collective pumping coefficient. default: 0.0

collective_dephasing: float Collective dephasing coefficient. default: 0.0

Examples

```
>>> from piqs import Dicke, jspin
>>> N = 2
>>> jx, jy, jz = jspin(N)
>>> jp = jspin(N, "+")
>>> jm = jspin(N, "-")
>>> ensemble = Dicke(N, emission=1.)
>>> L = ensemble.liouvillian()
```

Attributes

N: int The number of two-level systems.

hamiltonian [*qutip.Qobj*] A Hamiltonian in the Dicke basis.

The matrix dimensions are (nds, nds), with nds being the number of Dicke states. The Hamiltonian can be built with the operators given by the *jspin* function in the "dicke" basis.

emission: float Incoherent emission coefficient (also nonradiative emission). default: 0.0

dephasing: float Local dephasing coefficient. default: 0.0

pumping: float Incoherent pumping coefficient. default: 0.0

collective_emission: float Collective (superradiant) emmission coefficient. default: 0.0

collective_dephasing: float Collective dephasing coefficient. default: 0.0

collective_pumping: float Collective pumping coefficient. default: 0.0

nds: int The number of Dicke states.

dshape: tuple The shape of the Hilbert space in the Dicke or uncoupled basis. default: (nds, nds).

c_ops()
Build collapse operators in the full Hilbert space 2^N.

Returns

c_ops_list: list The list with the collapse operators in the 2^N Hilbert space.

coefficient_matrix()
Build coefficient matrix for ODE for a diagonal problem.

Returns

M: ndarray The matrix M of the coefficients for the ODE dp/dt = Mp. p is the vector of the diagonal matrix elements of the density matrix rho in the Dicke basis.

lindbladian()
Build the Lindbladian superoperator of the dissipative dynamics.

Returns

lindbladian [*qutip.Qobj*] The Lindbladian matrix as a *qutip.Qobj*.

liouvillian()
Build the total Liouvillian using the Dicke basis.

Returns

liouv [*qutip.Qobj*] The Liouvillian matrix for the system.

pisolve(*initial_state*, *tlist*, *options=None*)
Solve for diagonal Hamiltonians and initial states faster.

Parameters

initial_state [*qutip.Qobj*] An initial state specified as a density matrix of *qutip.Qbj* type.

tlist: ndarray A 1D numpy array of list of timesteps to integrate

options [*qutip.solver.Options*] The options for the solver.

Returns

result: list A dictionary of the type *qutip.solver.Result* which holds the results of the evolution.

class Pim(*N*, *emission=0.0*, *dephasing=0*, *pumping=0*, *collective_emission=0*, *collective_pumping=0*, *collective_dephasing=0*)

The Permutation Invariant Matrix class.

Initialize the class with the parameters for generating a Permutation Invariant matrix which evolves a given diagonal initial state *p* as:

dp/dt = Mp

Parameters

N: int The number of two-level systems.

emission: float Incoherent emission coefficient (also nonradiative emission). default: 0.0

dephasing: float Local dephasing coefficient. default: 0.0

pumping: float Incoherent pumping coefficient. default: 0.0

collective_emission: float Collective (superradiant) emmission coefficient. default: 0.0

collective_pumping: float Collective pumping coefficient. default: 0.0

collective_dephasing: float Collective dephasing coefficient. default: 0.0

Attributes

N: int The number of two-level systems.

emission: float Incoherent emission coefficient (also nonradiative emission). default: 0.0

dephasing: float Local dephasing coefficient. default: 0.0

pumping: float Incoherent pumping coefficient. default: 0.0

collective_emission: float Collective (superradiant) emmission coefficient. default: 0.0

collective_dephasing: float Collective dephasing coefficient. default: 0.0

collective_pumping: float Collective pumping coefficient. default: 0.0

M: dict A nested dictionary of the structure {row: {col: val}} which holds non zero elements of the matrix M

calculate_j_m(*dicke_row*, *dicke_col*)

Get the value of j and m for the particular Dicke space element.

Parameters

dicke_row, dicke_col: int The row and column from the Dicke space matrix

Returns

j, m: float The j and m values.

calculate_k(*dicke_row*, *dicke_col*)

Get k value from the current row and column element in the Dicke space.

Parameters

dicke_row, dicke_col: int The row and column from the Dicke space matrix.

Returns

——-

k: int The row index for the matrix M for given Dicke space element.

coefficient_matrix()
Generate the matrix M governing the dynamics for diagonal cases.

If the initial density matrix and the Hamiltonian is diagonal, the evolution of the system is given by the simple ODE: dp/dt = Mp.

isdicke(*dicke_row*, *dicke_col*)
Check if an element in a matrix is a valid element in the Dicke space. Dicke row: j value index. Dicke column: m value index. The function returns True if the element exists in the Dicke space and False otherwise.

Parameters

dicke_row, dicke_col [int] Index of the element in Dicke space which needs to be checked

solve(*rho0*, *tlist*, *options=None*)
Solve the ODE for the evolution of diagonal states and Hamiltonians.

tau1(*j*, *m*)
Calculate coefficient matrix element relative to (j, m, m).

tau2(*j*, *m*)
Calculate coefficient matrix element relative to (j, m+1, m+1).

tau3(*j*, *m*)
Calculate coefficient matrix element relative to (j+1, m+1, m+1).

tau4(*j*, *m*)
Calculate coefficient matrix element relative to (j-1, m+1, m+1).

tau5(*j*, *m*)
Calculate coefficient matrix element relative to (j+1, m, m).

tau6(*j*, *m*)
Calculate coefficient matrix element relative to (j-1, m, m).

tau7(*j*, *m*)
Calculate coefficient matrix element relative to (j+1, m-1, m-1).

tau8(*j*, *m*)
Calculate coefficient matrix element relative to (j, m-1, m-1).

tau9(*j*, *m*)
Calculate coefficient matrix element relative to (j-1, m-1, m-1).

tau_valid(*dicke_row*, *dicke_col*)
Find the Tau functions which are valid for this value of (dicke_row, dicke_col) given the number of TLS. This calculates the valid tau values and reurns a dictionary specifying the tau function name and the value.

Parameters

dicke_row, dicke_col [int] Index of the element in Dicke space which needs to be checked.

Returns

taus: dict A dictionary of key, val as {tau: value} consisting of the valid taus for this row and column of the Dicke space element.

5.1.10 One-Dimensional Lattice

class Lattice1d(*num_cell=10*, *boundary='periodic'*, *cell_num_site=1*, *cell_site_dof=[1]*, *Hamiltonian_of_cell=None*, *inter_hop=None*)

A class for representing a 1d crystal.

The Lattice1d class can be defined with any specific unit cells and a specified number of unit cells in the crystal. It can return dispersion relationship, position operators, Hamiltonian in the position represention etc.

Parameters

num_cell [int] The number of cells in the crystal.

boundary [str] Specification of the type of boundary the crystal is defined with.

cell_num_site [int] The number of sites in the unit cell.

cell_site_dof [list of int/ int] The tensor structure of the degrees of freedom at each site of a unit cell.

Hamiltonian_of_cell [qutip.Qobj] The Hamiltonian of the unit cell.

inter_hop [qutip.Qobj / list of Qobj] The coupling between the unit cell at i and at (i+unit vector)

Attributes

num_cell [int] The number of unit cells in the crystal.

cell_num_site [int] The nuber of sites in a unit cell.

length_for_site [int] The length of the dimension per site of a unit cell.

cell_tensor_config [list of int] The tensor structure of the cell in the form [cell_num_site,cell_site_dof[:][0]]

lattice_tensor_config [list of int] The tensor structure of the crystal in the form [num_cell,cell_num_site,cell_site_dof[:][0]]

length_of_unit_cell [int] The length of the dimension for a unit cell.

period_bnd_cond_x [int] 1 indicates "periodic" and 0 indicates "hardwall" boundary condition

inter_vec_list [list of list] The list of list of coefficients of inter unitcell vectors' components along Cartesian uit vectors.

lattice_vectors_list [list of list] The list of list of coefficients of lattice basis vectors' components along Cartesian unit vectors.

H_intra [qutip.Qobj] The Qobj storing the Hamiltonian of the unnit cell.

H_inter_list [list of Qobj/ qutip.Qobj] The list of coupling terms between unit cells of the lattice.

is_real [bool] Indicates if the Hamiltonian is real or not.

Hamiltonian()

Returns the lattice Hamiltonian for the instance of Lattice1d.

Returns

Qobj(Hamil) [qutip.Qobj] oper type Quantum object representing the lattice Hamiltonian.

basis(*cell*, *site*, *dof_ind*)

Returns a single particle wavefunction ket with the particle localized at a specified dof at a specified site of a specified cell.

Parameters

- **cell** (*int*) – The cell at which the particle is to be localized.
- **site** (*int*) – The site of the cell at which the particle is to be localized.
- **dof_ind** (*int/ list of int*) – The index of the degrees of freedom with which the sigle particle is to be localized.

Returns

vec_i [qutip.Qobj] ket type Quantum object representing the localized particle.

bloch_wave_functions()

Returns eigenvectors ($psi_n(k)$) of the Hamiltonian in a numpy.ndarray for translationally symmetric lattices with periodic boundary condition.

$$|\psi_n(k)\rangle = |k\rangle \otimes |u_n(k)\rangle \tag{5.1}$$
$$|u_n(k)\rangle = a_n(k)|a\rangle + b_n(k)|b\rangle \tag{5.2}$$
$$\tag{5.3}$$

Please see section 1.2 of Asbóth, J. K., Oroszlány, L., & Pályi, A. (2016). A short course on topological insulators. Lecture notes in physics, 919 for a review.

Returns

eigenstates [ordered np.array] eigenstates[j][0] is the jth eigenvalue. eigenstates[j][1] is the corresponding eigenvector.

bulk_Hamiltonians()

Returns the bulk momentum space Hamiltonian ($H(k)$) for the lattice at the good quantum numbers of k in a numpy ndarray of Qobj's.

Please see section 1.2 of Asbóth, J. K., Oroszlány, L., & Pályi, A. (2016). A short course on topological insulators. Lecture notes in physics, 919 for a review.

Returns

knxa [np.array] knxA[j][0] is the jth good Quantum number k.

qH_ks [np.ndarray of Qobj's] qH_ks[j] is the Oobj of type oper that holds a bulk Hamiltonian for a good quantum number k.

cell_periodic_parts()

Returns eigenvectors of the bulk Hamiltonian, i.e. the cell periodic part($u_n(k)$) of the Bloch wavefunctios in a numpy.ndarray for translationally symmetric lattices with periodic boundary condition.

$$|\psi_n(k)\rangle = |k\rangle \otimes |u_n(k)\rangle \tag{5.4}$$
$$|u_n(k)\rangle = a_n(k)|a\rangle + b_n(k)|b\rangle \tag{5.5}$$
$$\tag{5.6}$$

Please see section 1.2 of Asbóth, J. K., Oroszlány, L., & Pályi, A. (2016). A short course on topological insulators. Lecture notes in physics, 919 for a review.

Returns

knxa [np.array] knxA[j][0] is the jth good Quantum number k.

vec_kns [np.ndarray of Qobj's] vec_kns[j] is the Oobj of type ket that holds an eigenvector of the bulk Hamiltonian of the lattice.

display_lattice()

Produces a graphic portraying the lattice symbolically with a unit cell marked in it.

Returns

inter_T [Qobj] The coefficient of $psi_{i,N}^{dagger}psi_{0,i+1}$, i.e. the coupling between the two boundary sites of the two unit cells i and i+1.

display_unit_cell(*label_on=False*)
Produces a graphic displaying the unit cell features with labels on if defined by user. Also returns a dict of Qobj's corresponding to the labeled elements on the display.

Returns

Hcell [dict] Hcell[i][j] is the Hamiltonian segment for $H_{i,j}$ labeled on the graphic.

distribute_operator(*op*)
A function that returns an operator matrix that applies op to all the cells in the 1d lattice

Parameters **op** (qutip.Qobj) – Qobj representing the operator to be applied at all cells.

Returns

op_H [qutip.Qobj] Quantum object representing the operator with op applied at all cells.

get_dispersion(*knpoints=0*)
Returns dispersion relationship for the lattice with the specified number of unit cells with a k array and a band energy array.

Returns

knxa [np.array] knxA[j][0] is the jth good Quantum number k.

val_kns [np.array] val_kns[j][:] is the array of band energies of the jth band good at all the good Quantum numbers of k.

k()
Returns the crystal momentum operator. All degrees of freedom has the cell number at their correspondig entry in the position operator.

Returns

Qobj(ks) [qutip.Qobj] The crystal momentum operator in units of 1/a. L is the number of unit cells, a is the length of a unit cell which is always taken to be 1.

operator_at_cells(*op*, *cells*)
A function that returns an operator matrix that applies op to specific cells specified in the cells list

Parameters

op [qutip.Qobj] Qobj representing the operator to be applied at certain cells.

cells: list of int The cells at which the operator op is to be applied.

Returns

Qobj(op_H) [Qobj] Quantum object representing the operator with op applied at the specified cells.

operator_between_cells(*op*, *row_cell*, *col_cell*)
A function that returns an operator matrix that applies op to specific cells specified in the cells list

Parameters

op [qutip.Qobj] Qobj representing the operator to be put between cells row_cell and col_cell.

row_cell: int The row index for cell for the operator op to be applied.

col_cell: int The column index for cell for the operator op to be applied.

Returns

oper_bet_cell [Qobj] Quantum object representing the operator with op applied between the specified cells.

plot_dispersion()
Plots the dispersion relationship for the lattice with the specified number of unit cells. The dispersion of the infinte crystal is also plotted if num_cell is smaller than MAXc.

winding_number()
Returns the winding number for a lattice that has chiral symmetry and also plots the trajectory of (dx,dy)(dx,dy are the coefficients of sigmax and sigmay in the Hamiltonian respectively) on a plane.

Returns

winding_number [int or str] knxA[j][0] is the jth good Quantum number k.

x()
Returns the position operator. All degrees of freedom has the cell number at their correspondig entry in the position operator.

Returns

Qobj(xs) [qutip.Qobj] The position operator.

5.1.11 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)

5.1.12 Quantum information processing

class Gate(**args*, ***kwargs*)
Representation of a quantum gate, with its required parameters, and target and control qubits.

Parameters

name [string] Gate name.

targets [list or int] Gate targets.

controls [list or int] Gate controls.

arg_value [float] Argument value(phi).

arg_label [string] Label for gate representation.

classical_controls [int or list of int, optional] indices of classical bits to control gate on.

control_value [int, optional] value of classical bits to control on, the classical controls are interpreted as an integer with lowest bit being the first one. If not specified, then the value is interpreted to be 2 ** len(classical_controls) - 1 (i.e. all classical controls are 1).

class Measurement(*name*, *targets=None*, *index=None*, *classical_store=None*)

Representation of a quantum measurement, with its required parameters, and target qubits.

Parameters

name [string] Measurement name.

targets [list or int] Gate targets.

classical_store [int] Result of the measurment is stored in this classical register of the circuit.

measurement_comp_basis(*state*)

Measures a particular qubit (determined by the target) whose ket vector/ density matrix is specified in the computational basis and returns collapsed_states and probabilities (retains full dimension).

Parameters

state [ket or oper] state to be measured on specified by ket vector or density matrix

Returns

collapsed_states [List of Qobjs] the collapsed state obtained after measuring the qubits and obtaining the qubit specified by the target in the state specified by the index.

probabilities [List of floats] the probability of measuring a state in a the state specified by the index.

class QubitCircuit(*N*, *input_states=None*, *output_states=None*, *reverse_states=True*, *user_gates=None*, *dims=None*, *num_cbits=0*)

Representation of a quantum program/algorithm, maintaining a sequence of gates.

Parameters

N [int] Number of qubits in the system.

user_gates [dict] Define a dictionary of the custom gates. See examples for detail.

input_states [list] A list of string such as *0*,'+', "A", "Y". Only used for latex.

dims [list] A list of integer for the dimension of each composite system. e.g [2,2,2,2,2] for 5 qubits system. If None, qubits system will be the default option.

num_cbits [int] Number of classical bits in the system.

Examples

```
>>> def user_gate():
...     mat = np.array([[1.,   0],
...                     [0., 1.j]])
...     return Qobj(mat, dims=[[2], [2]])
>>> qubit_circuit = QubitCircuit(2, user_gates={"T":user_gate})
>>> qubit_circuit.add_gate("T", targets=[0])
```

add_1q_gate(*name*, *start=0*, *end=None*, *qubits=None*, *arg_value=None*, *arg_label=None*, *classical_controls=None*, *control_value=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 [int] Starting location of qubits.

end [int] 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*, *overwrite_user_gates=False*)
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 [int] The qubit on which the first gate is applied.

add_gate(*gate*, *targets=None*, *controls=None*, *arg_value=None*, *arg_label=None*, *index=None*, *classical_controls=None*, *control_value=None*)
Adds a gate with specified parameters to the circuit.

Parameters

gate: string or :class:`.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.

index [list] Positions to add the gate. Each index in the supplied list refers to a position in the original list of gates.

classical_controls [int or list of int, optional] indices of classical bits to control gate on.

control_value [int, optional] value of classical bits to control on, the classical controls are interpreted as an integer with lowest bit being the first one. If not specified, then the value is interpreted to be 2 ** len(classical_controls) - 1 (i.e. all classical controls are 1).

add_measurement(*measurement*, *targets=None*, *index=None*, *classical_store=None*)
Adds a measurement with specified parameters to the circuit.

Parameters

measurement: string Measurement name. If name is an instance of *Measuremnent*, parameters are unpacked and added.

targets: list Gate targets

index [list] Positions to add the gate.

classical_store [int] Classical register where result of measurement is stored.

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

qubit_circuit: ***QubitCircuit*** Return *QubitCircuit* of the gates for the qubit circuit with the resolved non-adjacent gates.

propagators(*expand=True*)
Propagator matrix calculator for N qubits returning the individual steps as unitary matrices operating from left to right. :returns: **U_list** – Return list of unitary matrices for the qubit circuit. :rtype: list

propagators_no_expand()
Propagator matrix calculator for N qubits returning the individual steps as unitary matrices operating from left to right.

Returns

U_list [list] Return list of unitary matrices for the qubit circuit.

remove_gate_or_measurement(*index=None*, *end=None*, *name=None*, *remove='first'*)
Remove a gate from a specific index or between two indexes or the first, last or all instances of a particular gate.

Parameters

index [int] Location of gate or measurement to be removed.

name [string] Gate or Measurement name to be removed.

remove [string] If first or all gates/measurements 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. Calls '_resolve_to_universal' for each gate, this function maps each 'GATENAME' with its corresponding '_gate_basis_2q' Subsequently calls _resolve_2q_basis for each basis, this function maps each '2QGATENAME' with its corresponding '_basis_'

Parameters

basis [list.] Basis of the resolved circuit.

Returns

qc [*QubitCircuit*] Return *QubitCircuit* of resolved gates for the qubit circuit in the desired basis.

reverse_circuit()
Reverse an entire circuit of unitary gates.

Returns

qubit_circuit [*QubitCircuit*] Return *QubitCircuit* of resolved gates for the qubit circuit in the reverse order.

run(*state*, *cbits=None*, *U_list=None*, *measure_results=None*, *precompute_unitary=False*)
Calculate the result of one instance of circuit run.

Parameters

state [ket or oper] state vector or density matrix input.

cbits [List of ints, optional] initialization of the classical bits.

U_list: list of Qobj, optional list of predefined unitaries corresponding to circuit.

measure_results [tuple of ints, optional] optional specification of each measurement result to enable post-selection. If specified, the measurement results are set to the tuple of bits (sequentially) instead of being chosen at random.

precompute_unitary: Boolean, optional Specify if computation is done by pre-computing and aggregating gate unitaries. Possibly a faster method in the case of large number of repeat runs with different state inputs.

Returns

final_state [Qobj] output state of the circuit run.

run_statistics(*state*, *U_list=None*, *cbits=None*, *precompute_unitary=False*)
Calculate all the possible outputs of a circuit (varied by measurement gates).

Parameters

state [ket or oper] state vector or density matrix input.

cbits [List of ints, optional] initialization of the classical bits.

U_list: list of Qobj, optional list of predefined unitaries corresponding to circuit.

measure_results [tuple of ints, optional] optional specification of each measurement result to enable post-selection. If specified, the measurement results are set to the tuple of bits (sequentially) instead of being chosen at random.

precompute_unitary: Boolean, optional Specify if computation is done by pre-computing and aggregating gate unitaries. Possibly a faster method in the case of large number of repeat runs with different state inputs.

Returns

result: CircuitResult Return a CircuitResult object containing output states and and their probabilities.

class CircuitResult(*final_states*, *probabilities*, *cbits=None*)

get_cbits(*index=None*)
Return list of classical bit outputs corresponding to the results.

Parameters

index: int Indicates i-th output, probability pair to be returned.

Returns

cbits: list of int or list of list of int list of classical bit outputs

get_final_states(*index=None*)
Return list of output states.

Parameters

index: int Indicates i-th state to be returned.

Returns

final_states: Qobj or list of Qobj. List of output kets or density matrices.

get_probabilities(*index=None*)
Return list of probabilities corresponding to the output states.

Parameters

index: int Indicates i-th probability to be returned.

Returns

probabilities: float or list of float Probabilities associated with each output state.

class CircuitSimulator(*qc*, *state=None*, *cbits=None*, *U_list=None*, *measure_results=None*, *mode='state_vector_simulator'*, *precompute_unitary=False*)

initialize(*state=None*, *cbits=None*, *measure_results=None*)
Reset Simulator state variables to start a new run.

Parameters

state: ket or oper ket or density matrix

cbits: list of int, optional initial value of classical bits

U_list: list of Qobj, optional list of predefined unitaries corresponding to circuit.

measure_results [tuple of ints, optional] optional specification of each measurement result to enable post-selection. If specified, the measurement results are set to the tuple of bits (sequentially) instead of being chosen at random.

run(*state*, *cbits=None*, *measure_results=None*)
Calculate the result of one instance of circuit run.

Parameters

state [ket or oper] state vector or density matrix input.

cbits [List of ints, optional] initialization of the classical bits.

measure_results [tuple of ints, optional] optional specification of each measurement result to enable post-selection. If specified, the measurement results are set to the tuple of bits (sequentially) instead of being chosen at random.

Returns

result: CircuitResult Return a CircuitResult object containing output state and probability.

run_statistics(*state*, *cbits=None*)
Calculate all the possible outputs of a circuit (varied by measurement gates).

Parameters

state [ket] state to be observed on specified by density matrix.

cbits [List of ints, optional] initialization of the classical bits.

Returns

result: CircuitResult Return a CircuitResult object containing output states and and their probabilities.

step()
Return state after one step of circuit evolution (gate or measurement).

Returns

state [ket or oper] state after one evolution step.

class Processor(*N*, *t1=None*, *t2=None*, *dims=None*, *spline_kind='step_func'*)
A simulator of a quantum device based on the QuTiP solver *qutip.mesolve*. It is defined by the available driving Hamiltonian and the decoherence time for each component systems. The processor can simulate the evolution under the given control pulses. Noisy evolution is supported by *Noise* and can be added to the processor.

Parameters

N: int The number of component systems.

t1: list or float, optional Characterize the decoherence of amplitude damping for each qubit. A list of size *N* or a float for all qubits.

t2: list of float, optional Characterize the decoherence of dephasing for each qubit. A list of size *N* or a float for all qubits.

dims: list, optional The dimension of each component system. Default value is a qubit system of `dim=[2,2,2,...,2]`

spline_kind: str, optional Type of the coefficient interpolation. Default is "step_func" Note that they have different requirement for the length of `coeff`.

- "step_func": The coefficient will be treated as a step function. E.g. `tlist=[0,1,2]` and `coeff=[3,2]`, means that the coefficient is 3 in t=[0,1) and 2 in t=[2,3). It requires `len(coeff)=len(tlist)-1` or `len(coeff)=len(tlist)`, but in the second case the last element of *coeff* has no effect.
- "cubic": Use cubic interpolation for the coefficient. It requires `len(coeff)=len(tlist)`

Attributes

N: int The number of component systems.

pulses: list of :class:`.Pulse` A list of control pulses of this device

t1: float or list Characterize the decoherence of amplitude damping of each qubit.

t2: float or list Characterize the decoherence of dephasing for each qubit.

noise: :class:`.Noise`, optional A list of noise objects. They will be processed when creating the noisy *qutip.QobjEvo* from the processor or run the simulation.

drift: :class:`qutip.qip.pulse.Drift` A *Drift* object representing the drift Hamiltonians.

dims: list The dimension of each component system. Default value is a qubit system of `dim=[2,2,2,...,2]`

spline_kind: str Type of the coefficient interpolation. See parameters of *Processor* for details.

add_control(*qobj*, *targets=None*, *cyclic_permutation=False*, *label=None*)

Add a control Hamiltonian to the processor. It creates a new *Pulse* object for the device that is turned off (`tlist = None`, `coeff = None`). To activate the pulse, one can set its *tlist* and *coeff*.

Parameters

qobj: :class:`qutip.Qobj` The Hamiltonian for the control pulse..

targets: list, optional The indices of the target qubits (or subquantum system of other dimensions).

cyclic_permutation: bool, optional If true, the Hamiltonian will be expanded for all cyclic permutation of the target qubits.

label: str, optional The label (name) of the pulse

add_drift(*qobj*, *targets*, *cyclic_permutation=False*)

Add a drift Hamiltonians. The drift Hamiltonians are intrinsic of the quantum system and cannot be controlled by external field.

Parameters

qobj: :class:`qutip.Qobj` The drift Hamiltonian.

targets: list The indices of the target qubits (or subquantum system of other dimensions).

add_noise(*noise*)

Add a noise object to the processor

Parameters

noise: :class:`.Noise` The noise object defined outside the processor

add_pulse(*pulse*)

Add a new pulse to the device.

Parameters

pulse: :class:`.Pulse` *Pulse* object to be added.

property coeffs
A list of the coefficients for all control pulses.

property ctrls
A list of Hamiltonians of all pulses.

eliminate_auxillary_modes(*U*)
Eliminate the auxillary modes like the cavity modes in cqed. (Defined in subclasses)

get_full_coeffs(*full_tlist=None*)
Return the full coefficients in a 2d matrix form. Each row corresponds to one pulse. If the *tlist* are different for different pulses, the length of each row will be same as the *full_tlist* (see method *get_full_tlist*). Interpolation is used for adding the missing coefficient according to *spline_kind*.

Returns

coeffs: array-like 2d The coefficients for all ideal pulses.

get_full_tlist(*tol=1e-10*)
Return the full tlist of the ideal pulses. If different pulses have different time steps, it will collect all the time steps in a sorted array.

Returns

full_tlist: array-like 1d The full time sequence for the ideal evolution.

get_noisy_pulses(*device_noise=False*, *drift=False*)
It takes the pulses defined in the *Processor* and adds noise according to *Processor.noise*. It does not modify the pulses saved in *Processor.pulses* but returns a new list. The length of the new list of noisy pulses might be longer because of drift Hamiltonian and device noise. They will be added to the end of the pulses list.

Parameters

device_noise: bool, optional If true, include pulse independent noise such as single qubit Relaxation. Default is False.

drift: bool, optional If true, include drift Hamiltonians. Default is False.

Returns

noisy_pulses: list of *Pulse* A list of noisy pulses.

get_operators_labels()
Get the labels for each Hamiltonian. It is used in the method``plot_pulses``. It is a 2-d nested list, in the plot, a different color will be used for each sublist.

get_qobjevo(*args=None*, *noisy=False*)
Create a *qutip.QobjEvo* representation of the evolution. It calls the method *get_noisy_pulses* and create the *QobjEvo* from it.

Parameters

args: dict, optional Arguments for *qutip.QobjEvo*

noisy: bool, optional If noise are included. Default is False.

Returns

qobjevo: *qutip.QobjEvo* The *qutip.QobjEvo* representation of the unitary evolution.

c_ops: list of *qutip.QobjEvo* A list of lindblad operators is also returned. if `noisy==Flase`, it is always an empty list.

load_circuit(*qc*)
Translate an *QubitCircuit* to its corresponding Hamiltonians. (Defined in subclasses)

plot_pulses(*title=None*, *figsize=(12, 6)*, *dpi=None*, *show_axis=False*, *rescale_pulse_coeffs=True*, *num_steps=1000*)
Plot the ideal pulse coefficients.

Parameters

title: str, optional Title for the plot.

figsize: tuple, optional The size of the figure.

dpi: int, optional The dpi of the figure.

show_axis: bool, optional If the axis are shown.

rescale_pulse_coeffs: bool, optional Rescale the hight of each pulses.

num_steps: int, optional Number of time steps in the plot.

Returns

fig: matplotlib.figure.Figure The *Figure* object for the plot.

ax: matplotlib.axes._subplots.AxesSubplot The axes for the plot.

Notes

`plot_pulses` only works for array_like coefficients

read_coeff(*file_name*, *inctime=True*)
Read the control amplitudes matrix and time list saved in the file by *save_amp*.

Parameters

file_name: string Name of the file.

inctime: bool, optional True if the time list in included in the first column.

Returns

tlist: array_like The time list read from the file.

coeffs: array_like The pulse matrix read from the file.

remove_pulse(*indices=None*, *label=None*)
Remove the control pulse with given indices.

Parameters

indices: int or list of int The indices of the control Hamiltonians to be removed.

label: str The label of the pulse

run(*qc=None*)
Calculate the propagator of the evolution by matrix exponentiation. This method won't include noise or collpase.

Parameters

qc: :class:`.QubitCircuit`, optional Takes the quantum circuit to be implemented. If not given, use the quantum circuit saved in the processor by *load_circuit*.

Returns

U_list: list The propagator matrix obtained from the physical implementation.

run_analytically(*init_state=None*, *qc=None*)
Simulate the state evolution under the given *qutip.QubitCircuit* with matrice exponentiation. It will calculate the propagator with matrix exponentiation and return a list of `qutip.Qobj`. This method won't include noise or collpase.

Parameters

qc: :class:`.QubitCircuit`, optional Takes the quantum circuit to be implemented. If not given, use the quantum circuit saved in the processor by `load_circuit`.

init_state: :class:`qutip.Qobj`, optional The initial state of the qubits in the register.

Returns

U_list: list A list of propagators obtained for the physical implementation.

run_state(*init_state=None*, *analytical=False*, *states=None*, *noisy=True*, *solver='mesolve'*, ***kwargs*)

If *analytical* is False, use *qutip.mesolve* to calculate the time of the state evolution and return the result. Other arguments of mesolve can be given as keyword arguments.

If *analytical* is True, calculate the propagator with matrix exponentiation and return a list of matrices. Noise will be neglected in this option.

Parameters

init_state: Qobj Initial density matrix or state vector (ket).

analytical: bool If True, calculate the evolution with matrices exponentiation.

states: :class:`qutip.Qobj`, optional Old API, same as init_state.

solver: str "mesolve" or "mcsolve"

****kwargs** Keyword arguments for the qutip solver.

Returns

evo_result: *qutip.solver.Result* If `analytical` is False, an instance of the class *qutip.solver.Result* will be returned.

If `analytical` is True, a list of matrices representation is returned.

save_coeff(*file_name*, *inctime=True*)

Save a file with the control amplitudes in each timeslot.

Parameters

file_name: string Name of the file.

inctime: bool, optional True if the time list should be included in the first column.

set_all_tlist(*tlist*)

Set the same *tlist* for all the pulses.

Parameters

tlist: array-like, optional A list of time at which the time-dependent coefficients are applied. See *Pulse* for detailed information`

class OptPulseProcessor(*N*, *drift=None*, *t1=None*, *t2=None*, *dims=None*)

A processor, which takes the Hamiltonian available as dynamic generators, calls the *qutip.control.optimize_pulse_unitary* function to find an optimized pulse sequence for the desired quantum circuit. The processor can simulate the evolution under the given control pulses using *qutip.mesolve*. (For attributes documentation, please refer to the parent class *Processor*)

Parameters

N: int The number of component systems.

drift: `:class:`qutip.Qobj` The drift Hamiltonian. The size must match the whole quantum system.

t1: list or float Characterize the decoherence of amplitude damping for each qubit. A list of size *N* or a float for all qubits.

t2: list of float Characterize the decoherence of dephasing for each qubit. A list of size *N* or a float for all qubits.

dims: list The dimension of each component system. Default value is a qubit system of `dim=[2,2,2,...,2]`

add_control(*qobj*, *targets=None*, *cyclic_permutation=False*, *label=None*)
Add a control Hamiltonian to the processor. It creates a new `Pulse` object for the device that is turned off (`tlist = None`, `coeff = None`). To activate the pulse, one can set its *tlist* and *coeff*.

Parameters

qobj: :class:`qutip.Qobj` The Hamiltonian for the control pulse..

targets: list, optional The indices of the target qubits (or subquantum system of other dimensions).

cyclic_permutation: bool, optional If true, the Hamiltonian will be expanded for all cyclic permutation of the target qubits.

label: str, optional The label (name) of the pulse

add_drift(*qobj*, *targets*, *cyclic_permutation=False*)
Add a drift Hamiltonians. The drift Hamiltonians are intrinsic of the quantum system and cannot be controlled by external field.

Parameters

qobj: :class:`qutip.Qobj` The drift Hamiltonian.

targets: list The indices of the target qubits (or subquantum system of other dimensions).

add_noise(*noise*)
Add a noise object to the processor

Parameters

noise: :class:`.Noise` The noise object defined outside the processor

add_pulse(*pulse*)
Add a new pulse to the device.

Parameters

pulse: :class:`.Pulse` *Pulse* object to be added.

property coeffs
A list of the coefficients for all control pulses.

property ctrls
A list of Hamiltonians of all pulses.

eliminate_auxillary_modes(*U*)
Eliminate the auxillary modes like the cavity modes in cqed. (Defined in subclasses)

get_full_coeffs(*full_tlist=None*)
Return the full coefficients in a 2d matrix form. Each row corresponds to one pulse. If the *tlist* are different for different pulses, the length of each row will be same as the *full_tlist* (see method *get_full_tlist*). Interpolation is used for adding the missing coefficient according to *spline_kind*.

Returns

coeffs: array-like 2d The coefficients for all ideal pulses.

get_full_tlist(*tol=1e-10*)
Return the full tlist of the ideal pulses. If different pulses have different time steps, it will collect all the time steps in a sorted array.

Returns

full_tlist: array-like 1d The full time sequence for the ideal evolution.

get_noisy_pulses(*device_noise=False*, *drift=False*)
It takes the pulses defined in the *Processor* and adds noise according to *Processor.noise*. It does not modify the pulses saved in *Processor.pulses* but returns a new list. The length of the new list of noisy pulses might be longer because of drift Hamiltonian and device noise. They will be added to the end of the pulses list.

Parameters

device_noise: bool, optional If true, include pulse independent noise such as single qubit Relaxation. Default is False.

drift: bool, optional If true, include drift Hamiltonians. Default is False.

Returns

noisy_pulses: list of *Pulse* A list of noisy pulses.

get_operators_labels()
Get the labels for each Hamiltonian. It is used in the method``plot_pulses``. It is a 2-d nested list, in the plot, a different color will be used for each sublist.

get_qobjevo(*args=None*, *noisy=False*)
Create a *qutip.QobjEvo* representation of the evolution. It calls the method *get_noisy_pulses* and create the *QobjEvo* from it.

Parameters

args: dict, optional Arguments for *qutip.QobjEvo*

noisy: bool, optional If noise are included. Default is False.

Returns

qobjevo: *qutip.QobjEvo* The *qutip.QobjEvo* representation of the unitary evolution.

c_ops: list of *qutip.QobjEvo* A list of lindblad operators is also returned. if `noisy==Flase`, it is always an empty list.

load_circuit(*qc*, *min_fid_err=inf*, *merge_gates=True*, *setting_args=None*, *verbose=False*, ***kwargs*)
Find the pulses realizing a given *QubitCircuit* using *qutip.control.optimize_pulse_unitary*. Further parameter for for *qutip.control.optimize_pulse_unitary* needs to be given as keyword arguments. By default, it first merge all the gates into one unitary and then find the control pulses for it. It can be turned off and one can set different parameters for different gates. See examples for details.

Parameters

qc: :class:`.QubitCircuit` or list of Qobj The quantum circuit to be translated.

min_fid_err: float, optional The minimal fidelity tolerance, if the fidelity error of any gate decomposition is higher, a warning will be given. Default is infinite.

merge_gates: boolean, optimal If True, merge all gate/Qobj into one Qobj and then find the optimal pulses for this unitary matrix. If False, find the optimal pulses for each gate/Qobj.

setting_args: dict, optional Only considered if merge_gates is False. It is a dictionary containing keyword arguments for different gates.

E.g.:

```
setting_args = {"SNOT": {"num_tslots": 10, "evo_time": 1},
                "SWAP": {"num_tslots": 30, "evo_time": 3},
                "CNOT": {"num_tslots": 30, "evo_time": 3}}
```

verbose: boolean, optional If true, the information for each decomposed gate will be shown. Default is False.

****kwargs** keyword arguments for *qutip.control.optimize_pulse_unitary*

Returns

tlist: array_like A NumPy array specifies the time of each coefficient

coeffs: array_like A 2d NumPy array of the shape (len(ctrls), len(tlist)-1). Each row corresponds to the control pulse sequence for one Hamiltonian.

Notes

`len(tlist) - 1 = coeffs.shape[1]` since tlist gives the beginning and the end of the pulses.

Examples

Same parameter for all the gates:

```
qc = QubitCircuit(N=1)
qc.add_gate("SNOT", 0)
num_tslots = 10
evo_time = 10
processor = OptPulseProcessor(N=1, drift=sigmaz(),
                              ctrls=[sigmax()])
# num_tslots and evo_time are two keyword arguments
tlist, coeffs = processor.load_circuit(
    qc, num_tslots=num_tslots, evo_time=evo_time)
```

Different parameters for different gates:

```
qc = QubitCircuit(N=2)
qc.add_gate("SNOT", 0)
qc.add_gate("SWAP", targets=[0, 1])
qc.add_gate('CNOT', controls=1, targets=[0])

processor = OptPulseProcessor(N=2, drift=tensor([sigmaz()]*2))
processor.add_control(sigmax(), cyclic_permutation=True)
processor.add_control(sigmay(), cyclic_permutation=True)
processor.add_control(tensor([sigmay(), sigmay()]))
setting_args = {"SNOT": {"num_tslots": 10, "evo_time": 1},
                "SWAP": {"num_tslots": 30, "evo_time": 3},
                "CNOT": {"num_tslots": 30, "evo_time": 3}}
tlist, coeffs = processor.load_circuit(qc,
                                       setting_args=setting_args,
                                       merge_gates=False)
```

plot_pulses(*title=None*, *figsize=(12, 6)*, *dpi=None*, *show_axis=False*, *rescale_pulse_coeffs=True*, *num_steps=1000*)

Plot the ideal pulse coefficients.

Parameters

title: str, optional Title for the plot.

figsize: tuple, optional The size of the figure.

dpi: int, optional The dpi of the figure.

show_axis: bool, optional If the axis are shown.

rescale_pulse_coeffs: bool, optional Rescale the hight of each pulses.

num_steps: int, optional Number of time steps in the plot.

Returns

fig: matplotlib.figure.Figure The *Figure* object for the plot.

ax: matplotlib.axes._subplots.AxesSubplot The axes for the plot.

Notes

`plot_pulses` only works for array_like coefficients

read_coeff(*file_name*, *inctime=True*)
Read the control amplitudes matrix and time list saved in the file by *save_amp*.

Parameters

file_name: string Name of the file.

inctime: bool, optional True if the time list in included in the first column.

Returns

tlist: array_like The time list read from the file.

coeffs: array_like The pulse matrix read from the file.

remove_pulse(*indices=None*, *label=None*)
Remove the control pulse with given indices.

Parameters

indices: int or list of int The indices of the control Hamiltonians to be removed.

label: str The label of the pulse

run(*qc=None*)
Calculate the propagator of the evolution by matrix exponentiation. This method won't include noise or collpase.

Parameters

qc: :class:`.QubitCircuit`, optional Takes the quantum circuit to be implemented. If not given, use the quantum circuit saved in the processor by *load_circuit*.

Returns

U_list: list The propagator matrix obtained from the physical implementation.

run_analytically(*init_state=None*, *qc=None*)
Simulate the state evolution under the given *qutip.QubitCircuit* with matrice exponentiation. It will calculate the propagator with matrix exponentiation and return a list of `qutip.Qobj`. This method won't include noise or collpase.

Parameters

qc: :class:`.QubitCircuit`, optional Takes the quantum circuit to be implemented. If not given, use the quantum circuit saved in the processor by `load_circuit`.

init_state: :class:`qutip.Qobj`, optional The initial state of the qubits in the register.

Returns

U_list: list A list of propagators obtained for the physical implementation.

run_state(*init_state=None*, *analytical=False*, *states=None*, *noisy=True*, *solver='mesolve'*, ***kwargs*)
If *analytical* is False, use `qutip.mesolve` to calculate the time of the state evolution and return the result. Other arguments of mesolve can be given as keyword arguments.

If *analytical* is True, calculate the propagator with matrix exponentiation and return a list of matrices. Noise will be neglected in this option.

Parameters

init_state: Qobj Initial density matrix or state vector (ket).

analytical: bool If True, calculate the evolution with matrices exponentiation.

states: :class:`qutip.Qobj`, optional Old API, same as init_state.

solver: str "mesolve" or "mcsolve"

****kwargs** Keyword arguments for the qutip solver.

Returns

evo_result: *qutip.solver.Result* If `analytical` is False, an instance of the class *qutip.solver.Result* will be returned.

If `analytical` is True, a list of matrices representation is returned.

save_coeff (*file_name*, *inctime=True*)
Save a file with the control amplitudes in each timeslot.

Parameters

file_name: string Name of the file.

inctime: bool, optional True if the time list should be included in the first column.

set_all_tlist (*tlist*)
Set the same *tlist* for all the pulses.

Parameters

tlist: array-like, optional A list of time at which the time-dependent coefficients are applied. See *Pulse* for detailed information`

class ModelProcessor (*N*, *correct_global_phase=True*, *t1=None*, *t2=None*)
The base class for a circuit processor simulating a physical device, e.g cavityQED, spinchain. The available Hamiltonian of the system is predefined. The processor can simulate the evolution under the given control pulses either numerically or analytically. It cannot be used alone, please refer to the sub-classes. (Only additional attributes are documented here, for others please refer to the parent class *Processor*)

Parameters

N: int The number of component systems.

correct_global_phase: boolean, optional If true, the analytical solution will track the global phase. It has no effect on the numerical solution.

t1: list or float Characterize the decoherence of amplitude damping for each qubit. A list of size *N* or a float for all qubits.

t2: list of float Characterize the decoherence of dephasing for each qubit. A list of size *N* or a float for all qubits.

Attributes

params: dict A Python dictionary contains the name and the value of the parameters in the physical realization, such as laser frequency, detuning etc.

correct_global_phase: float Save the global phase, the analytical solution will track the global phase. It has no effect on the numerical solution.

add_control (*qobj*, *targets=None*, *cyclic_permutation=False*, *label=None*)
Add a control Hamiltonian to the processor. It creates a new *Pulse* object for the device that is turned off (`tlist = None`, `coeff = None`). To activate the pulse, one can set its *tlist* and *coeff*.

Parameters

qobj: :class:`qutip.Qobj` The Hamiltonian for the control pulse..

targets: list, optional The indices of the target qubits (or subquantum system of other dimensions).

cyclic_permutation: bool, optional If true, the Hamiltonian will be expanded for all cyclic permutation of the target qubits.

label: str, optional The label (name) of the pulse

add_drift (*qobj*, *targets*, *cyclic_permutation=False*)
Add a drift Hamiltonians. The drift Hamiltonians are intrinsic of the quantum system and cannot be controlled by external field.

Parameters

qobj: :class:`qutip.Qobj` The drift Hamiltonian.

targets: list The indices of the target qubits (or subquantum system of other dimensions).

add_noise (*noise*)
Add a noise object to the processor

Parameters

noise: :class:`.Noise` The noise object defined outside the processor

add_pulse (*pulse*)
Add a new pulse to the device.

Parameters

pulse: :class:`.Pulse` *Pulse* object to be added.

property coeffs
A list of the coefficients for all control pulses.

property ctrls
A list of Hamiltonians of all pulses.

eliminate_auxillary_modes (*U*)
Eliminate the auxillary modes like the cavity modes in cqed. (Defined in subclasses)

get_full_coeffs (*full_tlist=None*)
Return the full coefficients in a 2d matrix form. Each row corresponds to one pulse. If the *tlist* are different for different pulses, the length of each row will be same as the *full_tlist* (see method *get_full_tlist*). Interpolation is used for adding the missing coefficient according to *spline_kind*.

Returns

coeffs: array-like 2d The coefficients for all ideal pulses.

get_full_tlist (*tol=1e-10*)
Return the full tlist of the ideal pulses. If different pulses have different time steps, it will collect all the time steps in a sorted array.

Returns

full_tlist: array-like 1d The full time sequence for the ideal evolution.

get_noisy_pulses (*device_noise=False*, *drift=False*)
It takes the pulses defined in the *Processor* and adds noise according to *Processor.noise*. It does not modify the pulses saved in *Processor.pulses* but returns a new list. The length of the new list of noisy pulses might be longer because of drift Hamiltonian and device noise. They will be added to the end of the pulses list.

Parameters

device_noise: bool, optional If true, include pulse independent noise such as single qubit Relaxation. Default is False.

drift: bool, optional If true, include drift Hamiltonians. Default is False.

Returns

noisy_pulses: list of *Pulse* A list of noisy pulses.

get_operators_labels()
Get the labels for each Hamiltonian. It is used in the method``plot_pulses``. It is a 2-d nested list, in the plot, a different color will be used for each sublist.

get_ops_and_u()
Get the labels for each Hamiltonian.

Returns

ctrls: list The list of Hamiltonians

coeffs: array_like The transposed pulse matrix

get_qobjevo(*args=None*, *noisy=False*)
Create a *qutip.QobjEvo* representation of the evolution. It calls the method *get_noisy_pulses* and create the *QobjEvo* from it.

Parameters

args: dict, optional Arguments for *qutip.QobjEvo*

noisy: bool, optional If noise are included. Default is False.

Returns

qobjevo: *qutip.QobjEvo* The *qutip.QobjEvo* representation of the unitary evolution.

c_ops: list of *qutip.QobjEvo* A list of lindblad operators is also returned. if `noisy==Flase`, it is always an empty list.

load_circuit(*qc*)
Translate an *QubitCircuit* to its corresponding Hamiltonians. (Defined in subclasses)

plot_pulses(*title=None*, *figsize=(12, 6)*, *dpi=None*, *show_axis=False*, *rescale_pulse_coeffs=True*, *num_steps=1000*)
Plot the ideal pulse coefficients.

Parameters

title: str, optional Title for the plot.

figsize: tuple, optional The size of the figure.

dpi: int, optional The dpi of the figure.

show_axis: bool, optional If the axis are shown.

rescale_pulse_coeffs: bool, optional Rescale the hight of each pulses.

num_steps: int, optional Number of time steps in the plot.

Returns

fig: matplotlib.figure.Figure The *Figure* object for the plot.

ax: matplotlib.axes._subplots.AxesSubplot The axes for the plot.

Notes

`plot_pulses` only works for array_like coefficients

pulse_matrix(*dt=0.01*)
Generates the pulse matrix for the desired physical system.

Returns

t, u, labels: Returns the total time and label for every operation.

read_coeff(*file_name*, *inctime=True*)
Read the control amplitudes matrix and time list saved in the file by *save_amp*.

Parameters

file_name: string Name of the file.

inctime: bool, optional True if the time list in included in the first column.

Returns

tlist: array_like The time list read from the file.

coeffs: array_like The pulse matrix read from the file.

remove_pulse(*indices=None*, *label=None*)
Remove the control pulse with given indices.

Parameters

indices: int or list of int The indices of the control Hamiltonians to be removed.

label: str The label of the pulse

run(*qc=None*)
Calculate the propagator of the evolution by matrix exponentiation. This method won't include noise or collpase.

Parameters

qc: :class:`.QubitCircuit`, optional Takes the quantum circuit to be implemented. If not given, use the quantum circuit saved in the processor by *load_circuit*.

Returns

U_list: list The propagator matrix obtained from the physical implementation.

run_analytically(*init_state=None*, *qc=None*)
Simulate the state evolution under the given *qutip.QubitCircuit* with matrice exponentiation. It will calculate the propagator with matrix exponentiation and return a list of `qutip.Qobj`. This method won't include noise or collpase.

Parameters

qc: :class:`.QubitCircuit`, optional Takes the quantum circuit to be implemented. If not given, use the quantum circuit saved in the processor by `load_circuit`.

init_state: :class:`qutip.Qobj`, optional The initial state of the qubits in the register.

Returns

U_list: list A list of propagators obtained for the physical implementation.

run_state(*init_state=None*, *analytical=False*, *qc=None*, *states=None*, ***kwargs*)
If *analytical* is False, use `qutip.mesolve` to calculate the time of the state evolution and return the result. Other arguments of mesolve can be given as keyword arguments. If *analytical* is True, calculate the propagator with matrix exponentiation and return a list of matrices.

Parameters

init_state: Qobj Initial density matrix or state vector (ket).

analytical: boolean If True, calculate the evolution with matrices exponentiation.

qc: :class:`.QubitCircuit`, optional A quantum circuit. If given, it first calls the `load_circuit` and then calculate the evolution.

states: :class:`qutip.Qobj`, optional Old API, same as init_state.

****kwargs** Keyword arguments for the qutip solver.

Returns

evo_result: *qutip.solver.Result* If `analytical` is False, an instance of the class *qutip.solver.Result* will be returned.

If `analytical` is True, a list of matrices representation is returned.

save_coeff (*file_name*, *inctime=True*)
Save a file with the control amplitudes in each timeslot.

Parameters

file_name: string Name of the file.

inctime: bool, optional True if the time list should be included in the first column.

set_all_tlist (*tlist*)
Set the same *tlist* for all the pulses.

Parameters

tlist: array-like, optional A list of time at which the time-dependent coefficients are applied. See *Pulse* for detailed information`

set_up_params ()
Save the parameters in the attribute *params* and check the validity. (Defined in subclasses)

Notes

All parameters will be multiplied by 2*pi for simplicity

to_array (*params*, *N*)
Transfer a parameter to an array.

class SpinChain (*N*, *correct_global_phase*, *sx*, *sz*, *sxsy*, *t1*, *t2*)
The processor based on the physical implementation of a spin chain qubits system. The available Hamiltonian of the system is predefined. The processor can simulate the evolution under the given control pulses either numerically or analytically. It is a base class and should not be used directly, please refer the the subclasses *qutip.qip.device.LinearSpinChain* and *qutip.qip.device.CircularSpinChain*. (Only additional attributes are documented here, for others please refer to the parent class *ModelProcessor*)

Parameters

N: int The number of qubits in the system.

correct_global_phase: float Save the global phase, the analytical solution will track the global phase. It has no effect on the numerical solution.

sx: int or list The delta for each of the qubits in the system.

sz: int or list The epsilon for each of the qubits in the system.

sxsy: int or list The interaction strength for each of the qubit pair in the system.

t1: list or float Characterize the decoherence of amplitude damping for each qubit. A list of size *N* or a float for all qubits.

t2: list of float Characterize the decoherence of dephasing for each qubit. A list of size *N* or a float for all qubits.

Attributes

sx: list The delta for each of the qubits in the system.

sz: list The epsilon for each of the qubits in the system.

sxsy: list The interaction strength for each of the qubit pair in the system.

sx_ops: list A list of sigmax Hamiltonians for each qubit.

sz_ops: list A list of sigmaz Hamiltonians for each qubit.

sxsy_ops: list A list of tensor(sigmax, sigmay) interacting Hamiltonians for each qubit.

sx_u: array_like Pulse matrix for sigmax Hamiltonians.

sz_u: array_like Pulse matrix for sigmaz Hamiltonians.

sxsy_u: array_like Pulse matrix for tensor(sigmax, sigmay) interacting Hamiltonians.

add_control(*qobj*, *targets=None*, *cyclic_permutation=False*, *label=None*)
Add a control Hamiltonian to the processor. It creates a new `Pulse` object for the device that is turned off (`tlist = None`, `coeff = None`). To activate the pulse, one can set its *tlist* and *coeff*.

Parameters

qobj: :class:`qutip.Qobj` The Hamiltonian for the control pulse..

targets: list, optional The indices of the target qubits (or subquantum system of other dimensions).

cyclic_permutation: bool, optional If true, the Hamiltonian will be expanded for all cyclic permutation of the target qubits.

label: str, optional The label (name) of the pulse

add_drift(*qobj*, *targets*, *cyclic_permutation=False*)
Add a drift Hamiltonians. The drift Hamiltonians are intrinsic of the quantum system and cannot be controlled by external field.

Parameters

qobj: :class:`qutip.Qobj` The drift Hamiltonian.

targets: list The indices of the target qubits (or subquantum system of other dimensions).

add_noise(*noise*)
Add a noise object to the processor

Parameters

noise: :class:`.Noise` The noise object defined outside the processor

add_pulse(*pulse*)
Add a new pulse to the device.

Parameters

pulse: :class:`.Pulse` *Pulse* object to be added.

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: :class:`.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.

property coeffs
A list of the coefficients for all control pulses.

property ctrls
A list of Hamiltonians of all pulses.

eliminate_auxillary_modes(*U*)
Eliminate the auxillary modes like the cavity modes in cqed. (Defined in subclasses)

get_full_coeffs(*full_tlist=None*)
Return the full coefficients in a 2d matrix form. Each row corresponds to one pulse. If the *tlist* are different for different pulses, the length of each row will be same as the *full_tlist* (see method *get_full_tlist*). Interpolation is used for adding the missing coefficient according to *spline_kind*.

Returns

coeffs: array-like 2d The coefficients for all ideal pulses.

get_full_tlist(*tol=1e-10*)
Return the full tlist of the ideal pulses. If different pulses have different time steps, it will collect all the time steps in a sorted array.

Returns

full_tlist: array-like 1d The full time sequence for the ideal evolution.

get_noisy_pulses(*device_noise=False*, *drift=False*)
It takes the pulses defined in the *Processor* and adds noise according to *Processor.noise*. It does not modify the pulses saved in *Processor.pulses* but returns a new list. The length of the new list of noisy pulses might be longer because of drift Hamiltonian and device noise. They will be added to the end of the pulses list.

Parameters

device_noise: bool, optional If true, include pulse independent noise such as single qubit Relaxation. Default is False.

drift: bool, optional If true, include drift Hamiltonians. Default is False.

Returns

noisy_pulses: list of *Pulse* A list of noisy pulses.

get_operators_labels()
Get the labels for each Hamiltonian. It is used in the method``plot_pulses``. It is a 2-d nested list, in the plot, a different color will be used for each sublist.

get_ops_and_u()
Get the labels for each Hamiltonian.

Returns

ctrls: list The list of Hamiltonians

coeffs: array_like The transposed pulse matrix

get_qobjevo(*args=None*, *noisy=False*)
Create a *qutip.QobjEvo* representation of the evolution. It calls the method *get_noisy_pulses* and create the *QobjEvo* from it.

Parameters

args: dict, optional Arguments for *qutip.QobjEvo*

noisy: bool, optional If noise are included. Default is False.

Returns

qobjevo: *qutip.QobjEvo* The *qutip.QobjEvo* representation of the unitary evolution.

c_ops: list of *qutip.QobjEvo* A list of lindblad operators is also returned. if `noisy==Flase`, it is always an empty list.

load_circuit(*qc*, *setup*, *schedule_mode='ASAP'*, *compiler=None*)

Decompose a *QubitCircuit* in to the control amplitude generating the corresponding evolution.

Parameters

qc: :class:`.QubitCircuit` Takes the quantum circuit to be implemented.

setup: string "linear" or "circular" for two sub-classes.

Returns

tlist: array_like A NumPy array specifies the time of each coefficient

coeffs: array_like A 2d NumPy array of the shape (len(ctrls), len(tlist)). Each row corresponds to the control pulse sequence for one Hamiltonian.

optimize_circuit(*qc*)

Take a quantum circuit/algorithm and convert it into the optimal form/basis for the desired physical system.

Parameters

qc: :class:`.QubitCircuit` Takes the quantum circuit to be implemented.

Returns

qc: *QubitCircuit* The circuit representation with elementary gates that can be implemented in this model.

plot_pulses(*title=None*, *figsize=(12, 6)*, *dpi=None*, *show_axis=False*, *rescale_pulse_coeffs=True*, *num_steps=1000*)

Plot the ideal pulse coefficients.

Parameters

title: str, optional Title for the plot.

figsize: tuple, optional The size of the figure.

dpi: int, optional The dpi of the figure.

show_axis: bool, optional If the axis are shown.

rescale_pulse_coeffs: bool, optional Rescale the hight of each pulses.

num_steps: int, optional Number of time steps in the plot.

Returns

fig: matplotlib.figure.Figure The *Figure* object for the plot.

ax: matplotlib.axes._subplots.AxesSubplot The axes for the plot.

Notes

`plot_pulses` only works for array_like coefficients

pulse_matrix(*dt=0.01*)
Generates the pulse matrix for the desired physical system.

Returns

t, u, labels: Returns the total time and label for every operation.

read_coeff(*file_name*, *inctime=True*)
Read the control amplitudes matrix and time list saved in the file by *save_amp*.

Parameters

file_name: string Name of the file.

inctime: bool, optional True if the time list in included in the first column.

Returns

tlist: array_like The time list read from the file.

coeffs: array_like The pulse matrix read from the file.

remove_pulse(*indices=None*, *label=None*)
Remove the control pulse with given indices.

Parameters

indices: int or list of int The indices of the control Hamiltonians to be removed.

label: str The label of the pulse

run(*qc=None*)
Calculate the propagator of the evolution by matrix exponentiation. This method won't include noise or collpase.

Parameters

qc: :class:`.QubitCircuit`, optional Takes the quantum circuit to be implemented. If not given, use the quantum circuit saved in the processor by *load_circuit*.

Returns

U_list: list The propagator matrix obtained from the physical implementation.

run_analytically(*init_state=None*, *qc=None*)
Simulate the state evolution under the given *qutip.QubitCircuit* with matrice exponentiation. It will calculate the propagator with matrix exponentiation and return a list of `qutip.Qobj`. This method won't include noise or collpase.

Parameters

qc: :class:`.QubitCircuit`, optional Takes the quantum circuit to be implemented. If not given, use the quantum circuit saved in the processor by `load_circuit`.

init_state: :class:`qutip.Qobj`, optional The initial state of the qubits in the register.

Returns

U_list: list A list of propagators obtained for the physical implementation.

run_state(*init_state=None*, *analytical=False*, *qc=None*, *states=None*, ***kwargs*)
If *analytical* is False, use `qutip.mesolve` to calculate the time of the state evolution and return the result. Other arguments of mesolve can be given as keyword arguments. If *analytical* is True, calculate the propagator with matrix exponentiation and return a list of matrices.

Parameters

init_state: Qobj Initial density matrix or state vector (ket).

analytical: boolean If True, calculate the evolution with matrices exponentiation.

qc: :class:`.QubitCircuit`, optional A quantum circuit. If given, it first calls the `load_circuit` and then calculate the evolution.

states: :class:`qutip.Qobj`, optional Old API, same as init_state.

****kwargs** Keyword arguments for the qutip solver.

Returns

evo_result: *qutip.solver.Result* If `analytical` is False, an instance of the class *qutip.solver.Result* will be returned.

If `analytical` is True, a list of matrices representation is returned.

save_coeff (*file_name*, *inctime=True*)
Save a file with the control amplitudes in each timeslot.

Parameters

file_name: string Name of the file.

inctime: bool, optional True if the time list should be included in the first column.

set_all_tlist (*tlist*)
Set the same *tlist* for all the pulses.

Parameters

tlist: array-like, optional A list of time at which the time-dependent coefficients are applied. See *Pulse* for detailed information`

set_up_ops (*N*)
Generate the Hamiltonians for the spinchain model and save them in the attribute *ctrls*.

Parameters

N: int The number of qubits in the system.

set_up_params (*sx*, *sz*)
Save the parameters in the attribute *params* and check the validity. The keys of *params* including "sx", "sz", and "sxsy", each mapped to a list for parameters corresponding to each qubits. For coupling strength "sxsy", list element i is the interaction between qubits i and i+1.

Parameters

sx: float or list The coefficient of sigmax in the model

sz: flaot or list The coefficient of sigmaz in the model

Notes

The coefficient of sxsy is defined in the submethods. All parameters will be multiplied by 2*pi for simplicity

to_array (*params*, *N*)
Transfer a parameter to an array.

class LinearSpinChain (*N*, *correct_global_phase=True*, *sx=0.25*, *sz=1.0*, *sxsy=0.1*, *t1=None*, *t2=None*)
A processor based on the physical implementation of a linear spin chain qubits system. The available Hamiltonian of the system is predefined. The processor can simulate the evolution under the given control pulses either numerically or analytically.

Parameters

N: int The number of qubits in the system.

correct_global_phase: float Save the global phase, the analytical solution will track the global phase. It has no effect on the numerical solution.

sx: int or list The delta for each of the qubits in the system.

sz: int or list The epsilon for each of the qubits in the system.

sxsy: int or list The interaction strength for each of the qubit pair in the system.

t1: list or float, optional Characterize the decoherence of amplitude damping for each qubit.

t2: list of float, optional Characterize the decoherence of dephasing for each qubit.

add_control(*qobj*, *targets=None*, *cyclic_permutation=False*, *label=None*)

Add a control Hamiltonian to the processor. It creates a new `Pulse` object for the device that is turned off (`tlist = None`, `coeff = None`). To activate the pulse, one can set its *tlist* and *coeff*.

Parameters

qobj: :class:`qutip.Qobj` The Hamiltonian for the control pulse..

targets: list, optional The indices of the target qubits (or subquantum system of other dimensions).

cyclic_permutation: bool, optional If true, the Hamiltonian will be expanded for all cyclic permutation of the target qubits.

label: str, optional The label (name) of the pulse

add_drift(*qobj*, *targets*, *cyclic_permutation=False*)

Add a drift Hamiltonians. The drift Hamiltonians are intrinsic of the quantum system and cannot be controlled by external field.

Parameters

qobj: :class:`qutip.Qobj` The drift Hamiltonian.

targets: list The indices of the target qubits (or subquantum system of other dimensions).

add_noise(*noise*)

Add a noise object to the processor

Parameters

noise: :class:`.Noise` The noise object defined outside the processor

add_pulse(*pulse*)

Add a new pulse to the device.

Parameters

pulse: :class:`.Pulse` *Pulse* object to be added.

adjacent_gates(*qc*)

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: :class:`.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.

property coeffs

A list of the coefficients for all control pulses.

property ctrls
A list of Hamiltonians of all pulses.

eliminate_auxillary_modes(*U*)
Eliminate the auxillary modes like the cavity modes in cqed. (Defined in subclasses)

get_full_coeffs(*full_tlist=None*)
Return the full coefficients in a 2d matrix form. Each row corresponds to one pulse. If the *tlist* are different for different pulses, the length of each row will be same as the *full_tlist* (see method *get_full_tlist*). Interpolation is used for adding the missing coefficient according to *spline_kind*.

Returns

coeffs: array-like 2d The coefficients for all ideal pulses.

get_full_tlist(*tol=1e-10*)
Return the full tlist of the ideal pulses. If different pulses have different time steps, it will collect all the time steps in a sorted array.

Returns

full_tlist: array-like 1d The full time sequence for the ideal evolution.

get_noisy_pulses(*device_noise=False*, *drift=False*)
It takes the pulses defined in the *Processor* and adds noise according to *Processor.noise*. It does not modify the pulses saved in *Processor.pulses* but returns a new list. The length of the new list of noisy pulses might be longer because of drift Hamiltonian and device noise. They will be added to the end of the pulses list.

Parameters

device_noise: bool, optional If true, include pulse independent noise such as single qubit Relaxation. Default is False.

drift: bool, optional If true, include drift Hamiltonians. Default is False.

Returns

noisy_pulses: list of *Pulse* A list of noisy pulses.

get_operators_labels()
Get the labels for each Hamiltonian. It is used in the method``plot_pulses``. It is a 2-d nested list, in the plot, a different color will be used for each sublist.

get_ops_and_u()
Get the labels for each Hamiltonian.

Returns

ctrls: list The list of Hamiltonians

coeffs: array_like The transposed pulse matrix

get_qobjevo(*args=None*, *noisy=False*)
Create a *qutip.QobjEvo* representation of the evolution. It calls the method *get_noisy_pulses* and create the *QobjEvo* from it.

Parameters

args: dict, optional Arguments for *qutip.QobjEvo*

noisy: bool, optional If noise are included. Default is False.

Returns

qobjevo: *qutip.QobjEvo* The *qutip.QobjEvo* representation of the unitary evolution.

c_ops: list of *qutip.QobjEvo* A list of lindblad operators is also returned. if `noisy==Flase`, it is always an empty list.

load_circuit(*qc*, *schedule_mode='ASAP'*, *compiler=None*)
Decompose a *QubitCircuit* in to the control amplitude generating the corresponding evolution.

Parameters

qc: :class:`.QubitCircuit` Takes the quantum circuit to be implemented.

setup: string "linear" or "circular" for two sub-classes.

Returns

tlist: array_like A NumPy array specifies the time of each coefficient

coeffs: array_like A 2d NumPy array of the shape (len(ctrls), len(tlist)). Each row corresponds to the control pulse sequence for one Hamiltonian.

optimize_circuit(*qc*)
Take a quantum circuit/algorithm and convert it into the optimal form/basis for the desired physical system.

Parameters

qc: :class:`.QubitCircuit` Takes the quantum circuit to be implemented.

Returns

qc: *QubitCircuit* The circuit representation with elementary gates that can be implemented in this model.

plot_pulses(*title=None*, *figsize=(12, 6)*, *dpi=None*, *show_axis=False*, *rescale_pulse_coeffs=True*, *num_steps=1000*)
Plot the ideal pulse coefficients.

Parameters

title: str, optional Title for the plot.

figsize: tuple, optional The size of the figure.

dpi: int, optional The dpi of the figure.

show_axis: bool, optional If the axis are shown.

rescale_pulse_coeffs: bool, optional Rescale the hight of each pulses.

num_steps: int, optional Number of time steps in the plot.

Returns

fig: matplotlib.figure.Figure The *Figure* object for the plot.

ax: matplotlib.axes._subplots.AxesSubplot The axes for the plot.

Notes

`plot_pulses` only works for array_like coefficients

pulse_matrix(*dt=0.01*)
Generates the pulse matrix for the desired physical system.

Returns

t, u, labels: Returns the total time and label for every operation.

read_coeff(*file_name*, *inctime=True*)
Read the control amplitudes matrix and time list saved in the file by *save_amp*.

Parameters

file_name: string Name of the file.

inctime: bool, optional True if the time list in included in the first column.

Returns

tlist: array_like The time list read from the file.

coeffs: array_like The pulse matrix read from the file.

remove_pulse(*indices=None*, *label=None*)
Remove the control pulse with given indices.

Parameters

indices: int or list of int The indices of the control Hamiltonians to be removed.

label: str The label of the pulse

run(*qc=None*)
Calculate the propagator of the evolution by matrix exponentiation. This method won't include noise or collpase.

Parameters

qc: :class:`.QubitCircuit`, optional Takes the quantum circuit to be implemented. If not given, use the quantum circuit saved in the processor by *load_circuit*.

Returns

U_list: list The propagator matrix obtained from the physical implementation.

run_analytically(*init_state=None*, *qc=None*)
Simulate the state evolution under the given *qutip.QubitCircuit* with matrice exponentiation. It will calculate the propagator with matrix exponentiation and return a list of `qutip.Qobj`. This method won't include noise or collpase.

Parameters

qc: :class:`.QubitCircuit`, optional Takes the quantum circuit to be implemented. If not given, use the quantum circuit saved in the processor by `load_circuit`.

init_state: :class:`qutip.Qobj`, optional The initial state of the qubits in the register.

Returns

U_list: list A list of propagators obtained for the physical implementation.

run_state(*init_state=None*, *analytical=False*, *qc=None*, *states=None*, ***kwargs*)
If *analytical* is False, use `qutip.mesolve` to calculate the time of the state evolution and return the result. Other arguments of mesolve can be given as keyword arguments. If *analytical* is True, calculate the propagator with matrix exponentiation and return a list of matrices.

Parameters

init_state: Qobj Initial density matrix or state vector (ket).

analytical: boolean If True, calculate the evolution with matrices exponentiation.

qc: :class:`.QubitCircuit`, optional A quantum circuit. If given, it first calls the `load_circuit` and then calculate the evolution.

states: :class:`qutip.Qobj`, optional Old API, same as init_state.

****kwargs** Keyword arguments for the qutip solver.

Returns

evo_result: *qutip.solver.Result* If `analytical` is False, an instance of the class `qutip.solver.Result` will be returned.

If `analytical` is True, a list of matrices representation is returned.

save_coeff(*file_name*, *inctime=True*)
Save a file with the control amplitudes in each timeslot.

Parameters

file_name: string Name of the file.

inctime: bool, optional True if the time list should be included in the first column.

set_all_tlist(*tlist*)
Set the same *tlist* for all the pulses.

Parameters

tlist: array-like, optional A list of time at which the time-dependent coefficients are applied. See `Pulse` for detailed information`

set_up_ops(*N*)
Generate the Hamiltonians for the spinchain model and save them in the attribute *ctrls*.

Parameters

N: int The number of qubits in the system.

set_up_params(*sx*, *sz*, *sxsy*)
Save the parameters in the attribute *params* and check the validity. The keys of *params* including "sx", "sz", and "sxsy", each mapped to a list for parameters corresponding to each qubits. For coupling strength "sxsy", list element i is the interaction between qubits i and i+1.

Parameters

sx: float or list The coefficient of sigmax in the model

sz: flaot or list The coefficient of sigmaz in the model

Notes

The coefficient of sxsy is defined in the submethods. All parameters will be multiplied by 2*pi for simplicity

to_array(*params*, *N*)
Transfer a parameter to an array.

class CircularSpinChain(*N*, *correct_global_phase=True*, *sx=0.25*, *sz=1.0*, *sxsy=0.1*, *t1=None*, *t2=None*)
A processor based on the physical implementation of a circular spin chain qubits system. The available Hamiltonian of the system is predefined. The processor can simulate the evolution under the given control pulses either numerically or analytically.

Parameters

N: int The number of qubits in the system.

correct_global_phase: float Save the global phase, the analytical solution will track the global phase. It has no effect on the numerical solution.

sx: int or list The delta for each of the qubits in the system.

sz: int or list The epsilon for each of the qubits in the system.

sxsy: int or list The interaction strength for each of the qubit pair in the system.

t1: list or float, optional Characterize the decoherence of amplitude damping for each qubit.

t2: list of float, optional Characterize the decoherence of dephasing for each qubit.

add_control(*qobj*, *targets=None*, *cyclic_permutation=False*, *label=None*)
Add a control Hamiltonian to the processor. It creates a new `Pulse` object for the device that is turned off (`tlist = None`, `coeff = None`). To activate the pulse, one can set its *tlist* and *coeff*.

Parameters

qobj: :class:`qutip.Qobj` The Hamiltonian for the control pulse..

targets: list, optional The indices of the target qubits (or subquantum system of other dimensions).

cyclic_permutation: bool, optional If true, the Hamiltonian will be expanded for all cyclic permutation of the target qubits.

label: str, optional The label (name) of the pulse

add_drift (*qobj*, *targets*, *cyclic_permutation=False*)
Add a drift Hamiltonians. The drift Hamiltonians are intrinsic of the quantum system and cannot be controlled by external field.

Parameters

qobj: :class:`qutip.Qobj` The drift Hamiltonian.

targets: list The indices of the target qubits (or subquantum system of other dimensions).

add_noise (*noise*)
Add a noise object to the processor

Parameters

noise: :class:`.Noise` The noise object defined outside the processor

add_pulse (*pulse*)
Add a new pulse to the device.

Parameters

pulse: :class:`.Pulse` *Pulse* object to be added.

adjacent_gates (*qc*)
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: :class:`.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.

property coeffs
A list of the coefficients for all control pulses.

property ctrls
A list of Hamiltonians of all pulses.

eliminate_auxillary_modes (*U*)
Eliminate the auxillary modes like the cavity modes in cqed. (Defined in subclasses)

get_full_coeffs (*full_tlist=None*)
Return the full coefficients in a 2d matrix form. Each row corresponds to one pulse. If the *tlist* are different for different pulses, the length of each row will be same as the *full_tlist* (see method *get_full_tlist*). Interpolation is used for adding the missing coefficient according to *spline_kind*.

Returns

coeffs: array-like 2d The coefficients for all ideal pulses.

get_full_tlist (*tol=1e-10*)
Return the full tlist of the ideal pulses. If different pulses have different time steps, it will collect all the time steps in a sorted array.

Returns

full_tlist: array-like 1d The full time sequence for the ideal evolution.

get_noisy_pulses(*device_noise=False*, *drift=False*)
It takes the pulses defined in the *Processor* and adds noise according to *Processor.noise*. It does not modify the pulses saved in *Processor.pulses* but returns a new list. The length of the new list of noisy pulses might be longer because of drift Hamiltonian and device noise. They will be added to the end of the pulses list.

Parameters

device_noise: bool, optional If true, include pulse independent noise such as single qubit Relaxation. Default is False.

drift: bool, optional If true, include drift Hamiltonians. Default is False.

Returns

noisy_pulses: list of *Pulse* A list of noisy pulses.

get_operators_labels()
Get the labels for each Hamiltonian. It is used in the method``plot_pulses``. It is a 2-d nested list, in the plot, a different color will be used for each sublist.

get_ops_and_u()
Get the labels for each Hamiltonian.

Returns

ctrls: list The list of Hamiltonians

coeffs: array_like The transposed pulse matrix

get_qobjevo(*args=None*, *noisy=False*)
Create a *qutip.QobjEvo* representation of the evolution. It calls the method *get_noisy_pulses* and create the *QobjEvo* from it.

Parameters

args: dict, optional Arguments for *qutip.QobjEvo*

noisy: bool, optional If noise are included. Default is False.

Returns

qobjevo: *qutip.QobjEvo* The *qutip.QobjEvo* representation of the unitary evolution.

c_ops: list of *qutip.QobjEvo* A list of lindblad operators is also returned. if `noisy==Flase`, it is always an empty list.

load_circuit(*qc*, *schedule_mode='ASAP'*, *compiler=None*)
Decompose a *QubitCircuit* in to the control amplitude generating the corresponding evolution.

Parameters

qc: :class:`.QubitCircuit` Takes the quantum circuit to be implemented.

setup: string "linear" or "circular" for two sub-classes.

Returns

tlist: array_like A NumPy array specifies the time of each coefficient

coeffs: array_like A 2d NumPy array of the shape (len(ctrls), len(tlist)). Each row corresponds to the control pulse sequence for one Hamiltonian.

optimize_circuit(*qc*)
Take a quantum circuit/algorithm and convert it into the optimal form/basis for the desired physical system.

Parameters

qc: :class:`.QubitCircuit` Takes the quantum circuit to be implemented.

Returns

qc: *QubitCircuit* The circuit representation with elementary gates that can be implemented in this model.

plot_pulses(*title=None*, *figsize=(12, 6)*, *dpi=None*, *show_axis=False*, *rescale_pulse_coeffs=True*, *num_steps=1000*)

Plot the ideal pulse coefficients.

Parameters

title: str, optional Title for the plot.

figsize: tuple, optional The size of the figure.

dpi: int, optional The dpi of the figure.

show_axis: bool, optional If the axis are shown.

rescale_pulse_coeffs: bool, optional Rescale the hight of each pulses.

num_steps: int, optional Number of time steps in the plot.

Returns

fig: matplotlib.figure.Figure The *Figure* object for the plot.

ax: matplotlib.axes._subplots.AxesSubplot The axes for the plot.

Notes

`plot_pulses` only works for array_like coefficients

pulse_matrix(*dt=0.01*)

Generates the pulse matrix for the desired physical system.

Returns

t, u, labels: Returns the total time and label for every operation.

read_coeff(*file_name*, *inctime=True*)

Read the control amplitudes matrix and time list saved in the file by *save_amp*.

Parameters

file_name: string Name of the file.

inctime: bool, optional True if the time list in included in the first column.

Returns

tlist: array_like The time list read from the file.

coeffs: array_like The pulse matrix read from the file.

remove_pulse(*indices=None*, *label=None*)

Remove the control pulse with given indices.

Parameters

indices: int or list of int The indices of the control Hamiltonians to be removed.

label: str The label of the pulse

run(*qc=None*)

Calculate the propagator of the evolution by matrix exponentiation. This method won't include noise or collpase.

Parameters

qc: :class:`.QubitCircuit`, optional Takes the quantum circuit to be implemented. If not given, use the quantum circuit saved in the processor by *load_circuit*.

Returns

U_list: list The propagator matrix obtained from the physical implementation.

run_analytically (*init_state=None*, *qc=None*)
Simulate the state evolution under the given *qutip.QubitCircuit* with matrice exponentiation. It will calculate the propagator with matrix exponentiation and return a list of `qutip.Qobj`. This method won't include noise or collpase.

Parameters

qc: :class:`.QubitCircuit`, optional Takes the quantum circuit to be implemented. If not given, use the quantum circuit saved in the processor by `load_circuit`.

init_state: :class:`qutip.Qobj`, optional The initial state of the qubits in the register.

Returns

U_list: list A list of propagators obtained for the physical implementation.

run_state (*init_state=None*, *analytical=False*, *qc=None*, *states=None*, ***kwargs*)
If *analytical* is False, use `qutip.mesolve` to calculate the time of the state evolution and return the result. Other arguments of mesolve can be given as keyword arguments. If *analytical* is True, calculate the propagator with matrix exponentiation and return a list of matrices.

Parameters

init_state: Qobj Initial density matrix or state vector (ket).

analytical: boolean If True, calculate the evolution with matrices exponentiation.

qc: :class:`.QubitCircuit`, optional A quantum circuit. If given, it first calls the `load_circuit` and then calculate the evolution.

states: :class:`qutip.Qobj`, optional Old API, same as init_state.

****kwargs** Keyword arguments for the qutip solver.

Returns

evo_result: *qutip.solver.Result* If `analytical` is False, an instance of the class `qutip.solver.Result` will be returned.

If `analytical` is True, a list of matrices representation is returned.

save_coeff (*file_name*, *inctime=True*)
Save a file with the control amplitudes in each timeslot.

Parameters

file_name: string Name of the file.

inctime: bool, optional True if the time list should be included in the first column.

set_all_tlist (*tlist*)
Set the same *tlist* for all the pulses.

Parameters

tlist: array-like, optional A list of time at which the time-dependent coefficients are applied. See `Pulse` for detailed information`

set_up_ops (*N*)
Generate the Hamiltonians for the spinchain model and save them in the attribute *ctrls*.

Parameters

N: int The number of qubits in the system.

set_up_params(*sx*, *sz*, *sxsy*)
Save the parameters in the attribute *params* and check the validity. The keys of *params* including "sx", "sz", and "sxsy", each mapped to a list for parameters corresponding to each qubits. For coupling strength "sxsy", list element i is the interaction between qubits i and i+1.

Parameters

sx: float or list The coefficient of sigmax in the model

sz: flaot or list The coefficient of sigmaz in the model

Notes

The coefficient of sxsy is defined in the submethods. All parameters will be multiplied by 2*pi for simplicity

to_array(*params*, *N*)
Transfer a parameter to an array.

class DispersiveCavityQED(*N*, *correct_global_phase=True*, *num_levels=10*, *deltamax=1.0*, *epsmax=9.5*, *w0=10.0*, *wq=None*, *eps=9.5*, *delta=0.0*, *g=0.01*, *t1=None*, *t2=None*)
The processor based on the physical implementation of a dispersive cavity QED system. The available Hamiltonian of the system is predefined. For a given pulse amplitude matrix, the processor can calculate the state evolution under the given control pulse, either analytically or numerically. (Only additional attributes are documented here, for others please refer to the parent class *ModelProcessor*)

Parameters

N: int The number of qubits in the system.

correct_global_phase: float, optional Save the global phase, the analytical solution will track the global phase. It has no effect on the numerical solution.

num_levels: int, optional The number of energy levels in the resonator.

deltamax: int or list, optional The coefficients of sigma-x for each of the qubits in the system.

epsmax: int or list, optional The coefficients of sigma-z for each of the qubits in the system.

w0: int, optional The base frequency of the resonator.

eps: int or list, optional The epsilon for each of the qubits in the system.

delta: int or list, optional The epsilon for each of the qubits in the system.

g: int or list, optional The interaction strength for each of the qubit with the resonator.

t1: list or float Characterize the decoherence of amplitude damping for each qubit. A list of size *N* or a float for all qubits.

t2: list of float Characterize the decoherence of dephasing for each qubit. A list of size *N* or a float for all qubits.

Attributes

sx_ops: list A list of sigmax Hamiltonians for each qubit.

sz_ops: list A list of sigmaz Hamiltonians for each qubit.

cavityqubit_ops: list A list of interacting Hamiltonians between cavity and each qubit.

sx_u: array_like Pulse matrix for sigmax Hamiltonians.

sz_u: array_like Pulse matrix for sigmaz Hamiltonians.

g_u: array_like Pulse matrix for interacting Hamiltonians between cavity and each qubit.

wq: list of float The frequency of the qubits calculated from eps and delta for each qubit.

Delta: list of float The detuning with respect to w0 calculated from wq and w0 for each qubit.

add_control(*qobj*, *targets=None*, *cyclic_permutation=False*, *label=None*)
Add a control Hamiltonian to the processor. It creates a new `Pulse` object for the device that is turned off (`tlist = None`, `coeff = None`). To activate the pulse, one can set its *tlist* and *coeff*.

Parameters

qobj: :class:`qutip.Qobj` The Hamiltonian for the control pulse..

targets: list, optional The indices of the target qubits (or subquantum system of other dimensions).

cyclic_permutation: bool, optional If true, the Hamiltonian will be expanded for all cyclic permutation of the target qubits.

label: str, optional The label (name) of the pulse

add_drift(*qobj*, *targets*, *cyclic_permutation=False*)
Add a drift Hamiltonians. The drift Hamiltonians are intrinsic of the quantum system and cannot be controlled by external field.

Parameters

qobj: :class:`qutip.Qobj` The drift Hamiltonian.

targets: list The indices of the target qubits (or subquantum system of other dimensions).

add_noise(*noise*)
Add a noise object to the processor

Parameters

noise: :class:`.Noise` The noise object defined outside the processor

add_pulse(*pulse*)
Add a new pulse to the device.

Parameters

pulse: :class:`.Pulse` *Pulse* object to be added.

property coeffs
A list of the coefficients for all control pulses.

property ctrls
A list of Hamiltonians of all pulses.

eliminate_auxillary_modes(*U*)
Eliminate the auxillary modes like the cavity modes in cqed.

get_full_coeffs(*full_tlist=None*)
Return the full coefficients in a 2d matrix form. Each row corresponds to one pulse. If the *tlist* are different for different pulses, the length of each row will be same as the *full_tlist* (see method *get_full_tlist*). Interpolation is used for adding the missing coefficient according to *spline_kind*.

Returns

coeffs: array-like 2d The coefficients for all ideal pulses.

get_full_tlist(*tol=1e-10*)
Return the full tlist of the ideal pulses. If different pulses have different time steps, it will collect all the time steps in a sorted array.

Returns

full_tlist: array-like 1d The full time sequence for the ideal evolution.

get_noisy_pulses(*device_noise=False*, *drift=False*)
It takes the pulses defined in the *Processor* and adds noise according to *Processor.noise*. It does not modify the pulses saved in *Processor.pulses* but returns a new list. The length of the new list of noisy pulses might be longer because of drift Hamiltonian and device noise. They will be added to the end of the pulses list.

Parameters

device_noise: bool, optional If true, include pulse independent noise such as single qubit Relaxation. Default is False.

drift: bool, optional If true, include drift Hamiltonians. Default is False.

Returns

noisy_pulses: list of *Pulse* A list of noisy pulses.

get_operators_labels()
Get the labels for each Hamiltonian. It is used in the method``plot_pulses``. It is a 2-d nested list, in the plot, a different color will be used for each sublist.

get_ops_and_u()
Get the labels for each Hamiltonian.

Returns

ctrls: list The list of Hamiltonians

coeffs: array_like The transposed pulse matrix

get_qobjevo(*args=None*, *noisy=False*)
Create a *qutip.QobjEvo* representation of the evolution. It calls the method *get_noisy_pulses* and create the *QobjEvo* from it.

Parameters

args: dict, optional Arguments for *qutip.QobjEvo*

noisy: bool, optional If noise are included. Default is False.

Returns

qobjevo: *qutip.QobjEvo* The *qutip.QobjEvo* representation of the unitary evolution.

c_ops: list of *qutip.QobjEvo* A list of lindblad operators is also returned. if `noisy==Flase`, it is always an empty list.

load_circuit(*qc*, *schedule_mode='ASAP'*, *compiler=None*)
Decompose a *QubitCircuit* in to the control amplitude generating the corresponding evolution.

Parameters

qc: :class:`.QubitCircuit` Takes the quantum circuit to be implemented.

Returns

tlist: array_like A NumPy array specifies the time of each coefficient

coeffs: array_like A 2d NumPy array of the shape (len(ctrls), len(tlist)). Each row corresponds to the control pulse sequence for one Hamiltonian.

optimize_circuit(*qc*)
Take a quantum circuit/algorithm and convert it into the optimal form/basis for the desired physical system.

Parameters

qc: :class:`.QubitCircuit` Takes the quantum circuit to be implemented.

Returns

qc: ***QubitCircuit*** The circuit representation with elementary gates that can be implemented in this model.

plot_pulses(*title=None*, *figsize=(12, 6)*, *dpi=None*, *show_axis=False*, *rescale_pulse_coeffs=True*, *num_steps=1000*)

Plot the ideal pulse coefficients.

Parameters

title: str, optional Title for the plot.

figsize: tuple, optional The size of the figure.

dpi: int, optional The dpi of the figure.

show_axis: bool, optional If the axis are shown.

rescale_pulse_coeffs: bool, optional Rescale the hight of each pulses.

num_steps: int, optional Number of time steps in the plot.

Returns

fig: matplotlib.figure.Figure The *Figure* object for the plot.

ax: matplotlib.axes._subplots.AxesSubplot The axes for the plot.

Notes

`plot_pulses` only works for array_like coefficients

pulse_matrix(*dt=0.01*)

Generates the pulse matrix for the desired physical system.

Returns

t, u, labels: Returns the total time and label for every operation.

read_coeff(*file_name*, *inctime=True*)

Read the control amplitudes matrix and time list saved in the file by *save_amp*.

Parameters

file_name: string Name of the file.

inctime: bool, optional True if the time list in included in the first column.

Returns

tlist: array_like The time list read from the file.

coeffs: array_like The pulse matrix read from the file.

remove_pulse(*indices=None*, *label=None*)

Remove the control pulse with given indices.

Parameters

indices: int or list of int The indices of the control Hamiltonians to be removed.

label: str The label of the pulse

run(*qc=None*)

Calculate the propagator of the evolution by matrix exponentiation. This method won't include noise or collpase.

Parameters

qc: :class:`.QubitCircuit`, optional Takes the quantum circuit to be implemented. If not given, use the quantum circuit saved in the processor by *load_circuit*.

Returns

U_list: list The propagator matrix obtained from the physical implementation.

run_analytically (*init_state=None*, *qc=None*)
Simulate the state evolution under the given *qutip.QubitCircuit* with matrice exponentiation. It will calculate the propagator with matrix exponentiation and return a list of `qutip.Qobj`. This method won't include noise or collpase.

Parameters

qc: :class:`.QubitCircuit`, optional Takes the quantum circuit to be implemented. If not given, use the quantum circuit saved in the processor by `load_circuit`.

init_state: :class:`qutip.Qobj`, optional The initial state of the qubits in the register.

Returns

U_list: list A list of propagators obtained for the physical implementation.

run_state (*init_state=None*, *analytical=False*, *qc=None*, *states=None*, ***kwargs*)
If *analytical* is False, use `qutip.mesolve` to calculate the time of the state evolution and return the result. Other arguments of mesolve can be given as keyword arguments. If *analytical* is True, calculate the propagator with matrix exponentiation and return a list of matrices.

Parameters

init_state: Qobj Initial density matrix or state vector (ket).

analytical: boolean If True, calculate the evolution with matrices exponentiation.

qc: :class:`.QubitCircuit`, optional A quantum circuit. If given, it first calls the `load_circuit` and then calculate the evolution.

states: :class:`qutip.Qobj`, optional Old API, same as init_state.

****kwargs** Keyword arguments for the qutip solver.

Returns

evo_result: *qutip.solver.Result* If `analytical` is False, an instance of the class `qutip.solver.Result` will be returned.

If `analytical` is True, a list of matrices representation is returned.

save_coeff (*file_name*, *inctime=True*)
Save a file with the control amplitudes in each timeslot.

Parameters

file_name: string Name of the file.

inctime: bool, optional True if the time list should be included in the first column.

set_all_tlist (*tlist*)
Set the same *tlist* for all the pulses.

Parameters

tlist: array-like, optional A list of time at which the time-dependent coefficients are applied. See `Pulse` for detailed information`

set_up_ops (*N*)
Generate the Hamiltonians for the spinchain model and save them in the attribute *ctrls*.

Parameters

N: int The number of qubits in the system.

set_up_params (*N*, *num_levels*, *deltamax*, *epsmax*, *w0*, *wq*, *eps*, *delta*, *g*)
Save the parameters in the attribute *params* and check the validity. The keys of *params* including "sx", "sz", "w0", "eps", "delta" and "g", each mapped to a list for parameters corresponding to each qubits. For coupling strength "g", list element i is the interaction between qubits i and i+1.

Parameters

N: int The number of qubits in the system.

num_levels: int The number of energy levels in the resonator.

deltamax: list The coefficients of sigma-x for each of the qubits in the system.

epsmax: list The coefficients of sigma-z for each of the qubits in the system.

wo: int The base frequency of the resonator.

wq: list The frequency of the qubits.

eps: list The epsilon for each of the qubits in the system.

delta: list The delta for each of the qubits in the system.

g: list The interaction strength for each of the qubit with the resonator.

Notes

All parameters will be multiplied by 2*pi for simplicity

to_array(*params*, *N*)
Transfer a parameter to an array.

class Noise
The base class representing noise in a processor. The noise object can be added to *Processor* and contributes to evolution.

get_noisy_dynamics(*dims*, *pulses*, *systematic_noise*)
Return a pulses list added with noise and the pulse independent noise in a dummy Pulse object.

Parameters

dims: list, optional The dimension of the components system, the default value is [2,2...,2] for qubits system.

pulses: list of :class:`.Pulse` The input pulses, on which the noise object is to be applied.

systematic_noise: :class:`.Pulse` The dummy pulse with no ideal control element.

Returns

noisy_pulses: list of *Pulse* Noisy pulses.

systematic_noise: *Pulse* The dummy pulse representing pulse independent noise.

class DecoherenceNoise(*c_ops*, *targets=None*, *coeff=None*, *tlist=None*, *all_qubits=False*)
The decoherence noise in a processor. It generates lindblad noise according to the given collapse operator *c_ops*.

Parameters

c_ops: :class:`qutip.Qobj` or list The Hamiltonian representing the dynamics of the noise.

targets: int or list, optional The indices of qubits that are acted on. Default is on all qubits

coeff: list, optional A list of the coefficients for the control Hamiltonians.

tlist: array_like, optional A NumPy array specifies the time of each coefficient.

all_qubits: bool, optional If *c_ops* contains only single qubits collapse operator, `all_qubits=True` will allow it to be applied to all qubits.

Attributes

c_ops: :class:`qutip.Qobj` or list The Hamiltonian representing the dynamics of the noise.

targets: int or list The indices of qubits that are acted on.

coeff: list A list of the coefficients for the control Hamiltonians.

tlist: array_like A NumPy array specifies the time of each coefficient.

all_qubits: bool If *c_ops* contains only single qubits collapse operator, `all_qubits=True` will allow it to be applied to all qubits.

get_noisy_dynamics(*dims=None*, *pulses=None*, *systematic_noise=None*)
Return a pulses list added with noise and the pulse independent noise in a dummy Pulse object.

Parameters

dims: list, optional The dimension of the components system, the default value is [2,2...,2] for qubits system.

pulses: list of :class:`.Pulse` The input pulses, on which the noise object is to be applied.

systematic_noise: :class:`.Pulse` The dummy pulse with no ideal control element.

Returns

noisy_pulses: list of *Pulse* Noisy pulses.

systematic_noise: *Pulse* The dummy pulse representing pulse independent noise.

class RelaxationNoise(*t1=None*, *t2=None*, *targets=None*)
The decoherence on each qubit characterized by two time scales t1 and t2.

Parameters

t1: float or list, optional Characterize the decoherence of amplitude damping for each qubit.

t2: float or list, optional Characterize the decoherence of dephasing for each qubit.

targets: int or list, optional The indices of qubits that are acted on. Default is on all qubits

Attributes

t1: float or list Characterize the decoherence of amplitude damping for each qubit.

t2: float or list Characterize the decoherence of dephasing for each qubit.

targets: int or list The indices of qubits that are acted on.

get_noisy_dynamics(*dims=None*, *pulses=None*, *systematic_noise=None*)
Return a pulses list added with noise and the pulse independent noise in a dummy Pulse object.

Parameters

dims: list, optional The dimension of the components system, the default value is [2,2...,2] for qubits system.

pulses: list of :class:`.Pulse` The input pulses, on which the noise object is to be applied.

systematic_noise: :class:`.Pulse` The dummy pulse with no ideal control element.

Returns

noisy_pulses: list of *Pulse* Noisy pulses.

systematic_noise: *Pulse* The dummy pulse representing pulse independent noise.

class ControlAmpNoise(*coeff*, *tlist=None*, *indices=None*)
The noise in the amplitude of the control pulse.

Parameters

coeff: list A list of the coefficients for the control Hamiltonians. For available choices, see *qutip.QobjEvo*.

tlist: array_like, optional A NumPy array specifies the time of each coefficient.

indices: list of int, optional The indices of target pulse in the list of pulses.

Attributes

———-

coeff: list A list of the coefficients for the control Hamiltonians. For available choices, see *qutip.QobjEvo*.

tlist: array_like A NumPy array specifies the time of each coefficient.

indices: list of int The indices of target pulse in the list of pulses.

get_noisy_dynamics(*dims=None*, *pulses=None*, *systematic_noise=None*)
Return a pulses list added with noise and the pulse independent noise in a dummy Pulse object.

Parameters

dims: list, optional The dimension of the components system, the default value is [2,2...,2] for qubits system.

pulses: list of :class:`.Pulse` The input pulses, on which the noise object is to be applied.

systematic_noise: :class:`.Pulse` The dummy pulse with no ideal control element.

Returns

noisy_pulses: list of *Pulse* Noisy pulses.

systematic_noise: *Pulse* The dummy pulse representing pulse independent noise.

class RandomNoise(*dt*, *rand_gen*, *indices=None*, ***kwargs*)
Random noise in the amplitude of the control pulse. The arguments for the random generator need to be given as key word arguments.

Parameters

dt: float, optional The time interval between two random amplitude. The coefficients of the noise are the same within this time range.

rand_gen: numpy.random, optional A random generator in numpy.random, it has to take a `size` parameter as the size of random numbers in the output array.

indices: list of int, optional The indices of target pulse in the list of pulses.

****kwargs:** Key word arguments for the random number generator.

Examples

```
>>> gaussnoise = RandomNoise(              dt=0.1, rand_gen=np.random.normal,␣
↪loc=mean, scale=std)
```

Attributes

dt: float, optional The time interval between two random amplitude. The coefficients of the noise are the same within this time range.

rand_gen: numpy.random, optional A random generator in numpy.random, it has to take a `size` parameter.

indices: list of int The indices of target pulse in the list of pulses.

****kwargs:** Key word arguments for the random number generator.

get_noisy_dynamics(*dims=None*, *pulses=None*, *systematic_noise=None*)
Return a pulses list added with noise and the pulse independent noise in a dummy Pulse object.

Parameters

dims: list, optional The dimension of the components system, the default value is [2,2...,2] for qubits system.

pulses: list of :class:`.Pulse` The input pulses, on which the noise object is to be applied.

systematic_noise: :class:`.Pulse` The dummy pulse with no ideal control element.

Returns

noisy_pulses: list of *Pulse* Noisy pulses.

systematic_noise: *Pulse* The dummy pulse representing pulse independent noise.

class Pulse(*qobj*, *targets*, *tlist=None*, *coeff=None*, *spline_kind=None*, *label=''*)
Representation of a control pulse and the pulse dependent noise. The pulse is characterized by the ideal control pulse, the coherent noise and the lindblad noise. The later two are lists of noisy evolution dynamics. Each dynamic element is characterized by four variables: *qobj*, *targets*, *tlist* and *coeff*.

See examples for different construction behavior.

Parameters

qobj: :class:'qutip.Qobj' The Hamiltonian of the ideal pulse.

targets: list target qubits of the ideal pulse (or subquantum system of other dimensions).

tlist: array-like, optional *tlist* of the ideal pulse. A list of time at which the time-dependent coefficients are applied. *tlist* does not have to be equidistant, but must have the same length or one element shorter compared to *coeff*. See documentation for the parameter *spline_kind*.

coeff: array-like or bool, optional Time-dependent coefficients of the ideal control pulse. If an array, the length must be the same or one element longer compared to *tlist*. See documentation for the parameter *spline_kind*. If a bool, the coefficient is a constant 1 or 0.

spline_kind: str, optional Type of the coefficient interpolation: "step_func" or "cubic".

-"step_func": The coefficient will be treated as a step function. E.g. `tlist=[0, 1, 2]` and `coeff=[3, 2]`, means that the coefficient is 3 in t=[0,1) and 2 in t=[2,3). It requires `len(coeff)=len(tlist)-1` or `len(coeff)=len(tlist)`, but in the second case the last element of *coeff* has no effect.

-"cubic": Use cubic interpolation for the coefficient. It requires `len(coeff)=len(tlist)`

label: str The label (name) of the pulse.

Examples

Create a pulse that is turned off

```
>>> Pulse(sigmaz(), 0)
>>> Pulse(sigmaz(), 0, None, None)
```

Create a time dependent pulse

```
>>> tlist = np.array([0., 1., 2., 4.])
>>> coeff = np.array([0.5, 1.2, 0.8])
>>> spline_kind = "step_func"
>>> Pulse(sigmaz(), 0, tlist=tlist, coeff=coeff, spline_kind="step_func")
```

Create a time independent pulse

```
>>> Pulse(sigmaz(), 0, coeff=True)
```

Create a constant pulse with time range

```
>>> Pulse(sigmaz(), 0, tlist=tlist, coeff=True)
```

Create an dummy Pulse (H=0)

```
>>> Pulse(None, None)
```

Attributes

ideal_pulse: :class:`._EvoElement` The ideal dynamic of the control pulse.

coherent_noise: list of :class:`._EvoElement` The coherent noise caused by the control pulse. Each dynamic element is still characterized by a time-dependent Hamiltonian.

lindblad_noise: list of :class:`._EvoElement` The dissipative noise of the control pulse. Each dynamic element will be treated as a (time-dependent) lindblad operator in the master equation.

spline_kind: str See parameter *spline_kind*.

label: str See parameter *label*.

add_coherent_noise(*qobj*, *targets*, *tlist=None*, *coeff=None*)
Add a new (time-dependent) Hamiltonian to the coherent noise.

Parameters

qobj: :class:'qutip.Qobj' The Hamiltonian of the pulse.

targets: list target qubits of the pulse (or subquantum system of other dimensions).

tlist: array-like, optional A list of time at which the time-dependent coefficients are applied. *tlist* does not have to be equidistant, but must have the same length or one element shorter compared to *coeff*. See documentation for the parameter *spline_kind* of `Pulse`.

coeff: array-like or bool, optional Time-dependent coefficients of the pulse noise. If an array, the length must be the same or one element longer compared to *tlist*. See documentation for the parameter *spline_kind* of `Pulse`. If a bool, the coefficient is a constant 1 or 0.

add_lindblad_noise(*qobj*, *targets*, *tlist=None*, *coeff=None*)
Add a new (time-dependent) lindblad noise to the coherent noise.

Parameters

qobj: :class:'qutip.Qobj' The collapse operator of the lindblad noise.

targets: list target qubits of the collapse operator (or subquantum system of other dimensions).

tlist: array-like, optional A list of time at which the time-dependent coefficients are applied. *tlist* does not have to be equidistant, but must have the same length or one element shorter compared to *coeff*. See documentation for the parameter *spline_kind* of `Pulse`.

coeff: array-like or bool, optional Time-dependent coefficients of the pulse noise. If an array, the length must be the same or one element longer compared to *tlist*. See documentation for the parameter *spline_kind* of `Pulse`. If a bool, the coefficient is a constant 1 or 0.

property coeff
See parameter *coeff*.

get_full_tlist(*tol=1e-10*)
Return the full tlist of the pulses and noise. It means that if different `tlist` are present, they will be merged to one with all time points stored in a sorted array.

Returns

full_tlist: array-like 1d The full time sequence for the noisy evolution.

get_ideal_qobj(*dims*)
Get the Hamiltonian of the ideal pulse.

Parameters

dims: int or list Dimension of the system. If int, we assume it is the number of qubits in the system. If list, it is the dimension of the component systems.

Returns

qobj: *qutip.Qobj* The Hamiltonian of the ideal pulse.

get_ideal_qobjevo(*dims*)
Get a *QobjEvo* representation of the ideal evolution.

Parameters

dims: int or list Dimension of the system. If int, we assume it is the number of qubits in the system. If list, it is the dimension of the component systems.

Returns

ideal_evo: *qutip.QobjEvo* A *QobjEvo* representing the ideal evolution.

get_noisy_qobjevo(*dims*)
Get the *QobjEvo* representation of the noisy evolution. The result can be used directly as input for the qutip solvers.

Parameters

dims: int or list Dimension of the system. If int, we assume it is the number of qubits in the system. If list, it is the dimension of the component systems.

Returns

noisy_evo: *qutip.QobjEvo* A `QobjEvo` representing the ideal evolution and coherent noise.

c_ops: list of *qutip.QobjEvo* A list of (time-dependent) lindbald operators.

print_info()
Print the information of the pulse, including the ideal dynamics, the coherent noise and the lindblad noise.

property qobj
See parameter *qobj*.

property targets
See parameter *targets*.

property tlist
See parameter *tlist*

class GateCompiler(*N*, *params=None*, *pulse_dict=None*)
Base class. It compiles a *QubitCircuit* into the pulse sequence for the processor. The core member function *compile* calls compiling method from the sub-class and concatenate the compiled pulses.

Parameters

N: int The number of the component systems.

params: dict, optional A Python dictionary contains the name and the value of the parameters, such as laser frequency, detuning etc. It will be saved in the class attributes and can be used to calculate the control pulses.

pulse_dict: dict, optional A map between the pulse label and its index in the pulse list. If given, the compiled pulse can be identified with `(pulse_label, coeff)`, instead of `(pulse_index, coeff)`. The number of key-value pairs should match the number of pulses in the processor. If it is empty, an integer `pulse_index` needs to be used in the compiling routine saved under the attributes `gate_compiler`.

Attributes

gate_compiler: dict The Python dictionary in the form of {gate_name: compiler_function}. It saves the compiling routine for each gate. See sub-classes for implementation. Note that for continuous pulse, the first coeff should always be 0.

args: dict Arguments for individual compiling routines. It adds more flexibility in customizing compiler.

compile(*circuit*, *schedule_mode=None*, *args=None*)

Compile the the native gates into control pulse sequence. It calls each compiling method and concatenates the compiled pulses.

Parameters

circuit: :class:`.QubitCircuit` or list of *Gate* A list of elementary gates that can be implemented in the corresponding hardware. The gate names have to be in *gate_compiler*.

schedule_mode: str, optional `"ASAP"` for "as soon as possible" or `"ALAP"` for "as late as possible" or `False` or `None` for no schedule. Default is None.

args: dict, optional A dictionary of arguments used in a specific gate compiler function.

Returns

tlist: array_like A NumPy array specifies the time of each coefficient

coeffs: array_like A 2d NumPy array of the shape `(len(ctrls), len(tlist))`. Each row corresponds to the control pulse sequence for one Hamiltonian.

globalphase_compiler(*gate*, *args*)

Compiler for the GLOBALPHASE gate

class CavityQEDCompiler(*N*, *params*, *pulse_dict*, *global_phase=0.0*)

Decompose a *QubitCircuit* into the pulse sequence for the processor.

Parameters

N: int The number of qubits in the system.

params: dict A Python dictionary contains the name and the value of the parameters. See *DispersiveCavityQED.set_up_params* for the definition.

global_phase: float, optional Record of the global phase change and will be returned.

pulse_dict: dict, optional A map between the pulse label and its index in the pulse list. If given, the compiled pulse can be identified with `(pulse_label, coeff)`, instead of `(pulse_index, coeff)`. The number of key-value pairs should match the number of pulses in the processor. If it is empty, an integer `pulse_index` needs to be used in the compiling routine saved under the attributes `gate_compiler`.

Attributes

N: int The number of the component systems.

params: dict A Python dictionary contains the name and the value of the parameters, such as laser frequency, detuning etc.

pulse_dict: dict A map between the pulse label and its index in the pulse list.

gate_compiler: dict The Python dictionary in the form of {gate_name: decompose_function}. It saves the decomposition scheme for each gate.

compile(*circuit*, *schedule_mode=None*, *args=None*)
Compile the the native gates into control pulse sequence. It calls each compiling method and concatenates the compiled pulses.

Parameters

circuit: :class:`.QubitCircuit` or list of *Gate* A list of elementary gates that can be implemented in the corresponding hardware. The gate names have to be in *gate_compiler*.

schedule_mode: str, optional `"ASAP"` for "as soon as possible" or `"ALAP"` for "as late as possible" or `False` or `None` for no schedule. Default is None.

args: dict, optional A dictionary of arguments used in a specific gate compiler function.

Returns

tlist: array_like A NumPy array specifies the time of each coefficient

coeffs: array_like A 2d NumPy array of the shape `(len(ctrls), len(tlist))`. Each row corresponds to the control pulse sequence for one Hamiltonian.

globalphase_compiler(*gate*, *args*)
Compiler for the GLOBALPHASE gate

iswap_compiler(*gate*, *args*)
Compiler for the ISWAP gate

rx_compiler(*gate*, *args*)
Compiler for the RX gate

rz_compiler(*gate*, *args*)
Compiler for the RZ gate

sqrtiswap_compiler(*gate*, *args*)
Compiler for the SQRTISWAP gate

Notes

This version of sqrtiswap_compiler has very low fidelity, please use iswap

class SpinChainCompiler(*N*, *params*, *pulse_dict*, *setup='linear'*, *global_phase=0.0*)
Compile a *QubitCircuit* into the pulse sequence for the processor.

Parameters

N: int The number of qubits in the system.

params: dict A Python dictionary contains the name and the value of the parameters. See *SpinChain.set_up_params* for the definition.

setup: string "linear" or "circular" for two sub-classes.

global_phase: bool Record of the global phase change and will be returned.

pulse_dict: dict, optional A map between the pulse label and its index in the pulse list. If given, the compiled pulse can be identified with `(pulse_label, coeff)`, instead of `(pulse_index, coeff)`. The number of key-value pairs should match the number of pulses in the processor. If it is empty, an integer `pulse_index` needs to be used in the compiling routine saved under the attributes `gate_compiler`.

Attributes

N: int The number of the component systems.

params: dict A Python dictionary contains the name and the value of the parameters, such as laser frequency, detuning etc.

pulse_dict: dict A map between the pulse label and its index in the pulse list.

gate_compiler: dict The Python dictionary in the form of {gate_name: decompose_function}. It saves the decomposition scheme for each gate.

setup: string "linear" or "circular" for two sub-classes.

global_phase: bool Record of the global phase change and will be returned.

compile(*circuit*, *schedule_mode=None*, *args=None*)
Compile the the native gates into control pulse sequence. It calls each compiling method and concatenates the compiled pulses.

Parameters

circuit: :class:`.QubitCircuit` or list of *Gate* A list of elementary gates that can be implemented in the corresponding hardware. The gate names have to be in *gate_compiler*.

schedule_mode: str, optional `"ASAP"` for "as soon as possible" or `"ALAP"` for "as late as possible" or `False` or `None` for no schedule. Default is None.

args: dict, optional A dictionary of arguments used in a specific gate compiler function.

Returns

tlist: array_like A NumPy array specifies the time of each coefficient

coeffs: array_like A 2d NumPy array of the shape `(len(ctrls), len(tlist))`. Each row corresponds to the control pulse sequence for one Hamiltonian.

globalphase_compiler(*gate*, *args*)
Compiler for the GLOBALPHASE gate

iswap_compiler(*gate*, *args*)
Compiler for the ISWAP gate

rx_compiler(*gate*, *args*)
Compiler for the RX gate

rz_compiler(*gate*, *args*)
Compiler for the RZ gate

sqrtiswap_compiler(*gate*, *args*)
Compiler for the SQRTISWAP gate

class Scheduler(*method='ALAP'*, *constraint_functions=None*)
A gate (pulse) scheduler for quantum circuits (instructions). It schedules a given circuit or instructions to reduce the total execution time by parallelization. It uses heuristic methods mainly from in https://doi.org/10.1117/12.666419.

The scheduler includes two methods, "ASAP", as soon as possible, and "ALAP", as late as possible. The later is commonly used in quantum computation because of the finite lifetime of qubits.

The scheduler aims at pulse schedule and therefore does not consider merging gates to reduce the gates number. It assumes that the input circuit is optimized at the gate level and matches the hardware topology.

Parameters

method: str "ASAP" for as soon as possible. "ALAP" for as late as possible.

constraint_functions: list, optional A list of hardware constraint functions. Default includes a function *qubit_contraint*, i.e. one qubit cannot be used by two gates at the same time.

apply_constraint(*ind1*, *ind2*, *instructions*)

Apply hardware constraint to check if two instructions can be executed in parallel.

Parameters

ind1, ind2: int indices of the two instructions

instructions: list The instruction list

commutation_rules(*ind1*, *ind2*, *instructions*)

Determine if two gates commute, given that their used qubits overlap. Since usually the input gates are already in a universal gate sets, it uses an oversimplified condition:

If the two gates do not have the same name, they are considered as not commuting. If they are the same gate and have the same controls or targets, they are considered as commuting. E.g. *CNOT 0, 1* commute with *CNOT 0, 2*.

schedule(*circuit*, *gates_schedule=False*, *return_cycles_list=False*, *random_shuffle=False*, *repeat_num=0*)

Schedule a *QubitCircuit*, a list of *Gates* or a list of *Instruction*. For pulse schedule, the execution time for each *Instruction* is given in its *duration* attributes.

The scheduler first generates a quantum gates dependency graph, containing information about which gates have to be executed before some other gates. The graph preserves the mobility of the gates, i.e. commuting gates are not dependent on each other, even if they use the same qubits. Next, it computes the longest distance of each node to the start and end nodes. The distance for each dependency arrow is defined by the execution time of the instruction (By default, it is 1 for all gates). This is used as a priority measure in the next step. The gate with a longer distance to the end node and a shorter distance to the start node has higher priority. In the last step, it uses a list-schedule algorithm with hardware constraint and priority and returns a list of cycles for gates/instructions.

For pulse schedule, an additional step is required to compute the start time of each instruction. It adds the additional dependency caused by hardware constraint to the graph and recomputes the distance of each node to the start and end node. This distance is then converted to the start time of each instruction.

Parameters

circuit: QubitCircuit or list For gate schedule, it should be a QubitCircuit or a list of Gate objects. For pulse schedule, it should be a list of Instruction objects, each with an attribute *duration* that indicates the execution time of this instruction.

gates_schedule: bool, optional *True*, if only gates schedule is needed. This saves some computation that is only useful to pulse schedule. If the input *circuit* is a *QubitCircuit*, it will be assigned to *True* automatically. Otherwise, the default is *False*.

return_cycles_list: bool, optional If *True*, the method returns the *cycles_list*, e.g. [{0, 2}, {1, 3}], which means that the first cycle contains gates0 and gates2 while the second cycle contains gates1 and gates3. It is only usefull for gates schedule.

random_shuffle: bool, optional If the commuting gates are randomly scuffled to explore larger search space.

repeat_num: int, optional Repeat the scheduling several times and use the best result. Used together with `random_shuffle=Ture`.

Returns

gate_cycle_indices or instruction_start_time: list The cycle indices for each gate or the start time for each instruction.

Examples

```
>>> from qutip.qip.circuit import QubitCircuit
>>> from qutip.qip.scheduler import Scheduler
>>> circuit = QubitCircuit(7)
>>> circuit.add_gate("SNOT", 3)  # gate0
>>> circuit.add_gate("CZ", 5, 3)  # gate1
>>> circuit.add_gate("CZ", 4, 3)  # gate2
>>> circuit.add_gate("CZ", 2, 3)  # gate3
>>> circuit.add_gate("CZ", 6, 5)  # gate4
>>> circuit.add_gate("CZ", 2, 6)  # gate5
>>> circuit.add_gate("SWAP", [0, 2])  # gate6
>>>
>>> scheduler = Scheduler("ASAP")
>>> scheduler.schedule(circuit, gates_schedule=True)
[0, 1, 3, 2, 2, 3, 4]
```

The result list is the cycle indices for each gate. It means that the circuit can be executed in 5 gate cycles: `[gate0, gate1, (gate3, gate4), (gate2, gate5), gate6]` Notice that gate3 and gate4 commute with gate2, therefore, the order is changed to reduce the number of cycles.

class Instruction(*gate*, *tlist=None*, *pulse_info=()*, *duration=1*)

The instruction that implements a quantum gate. It contains the control pulse required to implement the gate on a particular hardware model.

Parameters

gate: :class:`.Gate` The quantum gate.

duration: list, optional The execution time needed for the instruction.

tlist: array_like, optional A list of time at which the time-dependent coefficients are applied. See *Pulse* for detailed information`

pulse_info: list, optional A list of tuples, each tuple corresponding to a pair of pulse label and pulse coefficient, in the format (str, array_like). This pulses will implement the desired gate.

Attributes

targets: list, optional The target qubits.

controls: list, optional The control qubits.

used_qubits: set Union of the control and target qubits.

5.1.13 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

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 `alg`.

disp_conv_msg [bool] Set true to display a convergence message (for scipy.optimize.minimize methods anyway)

optim_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

approx_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)

termination_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 [*qutip.control.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.

dumping [string] The level of data dumping that will occur during the 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.

property 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.

References

[1], [2]

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_norm [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

evo_full_initial [Qobj] The evolution operator from t=0 to t=T based on the initial 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

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.

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 Optimizer object

tslot_computer [TimeslotComputer (subclass instance)] Used to manage when the timeslot dynamics generators, propagators, gradients etc are updated

prop_computer [PropagatorComputer (subclass instance)] Used to compute the propagators and their gradients

fid_computer [FidelityComputer (subclass instance)] Used to computer the fidelity error and the fidelity error gradient.

memory_optimization [int] Level of memory optimisation. Setting to 0 (default) means that execution speed is prioritized over memory. Setting to 1 means that some memory prioritisation steps will be taken, for instance using Qobj (and hence sparse arrays) as the the internal operator data type, and not caching some operators Potentially further memory saving maybe made with memory_optimization > 1. The options are processed in _set_memory_optimizations, see this for more information. Individual memory saving options can be switched by settting them directly (see below)

oper_dtype [type] Data type for internal dynamics generators, propagators and time evolution operators. This can be ndarray or Qobj. Qobj may perform better for larger systems, and will also perform better when (custom) fidelity measures use Qobj methods such as partial trace. See _choose_oper_dtype for how this is chosen when not specified

cache_phased_dyn_gen [bool] If True then the dynamics generators will be saved with and without the propagation prefactor (if there is one) Defaults to True when memory_optimization=0, otherwise False

cache_prop_grad [bool] If the True then the propagator gradients (for exact gradients) will be computed when the propagator are computed and cache until the are used by the fidelity computer. If False then the fidelity computer will calculate them as needed. Defaults to True when memory_optimization=0, otherwise False

cache_dyn_gen_eigenvectors_adj: bool If True then DynamicsUnitary will cached the adjoint of the Hamiltion eignvector matrix Defaults to True when memory_optimization=0, otherwise False

sparse_eigen_decomp: bool If True then DynamicsUnitary will use the sparse eigenvalue decomposition. Defaults to True when memory_optimization<=1, otherwise False

num_tslots [integer] Number of timeslots (aka timeslices)

num_ctrls [integer] calculate the of controls from the length of the control list

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 initialize_controls is called then num_tslots and evo_time are calculated from tau, otherwise tau is generated from num_tslots and evo_time, that is equal size time slices

time [array[num_tslots+1] of float] Cumulative time for the evolution, that is the time at the start of each time slice

drift_dyn_gen [Qobj or list of Qobj] Drift or system dynamics generator (Hamiltonian) Matrix defining the underlying dynamics of the system Can also be a list of Qobj (length num_tslots) for time varying drift dynamics

ctrl_dyn_gen [List of Qobj] Control dynamics generator (Hamiltonians) List of matrices defining the control dynamics

initial [Qobj] Starting state / gate The matrix giving the initial state / gate, i.e. at time 0 Typically the identity for gate evolution

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) for each control in each timeslot

initial_ctrl_scaling [float] Scale factor applied to be applied the control amplitudes when they are initialised This is used by the PulseGens rather than in any fucntions in this class

initial_ctrl_offset [float] Linear offset applied to be applied the control amplitudes when they are initialised This is used by the PulseGens rather than in any fucntions in this class

dyn_gen [List of Qobj] List of combined dynamics generators (Qobj) for each timeslot

prop [list of Qobj] List of propagators (Qobj) for each timeslot

prop_grad [array[num_tslots, num_ctrls] of Qobj] Array of propagator gradients (Qobj) for each timeslot, control

fwd_evo [List of Qobj] List of evolution operators (Qobj) from the initial to the given

onwd_evo [List of Qobj] List of evolution operators (Qobj) from the initial to the given

onto_evo [List of Qobj] List of evolution operators (Qobj) from the initial to the given

evo_current [Boolean] Used to flag that the dynamics used to calculate the evolution operators is current. It is set to False when the amplitudes change

fact_mat_round_prec [float] Rounding precision used when calculating the factor matrix to determine if two eigenvalues are equivalent Only used when the PropagatorComputer uses diagonalisation

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 recomputation. level 1 checks all propagators level 2 checks eigen basis as well Default is 0

unitarity_tol : Tolerance used in checking if operator is unitary Default is 1e-10

dump [*qutip.control.dump.DynamicsDump*] Store of historical calculation data. Set to None (Default) for no storing of historical data Use dumping property to set level of data dumping

dumping [string] The level of data dumping that will occur during the time evolution calculation.

dump_to_file [bool] If set True then data will be dumped to file during the calculations dumping will be set to SUMMARY during init_evo 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 dyn_params. If dump is None then will return None or will set dumping to SUMMARY when setting a path

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

property 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!

property dyn_gen
List of combined dynamics generators (Qobj) for each timeslot

property dyn_gen_phase
Some op that is applied to the dyn_gen before expontiating to get the propagator. See *phase_application* for how this is applied

flag_system_changed()
Flag evolution, fidelity and gradients as needing recalculation

property full_evo
Full evolution - time evolution at final time slot

property 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

property num_ctrls
calculate the of controls from the length of the control list sets the num_ctrls property, which can be used alternatively subsequently

property onto_evo
List of evolution operators (Qobj) from the initial to the given timeslot

property onwd_evo
List of evolution operators (Qobj) from the initial to the given timeslot

property phase_application
scalar(string), default='preop' Determines how the phase is applied to the dynamics generators

- 'preop' : P = expm(phase*dyn_gen)
- 'postop' : P = expm(dyn_gen*phase)
- 'custom' : Customised phase application

The 'custom' option assumes that the _apply_phase method has been set to a custom function.

Type phase_application

property prop
List of propagators (Qobj) for each timeslot

property 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.

initialize_controls(*amplitudes*, *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

property num_ctrls
calculate the of controls from the length of the control list sets the num_ctrls property, which can be used alternatively subsequently

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.

property dyn_gen_phase
The phasing operator for the symplectic group generators usually refered to as Omega By default this is applied as 'postop' dyn_gen*-Omega If phase_application is 'preop' it is applied as Omega*dyn_gen

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.

reset()
reset any configuration data

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

clear()
clear any temporarily held status data

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

reset()
reset any configuration data and clear any temporarily held status data

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

reset()
reset any configuration data and clear any temporarily held status data

class FidCompTraceDiffApprox(*dynamics*, *params=None*)
As FidCompTraceDiff, except uses the finite difference method to compute approximate gradients

Attributes

epsilon [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

reset()
reset any configuration data and clear any temporarily held status data

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

reset()
reset attributes to default values

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

decay_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

randomize_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 overides 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_per_iter [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_per_update [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_per_update [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

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

level [string] The level of data dumping that will occur.

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

property 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 *qutip.control.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

property dump_all
True if everything (ignoring the summary) is to be dumped

property 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 `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

property dump_all
True if all of the calculation objects are to be dumped

property 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).

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 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

5.2 Functions

5.2.1 Manipulation and Creation of States and Operators

Quantum States

basis(*dimensions*, *n=None*, *offset=None*)
Generates the vector representation of a Fock state.

Parameters

dimensions [int or list of ints] Number of Fock states in Hilbert space. If a list, then the resultant object will be a tensor product over spaces with those dimensions.

n [int or list of ints, optional (default 0 for all dimensions)] Integer corresponding to desired number state, defaults to 0 for all dimensions if omitted. The shape must match `dimensions`, e.g. if `dimensions` is a list, then `n` must either be omitted or a list of equal length.

offset [int or list of ints, optional (default 0 for all dimensions)] The lowest number state that is included in the finite number state representation of the state in the relevant dimension.

Returns

state [*qutip.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]]
>>> basis([2,2,2], [0,1,0])
Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = (8, 1), type = ket
Qobj data =
[[0.]
 [0.]
 [1.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
```

bell_state(*state='00'*)

Returns the selected Bell state:

$$|B_{00}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$
$$|B_{01}\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$$
$$|B_{10}\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$$
$$|B_{11}\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$$

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=None*)

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=None*)

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_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*.

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, list The number of states *nstates*, a dictionary for looking up state indices from a state tuple, and a list containing the state tuples ordered by state indices. state2idx and idx2state are reverses of each other, i.e., state2idx[idx2state[idx]] = idx and idx2state[state2idx[state]] = state.

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.

fock(*dimensions*, *n=None*, *offset=None*)

Bosonic Fock (number) state.

Same as *qutip.states.basis*.

Parameters

dimensions [int or list of ints] Number of Fock states in Hilbert space. If a list, then the resultant object will be a tensor product over spaces with those dimensions.

n [int or list of ints, optional (default 0 for all dimensions)] Integer corresponding to desired number state, defaults to 0 for all dimensions if omitted. The shape must match `dimensions`, e.g. if `dimensions` is a list, then `n` must either be omitted or a list of equal length.

offset [int or list of ints, optional (default 0 for all dimensions)] The lowest number state that is included in the finite number state representation of the state in the relevant dimension.

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(*dimensions*, *n=None*, *offset=None*)

Density matrix representation of a Fock state

Constructed via outer product of *qutip.states.fock*.

Parameters

dimensions [int or list of ints] Number of Fock states in Hilbert space. If a list, then the resultant object will be a tensor product over spaces with those dimensions.

n [int or list of ints, optional (default 0 for all dimensions)] Integer corresponding to desired number state, defaults to 0 for all dimensions if omitted. The shape must match `dimensions`, e.g. if `dimensions` is a list, then `n` must either be omitted or a list of equal length.

offset [int or list of ints, optional (default 0 for all dimensions)] The lowest number state that is included in the finite number state representation of the state in the relevant dimension.

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

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.]
```

(continues on next page)

(continued from previous page)

```
 [ 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]]
```

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.

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=None*)

The projection operator that projects state $|m\rangle$ on state $|n\rangle$.

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:

$$|S\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$$

that is identical to the fourth bell state.

Returns

Bell_state [qobj] $|B_{11}\rangle$ Bell state

spin_coherent(*j*, *theta*, *phi*, *type='ket'*)
Generate the coherent spin state $|\theta, \phi\rangle$.

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

spin_state(*j*, *m*, *type='ket'*)
Generates the spin state $|j, m\rangle$, 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

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 [tuple] The state number tuple corresponding to index *index* in standard enumeration ordering.

state_number_enumerate(*dims*, *excitations=None*)

An iterator that enumerates all the state number tuples (quantum numbers of 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.

excitations [integer (None)] Restrict state space to states with excitation numbers below or equal to this value.

Returns

state_number [tuple] Successive state number tuples 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_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*] The state as a *qutip.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 =
```

(continues on next page)

(continued from previous page)

```
[[ 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]]
```

triplet_states()
Returns a list of the two particle triplet-states:

$$|T_1\rangle = |11\rangle |T_2\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle) |T_3\rangle = |00\rangle$$

Returns

trip_states [list] 2 particle triplet states

w_state(*N=3*)
Returns the N-qubit W-state.

Parameters

N [int (default=3)] Number of qubits in state

Returns

W [qobj] N-qubit W-state

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*.

identity(*dims*)

Identity operator. Alternative name to *qeye*.

Parameters

dimensions [(int) or (list of int) or (list of list of int)] 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. This can produce either *oper* or *super* depending on the passed *dimensions*.

Returns

oper [qobj] Identity operator Qobj.

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.]]]
```

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.

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]]
```

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 for usage information.

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.        ]]
```

qeye(*dimensions*)
Identity operator.

Parameters

dimensions [(int) or (list of int) or (list of list of int)] 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. This can produce either *oper* or *super* depending on the passed *dimensions*.

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.]]
>>> qeye([2,2])
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.]]
```

qutrit_ops()
Operators for a three level system (qutrit).

Returns

opers: array *array* of qutrit operators.

qzero(*dimensions*)
Zero operator.

Parameters

dimensions [(int) or (list of int) or (list of list of int)] 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. This can produce either *oper* or *super* depending on the passed *dimensions*.

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

```
>>> sigmap()
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_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.

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.

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*] 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*] Operator 1.

a2 [*qutip.Qobj*] Operator 2.

z [float/complex] Squeezing parameter.

Returns

oper [*qutip.Qobj*] Squeezing operator.

tunneling(*N*, *m=1*)

Tunneling operator with elements of the form $\sum |N><N+m| + |N+m><N|$.

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.

Quantum Objects

The Quantum Object (Qobj) class, for representing quantum states and operators, and related functions.

dag(*A*)

Adjont operator (dagger) of a quantum object.

Parameters

A [*qutip.Qobj*] Input quantum object.

Returns

oper [*qutip.Qobj*] Adjoint of input operator

Notes

This function is for legacy compatibility only. It is recommended to use the `dag()` Qobj method.

dims (*inpt*)
Returns the dims attribute of a quantum object.

Parameters

inpt [*qutip.Qobj*] Input quantum object.

Returns

dims [list] A `list` of the quantum objects dimensions.

Notes

This function is for legacy compatibility only. Using the *Qobj.dims* attribute is recommended.

isbra (*Q*)
Determines if given quantum object is a bra-vector.

Parameters

Q [*qutip.Qobj*] Quantum object

Returns

isbra [bool] True if Qobj is bra-vector, False otherwise.

Notes

This function is for legacy compatibility only. Using the *Qobj.isbra* attribute is recommended.

Examples

```
>>> psi = basis(5,2)
>>> isket(psi)
False
```

isequal (*A*, *B*, *tol=None*)
Determines if two qobj objects are equal to within given tolerance.

Parameters

A [*qutip.Qobj*] Qobj one

B [*qutip.Qobj*] Qobj two

tol [float] Tolerence for equality to be valid

Returns

isequal [bool] True if qobjs are equal, False otherwise.

Notes

This function is for legacy compatibility only. Instead, it is recommended to use the equality operator of Qobj instances instead: A == B.

isherm(*Q*)

Determines if given operator is Hermitian.

Parameters

Q [*qutip.Qobj*] Quantum object

Returns

isherm [bool] True if operator is Hermitian, False otherwise.

Notes

This function is for legacy compatibility only. Using the *Qobj.isherm* attribute is recommended.

Examples

```
>>> a = destroy(4)
>>> isherm(a)
False
```

isket(*Q*)

Determines if given quantum object is a ket-vector.

Parameters

Q [*qutip.Qobj*] Quantum object

Returns

isket [bool] True if qobj is ket-vector, False otherwise.

Notes

This function is for legacy compatibility only. Using the *Qobj.isket* attribute is recommended.

Examples

```
>>> psi = basis(5,2)
>>> isket(psi)
True
```

isoper(*Q*)

Determines if given quantum object is a operator.

Parameters

Q [*qutip.Qobj*] Quantum object

Returns

isoper [bool] True if Qobj is operator, False otherwise.

Notes

This function is for legacy compatibility only. Using the *Qobj.isoper* attribute is recommended.

Examples

```
>>> a = destroy(5)
>>> isoper(a)
True
```

isoperbra(*Q*)

Determines if given quantum object is an operator in row vector form (operator-bra).

Parameters

Q [*qutip.Qobj*] Quantum object

Returns

isoperbra [bool] True if Qobj is operator-bra, False otherwise.

Notes

This function is for legacy compatibility only. Using the *Qobj.isoperbra* attribute is recommended.

isoperket(*Q*)

Determines if given quantum object is an operator in column vector form (operator-ket).

Parameters

Q [*qutip.Qobj*] Quantum object

Returns

isoperket [bool] True if Qobj is operator-ket, False otherwise.

Notes

This function is for legacy compatibility only. Using the *Qobj.isoperket* attribute is recommended.

issuper(*Q*)

Determines if given quantum object is a super-operator.

Parameters

Q [*qutip.Qobj*] Quantum object

Returns

issuper [bool] True if Qobj is superoperator, False otherwise.

Notes

This function is for legacy compatibility only. Using the *Qobj.issuper* attribute is recommended.

ptrace(*Q*, *sel*)

Partial trace of the Qobj with selected components remaining.

Parameters

Q [*qutip.Qobj*] Composite quantum object.

sel [int/list] An `int` or `list` of components to keep after partial trace.

Returns

oper [*qutip.Qobj*] Quantum object representing partial trace with selected components remaining.

Notes

This function is for legacy compatibility only. It is recommended to use the `ptrace()` Qobj method.

qobj_list_evaluate(*qobj_list*, *t*, *args*)
Deprecated: See Qobj.evaluate

shape(*inpt*)
Returns the shape attribute of a quantum object.

Parameters

inpt [*qutip.Qobj*] Input quantum object.

Returns

shape [list] A `list` of the quantum objects shape.

Notes

This function is for legacy compatibility only. Using the *Qobj.shape* attribute is recommended.

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*, *seed=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]].

seed [int] Seed for the random number generator.

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*, *seed=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*, *seed=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*, *seed=None*)

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).

seed [int] seed for the random number generator

Returns

oper [qobj] NxN Hermitian quantum operator.

Notes

If given a list/ndarray as input 'N', this function returns a random Hermitian object with eigenvalues given in the list/ndarray. This is accomplished via complex Jacobi rotations. While this method is ~50% faster than the corresponding (real only) Matlab code, it should not be repeatedly used for generating matrices larger than ~1000x1000.

rand_ket (*N=None*, *density=1*, *dims=None*, *seed=None*)

Creates a random Nx1 sparse ket vector.

Parameters

N [int] Number of rows for output quantum vector. If None, N is deduced from dims.

density [float] Density between [0,1] of output ket state.

dims [list] Dimensions of quantum object. Used for specifying tensor structure. Default is dims=[[N],[1]].

seed [int] Seed for the random number generator.

Returns

oper [qobj] Nx1 ket quantum state vector.

Raises

ValueError If neither *N* or *dims* are specified.

rand_ket_haar(*N=None*, *dims=None*, *seed=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. If None, N is deduced from dims.

dims [list of ints, or None] Dimensions of the resultant quantum object. If None, [[N],[1]] is used.

Returns

psi [Qobj] A random state vector drawn from the Haar measure.

Raises

ValueError If neither *N* or *dims* are specified.

rand_stochastic(*N*, *density=0.75*, *kind='left'*, *dims=None*, *seed=None*)

Generates a random stochastic matrix.

Parameters

N [int] Dimension of matrix.

density [float] Density between [0,1] of output density matrix.

kind [str (Default = 'left')] Generate 'left' or 'right' stochastic matrix.

dims [list] Dimensions of quantum object. Used for specifying tensor structure. Default is dims=[[N],[N]].

Returns

oper [qobj] Quantum operator form of stochastic matrix.

rand_super(*N=5*, *dims=None*, *seed=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*, *seed=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.

rand_unitary(*N*, *density=0.75*, *dims=None*, *seed=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*, *seed=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.

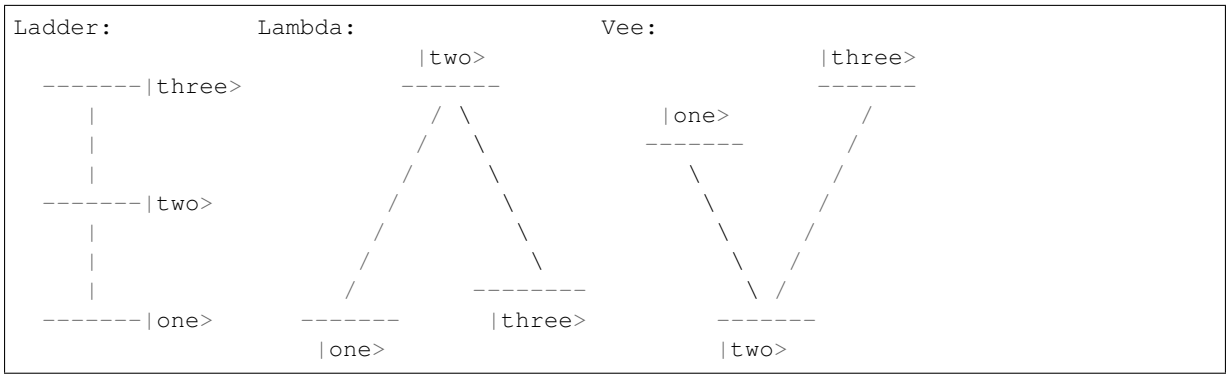
dims [list of lists of int, or None] Dimensions of quantum object. Used for specifying tensor structure. Default is dims=[[N],[N]].

Returns

U [Qobj] Unitary of dims `[[dim], [dim]]` drawn from the Haar measure.

Three-Level Atoms

This module provides functions that are useful for simulating the three level atom with QuTiP. A three level atom (qutrit) has three states, which are linked by dipole transitions so that 1 <-> 2 <-> 3. Depending on there relative energies they are in the ladder, lambda or vee configuration. The structure of the relevant operators is the same for any of the three configurations:



References

The naming of qutip operators follows the convention in [1] .

Notes

Contributed by Markus Baden, Oct. 07, 2011

three_level_basis()
Basis states for a three level atom.

Returns

states [array] *array* of three level atom basis vectors.

three_level_ops()
Operators for a three level system (qutrit)

Returns

ops [array] *array* of three level operators.

Superoperators and Liouvillians

lindblad_dissipator(*a*, *b=None*, *data_only=False*, *chi=None*)
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 or QobjEvo] Left part of collapse operator.

b [Qobj or QobjEvo (optional)] Right part of collapse operator. If not specified, b defaults to a.

Returns

D [qobj, QobjEvo] Lindblad dissipator superoperator.

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 or QobjEvo] System Hamiltonian.

c_ops [array_like of Qobj or QobjEvo] A `list` or `array` of collapse operators.

Returns

L [Qobj or QobjEvo] Liouvillian superoperator.

operator_to_vector(*op*)
Create a vector representation given a quantum operator in matrix form. The passed object should have a `Qobj.type` of 'oper' or 'super'; this function is not designed for general-purpose matrix reshaping.

Parameters

op [Qobj or QobjEvo] Quantum operator in matrix form. This must have a type of 'oper' or 'super'.

Returns

Qobj or QobjEvo The same object, but re-cast into a column-stacked-vector form of type 'operator-ket'. The output is the same type as the passed object.

spost(*A*)
Superoperator formed from post-multiplication by operator A

Parameters

A [Qobj or QobjEvo] Quantum operator for post multiplication.

Returns

super [Qobj or QobjEvo] Superoperator formed from input qauntum object.

spre(*A*)
Superoperator formed from pre-multiplication by operator A.

Parameters

A [Qobj or QobjEvo] Quantum operator for pre-multiplication.

Returns

super :Qobj or QobjEvo 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 or QobjEvo] Quantum operator for pre-multiplication.

B [Qobj or QobjEvo] Quantum operator for post-multiplication.

Returns

super [Qobj or QobjEvo] Superoperator formed from input quantum objects.

vector_to_operator(*op*)
Create a matrix representation given a quantum operator in vector form. The passed object should have a `Qobj.type` of 'operator-ket'; this function is not designed for general-purpose matrix reshaping.

Parameters

op [Qobj or QobjEvo] Quantum operator in column-stacked-vector form. This must have a type of 'operator-ket'.

Returns

Qobj or QobjEvo The same object, but re-cast into "standard" operator form. The output is the same type as the passed object.

Superoperator Representations

This module implements transformations between superoperator representations, including supermatrix, Kraus, Choi and Chi (process) matrix formalisms.

chi_to_choi(*q_oper*)
Converts a Chi matrix to a Choi matrix.

NOTE: this is only supported for qubits right now. Need to extend to Heisenberg-Weyl for other subsystem dimensions.

choi_to_chi(*q_oper*)
Converts a Choi matrix to a Chi matrix in the Pauli basis.

NOTE: this is only supported for qubits right now. Need to extend to Heisenberg-Weyl for other subsystem dimensions.

choi_to_kraus(*q_oper*, *tol=1e-09*)
Takes a Choi matrix and returns a list of Kraus operators. TODO: Create a new class structure for quantum channels, perhaps as a strict sub-class of Qobj.

choi_to_super(*q_oper*)
Takes a Choi matrix to a superoperator TODO: Sanitize input, Abstract-ify application of channels to states

kraus_to_choi(*kraus_list*)
Takes a list of Kraus operators and returns the Choi matrix for the channel represented by the Kraus operators in *kraus_list*

kraus_to_super(*kraus_list*)
Converts a list of Kraus operators and returns a super operator.

super_to_choi(*q_oper*)
Takes a superoperator to a Choi matrix TODO: Sanitize input, incorporate as method on Qobj if type=='super'

to_chi(*q_oper*)
Converts a Qobj representing a quantum map to a representation as a chi (process) matrix in the Pauli basis, 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 Chi representation. If `q_oper` is `type="oper"`, then it is taken to act by conjugation, such that `to_chi(A) == to_chi(sprepost(A, A.dag()))`.

Returns

chi [Qobj] A quantum object representing the same map as `q_oper`, such that `chi.superrep == "chi"`.

Raises

TypeError: if the given quantum object is not a map, or cannot be converted to Chi representation.

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_kraus(*q_oper*, *tol=1e-09*)
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]`.

tol [Float] Optional threshold parameter for eigenvalues/Kraus ops to be discarded. The default is to=1e-9.

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.

to_stinespring(*q_oper*)

Converts a Qobj representing a quantum map $Lambda$ to a pair of partial isometries A and B such that $Lambda(X) = Tr_2(A X B^dagger)$ for all inputs X, where the partial trace is taken over a a new index on the output dimensions of A and B.

For completely positive inputs, A will always equal B up to precision errors.

Parameters

q_oper [Qobj] Superoperator to be converted to a Stinespring pair.

Returns

A, B [Qobj] Quantum objects representing each of the Stinespring matrices for the input Qobj.

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.

Operators and Superoperator Dimensions

Internal use module for manipulating dims specifications.

collapse_dims_oper(*dims*)

Given the dimensions specifications for a ket-, bra- or oper-type Qobj, returns a dimensions specification describing the same shape by collapsing all composite systems. For instance, the bra-type dimensions specification `[[2, 3], [1]]` collapses to `[[6], [1]]`.

Parameters

dims [list of lists of ints] Dimensions specifications to be collapsed.

Returns

collapsed_dims [list of lists of ints] Collapsed dimensions specification describing the same shape such that `len(collapsed_dims[0]) == len(collapsed_dims[1]) == 1`.

collapse_dims_super(*dims*)

Given the dimensions specifications for an operator-ket-, operator-bra- or super-type Qobj, returns a dimensions specification describing the same shape by collapsing all composite systems. For instance, the super-type dimensions specification `[[[2, 3], [2, 3]], [[2, 3], [2, 3]]]` collapses to `[[[6], [6]], [[6], [6]]]`.

Parameters

dims [list of lists of ints] Dimensions specifications to be collapsed.

Returns

collapsed_dims [list of lists of ints] Collapsed dimensions specification describing the same shape such that `len(collapsed_dims[i][j]) == 1` for `i` and `j` in `range(2)`.

deep_remove(*l*, **what*)

Removes scalars from all levels of a nested list.

Given a list containing a mix of scalars and lists, returns a list of the same structure, but where one or more scalars have been removed.

Examples

```
>>> deep_remove([[[[0, 1, 2]], [3, 4], [5], [6, 7]]], 0, 5)
[[[[1, 2]], [3, 4], [], [6, 7]]]
```

dims_idxs_to_tensor_idxs(*dims*, *indices*)

Given the dims of a Qobj instance, and some indices into dims, returns the corresponding tensor indices. This helps resolve, for instance, that column-stacking for superoperators, oper-ket and oper-bra implies that the input and output tensor indices are reversed from their order in dims.

Parameters

dims [list] Dimensions specification for a Qobj.

indices [int, list or tuple] Indices to convert to tensor indices. Can be specified as a single index, or as a collection of indices. In the latter case, this can be nested arbitrarily deep. For instance, [0, [0, (2, 3)]].

Returns

tens_indices [int, list or tuple] Container of the same structure as indices containing the tensor indices for each element of indices.

dims_to_tensor_perm(*dims*)

Given the dims of a Qobj instance, returns a list representing a permutation from the flattening of that dims specification to the corresponding tensor indices.

Parameters

dims [list] Dimensions specification for a Qobj.

Returns

perm [list] A list such that `data[flatten(dims)[idx]]` gives the index of the tensor `data` corresponding to the `idx``th dimension of ``dims`.

dims_to_tensor_shape(*dims*)

Given the dims of a Qobj instance, returns the shape of the corresponding tensor. This helps, for instance, resolve the column-stacking convention for superoperators.

Parameters

dims [list] Dimensions specification for a Qobj.

Returns

tensor_shape [tuple] NumPy shape of the corresponding tensor.

enumerate_flat(*l*)
Labels the indices at which scalars occur in a flattened list.

Given a list containing a mix of scalars and lists, returns a list of the same structure, where each scalar has been replaced by an index into the flattened list.

Examples

```
>>> print(enumerate_flat([[[10], [20, 30]], 40]))
[[[0], [1, 2]], 3]
```

flatten(*l*)
Flattens a list of lists to the first level.

Given a list containing a mix of scalars and lists, flattens down to a list of the scalars within the original list.

Examples

```
>>> flatten([[[0], 1], 2])
[0, 1, 2]
```

is_scalar(*dims*)
Returns True if a dims specification is effectively a scalar (has dimension 1).

unflatten(*l*, *idxs*)
Unflattens a list by a given structure.

Given a list of scalars and a deep list of indices as produced by *flatten*, returns an "unflattened" form of the list. This perfectly inverts *flatten*.

Examples

```
>>> l = [[[10, 20, 30], [40, 50, 60]], [[70, 80, 90], [100, 110, 120]]]
>>> idxs = enumerate_flat(l)
>>> unflatten(flatten(l), idxs) == l
True
```

5.2.2 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
    True
```

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.

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))`.

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.

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]]
```

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

[1]

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_relative(*rho*, *sigma*, *base=2.718281828459045*, *sparse=False*, *tol=1e-12*)
Calculates the relative entropy S(rho||sigma) between two density matrices.

Parameters

rho [*qutip.Qobj*] First density matrix (or ket which will be converted to a density matrix).

sigma [*qutip.Qobj*] Second density matrix (or ket which will be converted to a density matrix).

base [{e,2}] Base of logarithm. Defaults to e.

sparse [bool] Flag to use sparse solver when determining the eigenvectors of the density matrices. Defaults to False.

tol [float] Tolerance to use to detect 0 eigenvalues or dot producted between eigenvectors. Defaults to 1e-12.

Returns

rel_ent [float] Value of relative entropy. Guaranteed to be greater than zero and should equal zero only when rho and sigma are identical.

References

See Nielsen & Chuang, "Quantum Computation and Quantum Information", Section 11.3.1, pg. 511 for a detailed explanation of quantum relative entropy.

Examples

First we define two density matrices:

```
>>> rho = qutip.ket2dm(qutip.ket("00"))
>>> sigma = rho + qutip.ket2dm(qutip.ket("01"))
>>> sigma = sigma.unit()
```

Then we calculate their relative entropy using base 2 (i.e. `log2`) and base e (i.e. `log`).

```
>>> qutip.entropy_relative(rho, sigma, base=2)
1.0
>>> qutip.entropy_relative(rho, sigma)
0.6931471805599453
```

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.

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.

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.

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.

dnorm(*A*, *B=None*, *solver='CVXOPT'*, *verbose=False*, *force_solve=False*, *sparse=True*)

Calculates the diamond norm of the quantum map q_oper, using the simplified semidefinite program of [Wat13].

The diamond norm SDP is solved by using CVXPY.

Parameters

A [Qobj] Quantum map to take the diamond norm of.

B [Qobj or None] If provided, the diamond norm of $A - B$ is taken instead.

solver [str] Solver to use with CVXPY. One of "CVXOPT" (default) or "SCS". The latter tends to be significantly faster, but somewhat less accurate.

verbose [bool] If True, prints additional information about the solution.

force_solve [bool] If True, forces dnorm to solve the associated SDP, even if a special case is known for the argument.

sparse [bool] Whether to use sparse matrices in the convex optimisation problem. Default True.

Returns

dn [float] Diamond norm of q_oper.

Raises

ImportError If CVXPY cannot be imported.

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)
>>> np.testing.assert_almost_equal(fidelity(x,y), 0.24104350624628332)
```

hellinger_dist (*A*, *B*, *sparse=False*, *tol=0*)

Calculates the quantum Hellinger distance between two density matrices.

Formula: hellinger_dist(A, B) = sqrt(2-2*Tr(sqrt(A)*sqrt(B)))

See: D. Spehner, F. Illuminati, M. Orszag, and W. Roga, "Geometric measures of quantum correlations with Bures and Hellinger distances" arXiv:1611.03449

Parameters

A [*qutip.Qobj*] Density matrix or state vector.

B [*qutip.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

hellinger_dist [float] Quantum Hellinger distance between A and B. Ranges from 0 to sqrt(2).

Examples

```
>>> x=fock_dm(5,3)
>>> y=coherent_dm(5,1)
>>> np.testing.assert_almost_equal(hellinger_dist(x,y), 1.3725145002591095)
```

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).

process_fidelity (*U1*, *U2*, *normalize=True*)

Calculate the process fidelity given two process operators.

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)
>>> np.testing.assert_almost_equal(tracedist(x,y), 0.9705143161472971)
```

unitarity(*oper*)

Returns the unitarity of a quantum map, defined as the Frobenius norm of the unital block of that map's superoperator representation.

Parameters

oper [Qobj] Quantum map under consideration.

Returns

u [float] Unitarity of `oper`.

Continuous 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.

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*, *g=1.4142135623730951*)

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.

g [float] Scaling factor for *a = 0.5 * g * (x + iy)*, default *g = sqrt(2)*. The value of *g* is related to the value of *hbar* in the commutation relation *[x, y] = i * hbar* via *hbar=2/g ** 2* giving the default value *hbar=1*.

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.

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.

logarithmic_negativity(*V*, *g=1.4142135623730951*)

Calculates the logarithmic negativity given a symmetrized covariance matrix, see *qutip.continuous_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.

g [float] Scaling factor for *a = 0.5 * g * (x + iy)*, default *g = sqrt(2)*. The value of *g* is related to the value of *hbar* in the commutation relation *[x, y] = i * hbar* via *hbar=2/g ** 2* giving the default value *hbar=1*.

Returns

N [float] The logarithmic negativity for the two-mode Gaussian state that is described by the the Wigner covariance matrix V.

wigner_covariance_matrix(*a1=None*, *a2=None*, *R=None*, *rho=None*, *g=1.4142135623730951*)

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.

g [float] Scaling factor for *a = 0.5 * g * (x + iy)*, default *g = sqrt(2)*. The value of *g* is related to the value of *hbar* in the commutation relation *[x, y] = i * hbar* via *hbar=2/g ** 2* giving the default value *hbar=1*.

Returns

cov_mat [ndarray] A 2-dimensional array of covariance values.

5.2.3 Measurement

Measurement of quantum states

Module for measuring quantum objects.

measure(*state*, *ops*, *targets=None*)

A dispatch method that provides measurement results handling both observable style measurements and projector style measurements (POVMs and PVMs).

For return signatures, please check:

- *measure_observable* for observable measurements.
- *measure_povm* for POVM measurements.

Parameters

state [*Qobj*] The ket or density matrix specifying the state to measure.

ops [*Qobj* or list of *Qobj*]

- measurement observable (*Qobj*); or
- list of measurement operators M_i or kets (list of *Qobj*) Either:
  1. specifying a POVM s.t. $E_i = M_i^\dagger M_i$
  2. projection operators if ops correspond to projectors (s.t. $E_i = M_i^\dagger = M_i$)
  3. kets (transformed to projectors)

targets [list of ints, optional] Specifies a list of target "qubit" indices on which to apply the measurement using *qutip.qip.operations.gates.expand_operator* to expand ops into full dimension.

measure_observable(*state*, *op*, *targets=None*)

Perform a measurement specified by an operator on the given state.

This function simulates the classic quantum measurement described in many introductory texts on quantum mechanics. The measurement collapses the state to one of the eigenstates of the given operator and the result of the measurement is the corresponding eigenvalue.

Parameters

state [*Qobj*] The ket or density matrix specifying the state to measure.

op [*Qobj*] The measurement operator.

targets [list of ints, optional] Specifies a list of target "qubit" indices on which to apply the measurement using *qutip.qip.operations.gates.expand_operator* to expand op into full dimension.

Returns

measured_value [float] The result of the measurement (one of the eigenvalues of op).

state [*Qobj*] The new state (a ket if a ket was given, otherwise a density matrix).

Examples

Measure the z-component of the spin of the spin-up basis state:

```
>>> measure_observable(basis(2, 0), sigmaz())
(1.0, Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[-1.]
 [ 0.]])
```

Since the spin-up basis is an eigenstate of sigmaz, this measurement always returns 1 as the measurement result (the eigenvalue of the spin-up basis) and the original state (up to a global phase).

Measure the x-component of the spin of the spin-down basis state:

```
>>> measure_observable(basis(2, 1), sigmax())
(-1.0, Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[-0.70710678]
 [ 0.70710678]])
```

This measurement returns 1 fifty percent of the time and -1 the other fifty percent of the time. The new state returned is the corresponding eigenstate of sigmax.

One may also perform a measurement on a density matrix. Below we perform the same measurement as above, but on the density matrix representing the pure spin-down state:

```
>>> measure_observable(ket2dm(basis(2, 1)), sigmax())
(-1.0, Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper
Qobj data =
[[ 0.5 -0.5]
 [-0.5  0.5]])
```

The measurement result is the same, but the new state is returned as a density matrix.

measure_povm(*state*, *ops*, *targets=None*)

Perform a measurement specified by list of POVMs.

This function simulates a POVM measurement. The measurement collapses the state to one of the resultant states of the measurement and returns the index of the operator corresponding to the collapsed state as well as the collapsed state.

Parameters

state [*Qobj*] The ket or density matrix specifying the state to measure.

ops [list of *Qobj*] List of measurement operators M_i or kets. Either:

1. specifying a POVM s.t. $E_i = M_i^\dagger M_i$
2. projection operators if ops correspond to projectors (s.t. $E_i = M_i^\dagger = M_i$)
3. kets (transformed to projectors)

targets [list of ints, optional] Specifies a list of target "qubit" indices on which to apply the measurement using *qutip.qip.operations.gates.expand_operator* to expand `ops` into full dimension.

Returns

index [float] The resultant index of the measurement.

state [*Qobj*] The new state (a ket if a ket was given, otherwise a density matrix).

measurement_statistics(*state*, *ops*, *targets=None*)

A dispatch method that provides measurement statistics handling both observable style measurements and projector style measurements(POVMs and PVMs).

For return signatures, please check:

- *measurement_statistics_observable* for observable measurements.
- *measurement_statistics_povm* for POVM measurements.

Parameters

state [*Qobj*] The ket or density matrix specifying the state to measure.

ops [*Qobj* or list of *Qobj*]

- measurement observable (:class:.Qobj); or
- list of measurement operators M_i or kets (list of *Qobj*) Either:
  1. specifying a POVM s.t. $E_i = M_i^\dagger * M_i$
  2. projection operators if ops correspond to projectors (s.t. $E_i = M_i^\dagger = M_i$)
  3. kets (transformed to projectors)

targets [list of ints, optional] Specifies a list of target "qubit" indices on which to apply the measurement using *qutip.qip.operations.gates.expand_operator* to expand ops into full dimension.

measurement_statistics_observable(*state*, *op*, *targets=None*)

Return the measurement eigenvalues, eigenstates (or projectors) and measurement probabilities for the given state and measurement operator.

Parameters

state [*Qobj*] The ket or density matrix specifying the state to measure.

op [*Qobj*] The measurement operator.

targets [list of ints, optional] Specifies a list of targets "qubit" indices on which to apply the measurement using *qutip.qip.operations.gates.expand_operator* to expand op into full dimension.

Returns

eigenvalues: list of float The list of eigenvalues of the measurement operator.

eigenstates_or_projectors: list of *Qobj* If the state was a ket, return the eigenstates of the measurement operator. Otherwise return the projectors onto the eigenstates.

probabilities: list of float The probability of measuring the state as being in the corresponding eigenstate (and the measurement result being the corresponding eigenvalue).

measurement_statistics_povm(*state*, *ops*, *targets=None*)

Returns measurement statistics (resultant states and probabilities) for a measurement specified by a set of positive operator valued measurements on a specified ket or density matrix.

Parameters

state [*Qobj*] The ket or density matrix specifying the state to measure.

ops [list of *Qobj*] List of measurement operators M_i or kets. Either:

1. specifying a POVM s.t. $E_i = M_i^\dagger M_i$
2. projection operators if ops correspond to projectors (s.t. $E_i = M_i^\dagger = M_i$)
3. kets (transformed to projectors)

targets [list of ints, optional] Specifies a list of target "qubit" indices on which to apply the measurement using qutip.qip.operations.gates.expand_operator to expand ops into full dimension.

Returns

collapsed_states [list of *Qobj*] The collapsed states obtained after measuring the qubits and obtaining the qubit specified by the target in the state specified by the index.

probabilities [list of floats] The probability of measuring a state in a the state specified by the index.

5.2.4 Dynamics and Time-Evolution

Schrödinger Equation

This module provides solvers for the unitary Schrodinger equation.

sesolve(*H*, *psi0*, *tlist*, *e_ops=None*, *args=None*, *options=None*, *progress_bar=None*, *_safe_mode=True*)

Schrödinger equation evolution of a state vector or unitary matrix for a given Hamiltonian.

Evolve the state vector (`psi0`) using a given Hamiltonian (`H`), by integrating the set of ordinary differential equations that define the system. Alternatively evolve a unitary matrix in solving the Schrodinger operator equation.

The output is either the state vector or unitary matrix 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. `e_ops` cannot be used in conjunction with solving the Schrodinger operator equation

Parameters

H [*Qobj*, *QobjEvo*, list, or callable] System Hamiltonian as a *Qobj* , list of *Qobj* and coefficient, *QobjEvo*, or a callback function for time-dependent Hamiltonians. List format and options can be found in QobjEvo's description.

psi0 [*Qobj*] Initial state vector (ket) or initial unitary operator `psi0 = U`.

tlist [array_like of float] List of times for t.

e_ops [None / list / callback function, optional] A list of operators as *Qobj* and/or callable functions (can be mixed) or a single callable function. For callable functions, they are called as `f(t, state)` and return the expectation value. A single callback's expectation value can be any type, but a callback part of a list must return a number as the expectation value. For operators, the result's expect will be computed by *qutip.expect* when the state is a `ket`. For operator evolution, the overlap is computed by:

```
(e_ops[i].dag() * op(t)).tr()
```

args [dict, optional] Dictionary of scope parameters for time-dependent Hamiltonians.

options [*Options*, optional] Options for the ODE solver.

progress_bar [`BaseProgressBar`, optional] Optional instance of `BaseProgressBar`, or a subclass thereof, for showing the progress of the simulation.

Returns

output: ***Result*** An instance of the class *Options*, which contains either an array of expectation values for the times specified by `tlist`, or an array or state vectors 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=None*, *e_ops=None*, *args=None*, *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 or Liouvillian (*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 the solution of master equations that are not in standard Lindblad form.

Time-dependent operators

For time-dependent problems, *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*.

Alternatively, *H* (but not *c_ops*) can be a callback function with the signature *f(t, args) -> Qobj* (*callback format*), which can return the Hamiltonian or Liouvillian superoperator at any point in time. If the equation cannot be put in standard Lindblad form, then this time-dependence format must be used.

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 their 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 without further transformation. This allows for using mesolve for solving master equations that are not in standard Lindblad form.

Note: On using callback functions: 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 a 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 [None / list of *qutip.Qobj*] single collapse operator, or list of collapse operators, or a list of Liouvillian superoperators.

e_ops [None / list / callback function, optional] A list of operators as *Qobj* and/or callable functions (can be mixed) or a single callable function. For operators, the result's expect will be computed by *qutip.expect*. For callable functions, they are called as `f(t, state)` and return the expectation value. A single callback's expectation value can be any type, but a callback part of a list must return a number as the expectation value.

args [None / *dictionary*] dictionary of parameters for time-dependent Hamiltonians and collapse operators.

options [None / *qutip.solver.Options*] with options for the solver.

progress_bar [None / 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 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=0*, *args={}*, *options=None*, *progress_bar=True*, *map_func=<function parallel_map>*, *map_kwargs={}*, *_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:

```
def H1_coeff(t,args):
    return sin(args['w']*t)

def C1_coeff(t,args):
    return exp(-args['a']*t)
```

(continues on next page)

(continued from previous page)

```
H = [H0, [H1, H1_coeff]]

c_ops = [C0, [C1, C1_coeff]]

args={'a': A, 'w': W}
```

or in String (Cython) format we could write:

```
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*, `list`] 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 [*qutip.Qobj*, `list`] single collapse operator or a `list` of collapse operators.

e_ops [*qutip.Qobj*, `list`] single operator as Qobj or `list` or equivalent of Qobj 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*).

Deprecated since version 4.6.0: *essolve* will be removed in QuTiP 5. Please use *sesolve* or *mesolve* for general-purpose integration of the Schroedinger/Lindblad master equation. This will likely be faster than *essolve* for you.

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*.

Deprecated since version 4.6.0: *ode2es* will be removed in QuTiP 5. Please use *qutip.Qobj.eigenstates* to get the eigenstates and -values, and use *QobjEvo* for general time-dependence.

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.

Krylov Subspace Solver

krylovsolve(*H: qutip.qobj.Qobj*, *psi0: qutip.qobj.Qobj*, *tlist: numpy.array*, *krylov_dim: int*, *e_ops=None*, *options=None*, *progress_bar: Optional[bool] = None*, *sparse: bool = False*)

Time evolution of state vectors for time independent Hamiltonians. Evolve the state vector ("psi0") finding an approximation for the time evolution operator of Hamiltonian ("H") by obtaining the projection of the time evolution operator on a set of small dimensional Krylov subspaces (m << dim(H)).

The output is either the state vector or the expectation values of supplied operators ("e_ops") at arbitrary points at ("tlist").

Additional options

Additional options to krylovsolve can be set with the following:

- "store_states": stores states even though expectation values are requested via the "e_ops" argument.
- "store_final_state": store final state even though expectation values are requested via the "e_ops" argument.

Parameters

H [*qutip.Qobj*] System Hamiltonian.

psi0 [:class: *qutip.Qobj*] Initial state vector (ket).

tlist [None / *list* / *array*] List of times on which to evolve the initial state. If None, nothing happens but the code won't break.

krylov_dim: int Dimension of Krylov approximation subspaces used for the time evolution approximation.

e_ops [None / list of *qutip.Qobj*] Single operator or list of operators for which to evaluate expectation values.

options [Options]

Instance of ODE solver options, as well as krylov parameters.

atol: controls (approximately) the error desired for the final solution. (Defaults to 1e-8)

nsteps: maximum number of krylov's internal number of Lanczos iterations. (Defaults to 10000)

progress_bar [None / BaseProgressBar] Optional instance of BaseProgressBar, or a subclass thereof, for showing the progress of the simulation.

sparse [bool (default False)] Use np.array to represent system Hamiltonians. If True, scipy sparse arrays are used instead.

Returns

result: *qutip.solver.Result* An instance of the class *qutip.solver.Result*, which contains either an *array result.expect* of expectation values for the times *tlist*, or an *array result.states* of state vectors corresponding to the times *tlist* [if *e_ops* is an empty list].

Bloch-Redfield Master Equation

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.solver.Options*] Options for the ODE solver.

Returns

output: *qutip.solver.Result* An instance of the class *qutip.solver.Result*, which contains either an *array* of expectation values for the times specified by *tlist*.

bloch_redfield_tensor()

Calculates the time-independent Bloch-Redfield tensor for a system given a set of operators and corresponding spectral functions that describes the system's couplingto its environment.

Parameters

H [*qutip.qobj*]

System Hamiltonian.

a_ops [list] Nested list of system operators that couple to the environment, and the corresponding bath spectra represented as Python functions.

spectra_cb [list] Depreciated.

c_ops [list] List of system collapse operators.

use_secular [bool {True, False}] Flag that indicates if the secular approximation should be used.

sec_cutoff [float {0.1}] Threshold for secular approximation.

atol [float {qutip.settings.atol}] Threshold for removing small parameters.

Returns

R, kets: *qutip.Qobj*, list of *qutip.Qobj* R is the Bloch-Redfield tensor and kets is a list eigenstates of the Hamiltonian.

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*, *verbose=False*)

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] 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*.

Floquet States and Floquet-Markov Master Equation

floquet_basis_transform(*f_modes*, *f_energies*, *rho0*)
Make a basis transform that takes rho0 from the floquet basis to the computational basis.

floquet_markov_mesolve(*R*, *rho0*, *tlist*, *e_ops*, *options=None*, *floquet_basis=True*, *f_modes_0=None*, *f_modes_table_t=None*, *f_energies=None*, *T=None*)
Solve the dynamics for the system using the Floquet-Markov master equation.

Note: It is important to understand in which frame and basis the results are returned here.

Parameters

R [array] The Floquet-Markov master equation tensor *R*.

rho0 [*qutip.qobj*] Initial density matrix. If `f_modes_0` is not passed, this density matrix is assumed to be in the Floquet picture.

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.solver.Options*] options for the ODE solver.

floquet_basis: bool, True If `True`, states and expectation values will be returned in the Floquet basis. If `False`, a transformation will be made to the computational basis; this will be in the lab frame if `f_modes_table`, `T` and ``f_energies`` are all supplied, or the interaction picture (defined purely be f_modes_0) if they are not.

f_modes_0 [list of *qutip.qobj* (kets), optional] A list of initial Floquet modes, used to transform the given starting density matrix into the Floquet basis. If this is not passed, it is assumed that `rho` is already in the Floquet basis.

f_modes_table_t [nested list of *qutip.qobj* (kets), optional] A lookup-table of Floquet modes at times precalculated by *qutip.floquet.floquet_modes_table*. Necessary if `floquet_basis` is `False` and the transformation should be made back to the lab frame.

f_energies [array_like of float, optional] The precalculated Floquet quasienergies. Necessary if `floquet_basis` is `False` and the transformation should be made back to the lab frame.

T [float, optional] The time period of driving. Necessary if `floquet_basis` is `False` and the transformation should be made back to the lab frame.

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*.

floquet_master_equation_rates(*f_modes_0*, *f_energies*, *c_op*, *H*, *T*, *args*, *J_cb*, *w_th*, *kmax=5*, *f_modes_table_t=None*)

Calculate the rates and matrix elements for the Floquet-Markov master equation.

Parameters

f_modes_0 [list of *qutip.qobj* (kets)] A list of initial Floquet modes.

f_energies [array] The Floquet energies.

c_op [*qutip.qobj*] The collapse operators describing the dissipation.

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.

J_cb [callback functions] A callback function that computes the noise power spectrum, as a function of frequency, associated with the collapse operator *c_op*.

w_th [float] The temperature in units of frequency.

kmax [int] The truncation of the number of sidebands (default 5).

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* (optional).

options [*qutip.solver.Options*] options for the ODE solver.

Returns

output [list] A list (Delta, X, Gamma, A) containing the matrices Delta, X, Gamma and A used in the construction of the Floquet-Markov master equation.

floquet_master_equation_steadystate(*H*, *A*)

Returns the steadystate density matrix (in the floquet basis!) for the Floquet-Markov master equation.

floquet_modes(*H*, *T*, *args=None*, *sort=False*, *U=None*, *options=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*.

options [*qutip.solver.Options*] options for the ODE solver. For the propagator U.

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*, *options=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.

options [*qutip.solver.Options*] options for the ODE solver. For the propagator.

Returns

output [list of kets] The Floquet modes as kets at time t

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_modes_table (*f_modes_0*, *f_energies*, *tlist*, *H*, *T*, *args=None*, *options=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

options [*qutip.solver.Options*] options for the ODE solver.

Returns

output [nested list] A nested list of Floquet modes as kets for each time in *tlist*

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.

floquet_states (*f_modes_t*, *f_energies*, *t*)
Evaluate the floquet states at time t given the Floquet modes at that time.

Parameters

f_modes_t [list of *qutip.qobj* (kets)] A list of Floquet modes for time t.

f_energies [array] The Floquet energies.

t [float] The time for which to evaluate the Floquet states.

Returns

output [list] A list of Floquet states for the time t.

floquet_states_t (*f_modes_0*, *f_energies*, *t*, *H*, *T*, *args=None*, *options=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.

options [*qutip.solver.Options*] options for the ODE solver.

Returns

output [list] A list of Floquet states for the time t.

floquet_wavefunction (*f_modes_t*, *f_energies*, *f_coeff*, *t*)
Evaluate the wavefunction for a time t using the Floquet state decompositon, given the Floquet modes at time *t*.

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.

Returns

output [*qutip.qobj*] The wavefunction for the time t.

floquet_wavefunction_t (*f_modes_0*, *f_energies*, *f_coeff*, *t*, *H*, *T*, *args=None*, *options=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.

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*, *options_modes=None*)

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*] options for the ODE solver. For solving the master equation.

floquet_basis [bool] Will return results in Floquet basis or computational basis (optional).

k_max [int] The truncation of the number of sidebands (default 5).

options_modes [*qutip.solver.Options*] options for the ODE solver. For computing Floquet modes.

Returns

output [*qutip.solver.Result*] An instance of the class *qutip.solver.Result*, which contains either an *array* of expectation values for the times specified by *tlist*.

fsesolve(*H*, *psi0*, *tlist*, *e_ops=[]*, *T=None*, *args={}*, *Tsteps=100*, *options_modes=None*)
Solve the Schrodinger equation using the Floquet formalism.

Parameters

H [*qutip.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.

options_modes [*qutip.solver.Options*] options for the ODE solver.

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

general_stochastic(*state0*, *times*, *d1*, *d2*, *e_ops=[]*, *m_ops=[]*, *_safe_mode=True*, *len_d2=1*, *args={}*, ***kwargs*)
Solve stochastic general equation. Dispatch to specific solvers depending on the value of the *solver* keyword argument.

Parameters

state0 [*qutip.Qobj*] Initial state vector (ket) or density matrix as a vector.

times [*list / array*] List of times for *t*. Must be uniformly spaced.

d1 [function, callable class] Function representing the deterministic evolution of the system.

def d1(time (double), state (as a np.array vector)): return 1d np.array

d2 [function, callable class] Function representing the stochastic evolution of the system.

def d2(time (double), state (as a np.array vector)): return 2d np.array (N_sc_ops, len(state0))

len_d2 [int] Number of output vector produced by d2

e_ops [list of *qutip.Qobj*] single operator or list of operators for which to evaluate expectation values. Must be a superoperator if the state vector is a density matrix.

kwargs [*dictionary*] Optional keyword arguments. See *qutip.stochastic.StochasticSolverOptions*.

Returns

output: ***qutip.solver.Result*** An instance of the class *qutip.solver.Result*.

photocurrent_mesolve(*H*, *rho0*, *times*, *c_ops=[]*, *sc_ops=[]*, *e_ops=[]*, *_safe_mode=True*, *args={}*, ***kwargs*)
Solve stochastic master equation using the photocurrent method.

Parameters

H [*qutip.Qobj*, or time dependent system.] System Hamiltonian. Can depend on time, see StochasticSolverOptions help for format.

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*, or time dependent Qobjs.] Deterministic collapse operator which will contribute with a standard Lindblad type of dissipation. Can depend on time, see StochasticSolverOptions help for format.

sc_ops [list of *qutip.Qobj*, or time dependent Qobjs.] 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. Can depend on time, see StochasticSolverOptions help for format.

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.Result*** An instance of the class *qutip.solver.Result*.

photocurrent_sesolve(*H*, *psi0*, *times*, *sc_ops=[]*, *e_ops=[]*, *_safe_mode=True*, *args={}*, ***kwargs*)

Solve stochastic schrodinger equation using the photocurrent method.

Parameters

H [*qutip.Qobj*, or time dependent system.] System Hamiltonian. Can depend on time, see StochasticSolverOptions help for format.

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*, or time dependent Qobjs.] 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. Can depend on time, see StochasticSolverOptions help for format.

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.Result*** An instance of the class *qutip.solver.Result*.

smepdpsolve(*H*, *rho0*, *times*, *c_ops*, *e_ops*, ***kwargs*)

A stochastic (piecewse 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.Result* An instance of the class *qutip.solver.Result*.

smesolve(*H*, *rho0*, *times*, *c_ops=[]*, *sc_ops=[]*, *e_ops=[]*, *_safe_mode=True*, *args={}*, ***kwargs*)
Solve stochastic master equation. Dispatch to specific solvers depending on the value of the *solver* keyword argument.

Parameters

H [*qutip.Qobj*, or time dependent system.] System Hamiltonian. Can depend on time, see StochasticSolverOptions help for format.

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*, or time dependent Qobjs.] Deterministic collapse operator which will contribute with a standard Lindblad type of dissipation. Can depend on time, see StochasticSolverOptions help for format.

sc_ops [list of *qutip.Qobj*, or time dependent Qobjs.] 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. Can depend on time, see StochasticSolverOptions help for format.

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.Result* An instance of the class *qutip.solver.Result*.

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.Result* An instance of the class *qutip.solver.Result*.

ssesolve(*H*, *psi0*, *times*, *sc_ops=[]*, *e_ops=[]*, *_safe_mode=True*, *args={}*, ***kwargs*)
Solve stochastic schrodinger equation. Dispatch to specific solvers depending on the value of the *solver* keyword argument.

Parameters

H [*qutip.Qobj*, or time dependent system.] System Hamiltonian. Can depend on time, see StochasticSolverOptions help for format.

psi0 [*qutip.Qobj*] State vector (ket).

times [*list / array*] List of times for t. Must be uniformly spaced.

sc_ops [list of *qutip.Qobj*, or time dependent Qobjs.] 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. Can depend on time, see StochasticSolverOptions help for format.

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.Result* An instance of the class *qutip.solver.Result*.

stochastic_solvers()

This function is purely a reference point for documenting the available stochastic solver methods, and takes no actions.

Notes

Available solvers for *ssesolve* and *smesolve*

euler-maruyama A simple generalization of the Euler method for ordinary differential equations to stochastic differential equations. Only solver which could take non-commuting `sc_ops`. *not tested*

- Order 0.5
- Code: `'euler-maruyama'`, `'euler'` or `0.5`

milstein An order 1.0 strong Taylor scheme. Better approximate numerical solution to stochastic differential equations. See eq. (2.9) of chapter 12.2 of [1].

- Order strong 1.0
- Code: `'milstein'` or `1.0`

milstein-imp An order 1.0 implicit strong Taylor scheme. Implicit Milstein scheme for the numerical simulation of stiff stochastic differential equations.

- Order strong 1.0
- Code: `'milstein-imp'`

predictor-corrector Generalization of the trapezoidal method to stochastic differential equations. More stable than explicit methods. See eq. (5.4) of chapter 15.5 of [1].

- Order strong 0.5, weak 1.0
- Codes to only correct the stochastic part ($\alpha = 0$, $\eta = 1/2$): `'pred-corr'`, `'predictor-corrector'` or `'pc-euler'`
- Codes to correct both the stochastic and deterministic parts ($\alpha = 1/2$, $\eta = 1/2$): `'pc-euler-imp'`, `'pc-euler-2'` or `'pred-corr-2'`

platen Explicit scheme, creates the Milstein using finite differences instead of analytic derivatives. Also contains some higher order terms, thus converges better than Milstein while staying strong order 1.0. Does not require derivatives, therefore usable by *general_stochastic*. See eq. (7.47) of chapter 7 of [2].

- Order strong 1.0, weak 2.0
- Code: `'platen'`, `'platen1'` or `'explicit1'`

rouchon Scheme keeping the positivity of the density matrix (*smesolve* only). See eq. (4) with $\eta = 1$ of [3].

- Order strong 1.0?
- Code: `'rouchon'` or `'Rouchon'`

taylor1.5 Order 1.5 strong Taylor scheme. Solver with more terms of the Ito-Taylor expansion. Default solver for *smesolve* and *ssesolve*. See eq. (4.6) of chapter 10.4 of [1].

- Order strong 1.5
- Code: `'taylor1.5'`, `'taylor15'`, `1.5`, or `None`

taylor1.5-imp Order 1.5 implicit strong Taylor scheme. Implicit Taylor 1.5 ($\alpha = 1/2$, β doesn't matter). See eq. (2.18) of chapter 12.2 of [1].

- Order strong 1.5
- Code: `'taylor1.5-imp'` or `'taylor15-imp'`

explicit1.5 Explicit order 1.5 strong schemes. Reproduce the order 1.5 strong Taylor scheme using finite difference instead of derivatives. Slower than `taylor15` but usable by *general_stochastic*. See eq. (2.13) of chapter 11.2 of [1].

- Order strong 1.5
- Code: `'explicit1.5'`, `'explicit15'` or `'platen15'`

taylor2.0 Order 2 strong Taylor scheme. Solver with more terms of the Stratonovich expansion. See eq. (5.2) of chapter 10.5 of [1].

- Order strong 2.0
- Code: `'taylor2.0'`, `'taylor20'` or `2.0`

All solvers, except taylor2.0, are usable in both smesolve and ssesolve and for both heterodyne and homodyne. taylor2.0 only works for 1 stochastic operator independent of time with the homodyne method. *general_stochastic* only accepts the derivative-free solvers: `'euler'`, `'platen'` and `'explicit1.5'`.

Available solvers for *photocurrent_sesolve* and *photocurrent_mesolve* Photocurrent use ordinary differential equations between stochastic "jump/collapse".

euler Euler method for ordinary differential equations between jumps. Only one jump per time interval. Default solver. See eqs. (4.19) and (4.4) of chapter 4 of [4].

- Order 1.0
- Code: `'euler'`

predictor–corrector predictor–corrector method (PECE) for ordinary differential equations. Uses the Poisson distribution to obtain the number of jumps at each timestep.

- Order 2.0
- Code: `'pred-corr'`

References

[1], [2], [3], [4]

Correlation Functions

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.

args [dict] Dictionary of arguments to be passed to solver.

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.

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_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 *taulist*.

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 $t \to \infty$; 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 {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.

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.

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_correlation_fft(*tlist*, *y*, *inverse=False*)

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.

inverse: boolean boolean parameter for using a positive exponent in the Fourier Transform instead. Default is False.

Returns

w, S [tuple] Returns an array of angular frequencies 'w' and the corresponding two-sided power spectrum 'S(w)'.

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_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*.

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.

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.

steadystate(*A*, *c_op_list=[]*, *method='direct'*, *solver=None*, ***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.

solver [{'scipy', 'mkl'}, optional] Selects the sparse solver to use. Default is to auto-select based on the availability of the MKL library.

method [str, default 'direct'] The allowed methods are

- '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, default False] Return a dictionary of solver-specific infomation about the solution and how it was obtained.

sparse [bool, 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, default False] Use reverse Cuthill-Mckee reordering to minimize fill-in in the LU factorization of the Liouvillian.

use_wbm [bool, 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.

max_iter_refine [int, default 10] MKL ONLY. Max. number of iterative refinements to perform.

scaling_vectors [bool] MKL ONLY. Scale matrix to unit norm columns and rows.

weighted_matching [bool] MKL ONLY. Use weighted matching to better condition diagonal.

x0 [ndarray, optional] ITERATIVE ONLY. Initial guess for solution vector.

maxiter [int, default 1000] ITERATIVE ONLY. Maximum number of iterations to perform.

tol [float, default 1e-12] ITERATIVE ONLY. Tolerance used for terminating solver.

mtol [float, optional] ITERATIVE 'power' methods ONLY. Tolerance for lu solve method. If None given then `max(0.1*tol, 1e-15)` is used.

matol [float, default 1e-15] ITERATIVE ONLY. Absolute tolerance for lu solve method.

permc_spec [str, optional] ITERATIVE ONLY. Column ordering used internally by superLU for the 'direct' LU decomposition method. Options include 'COLAMD' (default) and 'NATURAL'. If using RCM then this is set to 'NATURAL' automatically unless explicitly specified.

use_precond [bool, 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, 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, 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] 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, 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).

Propagators

propagator(*H*, *t*, *c_op_list=[]*, *args={}*, *options=None*, *unitary_mode='batch'*, *parallel=False*, *progress_bar=None*, *_safe_mode=True*, ***kwargs*)

Calculate the propagator U(t) for the density matrix or wave function such that $\psi(t) = U(t)\psi(0)$ or $\rho_{\rm vec}(t) = U(t)\rho_{\rm vec}(0)$ where $\rho_{\rm vec}$ is the vector representation of the density matrix.

Parameters

H [qobj or list] Hamiltonian as a Qobj instance of a nested list of Qobjs and coefficients in the list-string or list-function format for time-dependent Hamiltonians (see description in *qutip.mesolve*).

t [float or array-like] Time or list of times for which to evaluate the propagator.

c_op_list [list] List of qobj collapse operators.

args [list/array/dictionary] Parameters to callback functions for time-dependent Hamiltonians and collapse operators.

options [*qutip.solver.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_clear()
Resets the string-format time-dependent Hamiltonian parameters.

Returns

Nothing, just clears data from internal config module.

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.

Scattering in Quantum Optical Systems

Photon scattering in quantum optical systems

This module includes a collection of functions for numerically computing photon scattering in driven arbitrary systems coupled to some configuration of output waveguides. The implementation of these functions closely follows the mathematical treatment given in K.A. Fischer, et. al., Scattering of Coherent Pulses from Quantum Optical Systems (2017, arXiv:1710.02875).

scattering_probability(*H*, *psi0*, *n_emissions*, *c_ops*, *tlist*, *system_zero_state=None*, *construct_effective_hamiltonian=True*)
Compute the integrated probability of scattering n photons in an arbitrary system. This function accepts a nonlinearly spaced array of times.

Parameters

H [:class: qutip.Qobj or list] System-waveguide(s) Hamiltonian or effective Hamiltonian in Qobj or list-callback format. If construct_effective_hamiltonian is not specified, an effective Hamiltonian is constructed from H and *c_ops*.

psi0 [:class: qutip.Qobj] Initial state density matrix $\rho(t_0)$ or state vector $\psi(t_0)$.

n_emissions [int] Number of photons emitted by the system (into any combination of waveguides).

c_ops [list] List of collapse operators for each waveguide; these are assumed to include spontaneous decay rates, e.g. $\sigma = \sqrt{\gamma} \cdot a$.

tlist [array_like] List of times for τ_i. tlist should contain 0 and exceed the pulse duration / temporal region of interest; tlist need not be linearly spaced.

system_zero_state [:class: qutip.Qobj] State representing zero excitations in the system. Defaults to *basis(systemDims, 0)*.

construct_effective_hamiltonian [bool] Whether an effective Hamiltonian should be constructed from H and c_ops: $H_{eff} = H - \frac{i}{2}\sum_n \sigma_n^\dagger \sigma_n$ Default: True.

Returns

scattering_prob [float] The probability of scattering n photons from the system over the time range specified.

temporal_basis_vector(*waveguide_emission_indices*, *n_time_bins*)

Generate a temporal basis vector for emissions at specified time bins into specified waveguides.

Parameters

waveguide_emission_indices [list or tuple] List of indices where photon emission occurs for each waveguide, e.g. [[t1_wg1], [t1_wg2, t2_wg2], [], [t1_wg4, t2_wg4, t3_wg4]].

n_time_bins [int] Number of time bins; the range over which each index can vary.

Returns

temporal_basis_vector [:class: qutip.Qobj] A basis vector representing photon scattering at the specified indices. If there are W waveguides, T times, and N photon emissions, then the basis vector has dimensionality (W*T)^N.

temporal_scattered_state(*H*, *psi0*, *n_emissions*, *c_ops*, *tlist*, *system_zero_state=None*, *construct_effective_hamiltonian=True*)

Compute the scattered n-photon state projected onto the temporal basis.

Parameters

H [:class: qutip.Qobj or list] System-waveguide(s) Hamiltonian or effective Hamiltonian in Qobj or list-callback format. If construct_effective_hamiltonian is not specified, an effective Hamiltonian is constructed from *H* and *c_ops*.

psi0 [:class: qutip.Qobj] Initial state density matrix $\rho(t_0)$ or state vector $\psi(t_0)$.

n_emissions [int] Number of photon emissions to calculate.

c_ops [list] List of collapse operators for each waveguide; these are assumed to include spontaneous decay rates, e.g. $\sigma = \sqrt{\gamma} \cdot a$

tlist [array_like] List of times for τ_i. tlist should contain 0 and exceed the pulse duration / temporal region of interest.

system_zero_state [:class: qutip.Qobj] State representing zero excitations in the system. Defaults to $\psi(t_0)$

construct_effective_hamiltonian [bool] Whether an effective Hamiltonian should be constructed from H and c_ops: $H_{eff} = H - \frac{i}{2}\sum_n \sigma_n^\dagger \sigma_n$ Default: True.

Returns

phi_n [:class: qutip.Qobj] The scattered bath state projected onto the temporal basis given by tlist. If there are W waveguides, T times, and N photon emissions, then the state is a tensor product state with dimensionality T^(W*N).

Permutational Invariance

Permutational Invariant Quantum Solver (PIQS)

This module calculates the Liouvillian for the dynamics of ensembles of identical two-level systems (TLS) in the presence of local and collective processes by exploiting permutational symmetry and using the Dicke basis. It also allows to characterize nonlinear functions of the density matrix.

am(*j*, *m*)
Calculate the operator `am` used later.

The action of `ap` is given by: $J_-|j,m\rangle = A_-(jm)|j,m-1\rangle$

Parameters

j: float The value for j.

m: float The value for m.

Returns

a_minus: float The value of a_-.

ap(*j*, *m*)
Calculate the coefficient `ap` by applying $J_+|j,m\rangle$.

The action of ap is given by: $J_+|j,m\rangle = A_+(j,m)|j,m+1\rangle$

Parameters

j, m: float The value for j and m in the dicke basis $|j,m\rangle$.

Returns

a_plus: float The value of a_+.

block_matrix(*N*, *elements='ones'*)
Construct the block-diagonal matrix for the Dicke basis.

Parameters

N [int] Number of two-level systems.

elements [str {'ones' (default),'degeneracy'}]

Returns

block_matr [ndarray] A 2D block-diagonal matrix with dimension (nds,nds), where nds is the number of Dicke states for N two-level systems. Filled with ones or the value of degeneracy at each matrix element.

collapse_uncoupled(*N*, *emission=0.0*, *dephasing=0.0*, *pumping=0.0*, *collective_emission=0.0*, *collective_dephasing=0.0*, *collective_pumping=0.0*)
Create the collapse operators (c_ops) of the Lindbladian in the uncoupled basis

These operators are in the uncoupled basis of the two-level system (TLS) SU(2) Pauli matrices.

Parameters

N: int The number of two-level systems.

emission: float Incoherent emission coefficient (also nonradiative emission). default: 0.0

dephasing: float Local dephasing coefficient. default: 0.0

pumping: float Incoherent pumping coefficient. default: 0.0

collective_emission: float Collective (superradiant) emmission coefficient. default: 0.0

collective_pumping: float Collective pumping coefficient. default: 0.0

collective_dephasing: float Collective dephasing coefficient. default: 0.0

Returns

c_ops: list The list of collapse operators as *qutip.Qobj* for the system.

Notes

The collapse operator list can be given to *qutip.mesolve*. Notice that the operators are placed in a Hilbert space of dimension 2^N. Thus the method is suitable only for small N (of the order of 10).

css (*N*, *x=0.7071067811865475*, *y=0.7071067811865475*, *basis='dicke'*, *coordinates='cartesian'*)
Generate the density matrix of the Coherent Spin State (CSS).

It can be defined as, $|CSS\rangle = \prod_i^N (a|1\rangle_i + b|0\rangle_i)$ with $a = sin(\frac{\theta}{2})$, $b = e^{i\phi}\cos\left(\frac{\theta}{2}\right)$. The default basis is that of Dicke space $|j, m\rangle\langle j, m'|$. The default state is the symmetric CSS, $|CSS\rangle = |+\rangle$.

Parameters

N: int The number of two-level systems.

x, y: float The coefficients of the CSS state.

basis: str The basis to use. Either "dicke" or "uncoupled".

coordinates: str Either "cartesian" or "polar". If polar then the coefficients are constructed as sin(x/2), cos(x/2)e^(iy).

Returns

rho: :class: qutip.Qobj The CSS state density matrix.

dicke (*N*, *j*, *m*)
Generate a Dicke state as a pure density matrix in the Dicke basis.

For instance, the superradiant state given by $|j, m\rangle = |1, 0\rangle$ for N = 2, and the state is represented as a density matrix of size (nds, nds) or (4, 4), with the (1, 1) element set to 1.

Parameters

N: int The number of two-level systems.

j: float The eigenvalue j of the Dicke state (j, m).

m: float The eigenvalue m of the Dicke state (j, m).

Returns

rho: :class: qutip.Qobj The density matrix.

dicke_basis (*N*, *jmm1=None*)
Initialize the density matrix of a Dicke state for several (j, m, m1).

This function can be used to build arbitrary states in the Dicke basis $|j, m\rangle\langle j, m'|$. We create coefficients for each (j, m, m1) value in the dictionary jmm1. The mapping for the (i, k) index of the density matrix to the $|j, m\rangle$ values is given by the cythonized function *jmm1_dictionary*. A density matrix is created from the given dictionary of coefficients for each (j, m, m1).

Parameters

N: int The number of two-level systems.

jmm1: dict A dictionary of {(j, m, m1): p} that gives a density p for the (j, m, m1) matrix element.

Returns

rho: :class: qutip.Qobj The density matrix in the Dicke basis.

dicke_blocks (*rho*)
Create the list of blocks for block-diagonal density matrix in the Dicke basis.

Parameters

rho [*qutip.Qobj*] A 2D block-diagonal matrix of ones with dimension (nds,nds), where nds is the number of Dicke states for N two-level systems.

Returns

square_blocks: list of np.array Give back the blocks list.

dicke_blocks_full(*rho*)

Give the full (2^N-dimensional) list of blocks for a Dicke-basis matrix.

Parameters

rho [*qutip.Qobj*] A 2D block-diagonal matrix of ones with dimension (nds,nds), where nds is the number of Dicke states for N two-level systems.

Returns

full_blocks [list] The list of blocks expanded in the 2^N space for N qubits.

dicke_function_trace(*f*, *rho*)

Calculate the trace of a function on a Dicke density matrix. :param f: A Taylor-expandable function of *rho*. :type f: function :param rho: A density matrix in the Dicke basis. :type rho: *qutip.Qobj*

Returns

res [float] Trace of a nonlinear function on *rho*.

energy_degeneracy(*N*, *m*)

Calculate the number of Dicke states with same energy.

The use of the *Decimals* class allows to explore N > 1000, unlike the built-in function *scipy.special.binom*

Parameters

N: int The number of two-level systems.

m: float Total spin z-axis projection eigenvalue. This is proportional to the total energy.

Returns

degeneracy: int The energy degeneracy

entropy_vn_dicke(*rho*)

Von Neumann Entropy of a Dicke-basis density matrix.

Parameters

rho [*qutip.Qobj*] A 2D block-diagonal matrix of ones with dimension (nds,nds), where nds is the number of Dicke states for N two-level systems.

Returns

entropy_dm: float Entropy. Use degeneracy to multiply each block.

excited(*N*, *basis='dicke'*)

Generate the density matrix for the excited state.

This state is given by (N/2, N/2) in the default Dicke basis. If the argument *basis* is "uncoupled" then it generates the state in a 2**N dim Hilbert space.

Parameters

N: int The number of two-level systems.

basis: str The basis to use. Either "dicke" or "uncoupled".

Returns

state: :class: qutip.Qobj The excited state density matrix in the requested basis.

ghz(*N*, *basis='dicke'*)

Generate the density matrix of the GHZ state.

If the argument *basis* is "uncoupled" then it generates the state in a 2^N-dimensional Hilbert space.

Parameters

N: int The number of two-level systems.

basis: str The basis to use. Either "dicke" or "uncoupled".

Returns

state: :class: qutip.Qobj The GHZ state density matrix in the requested basis.

ground(*N*, *basis='dicke'*)

Generate the density matrix of the ground state.

This state is given by (N/2, -N/2) in the Dicke basis. If the argument *basis* is "uncoupled" then it generates the state in a 2^N-dimensional Hilbert space.

Parameters

N: int The number of two-level systems.

basis: str The basis to use. Either "dicke" or "uncoupled"

Returns

state: :class: qutip.Qobj The ground state density matrix in the requested basis.

identity_uncoupled(*N*)

Generate the identity in a 2^N-dimensional Hilbert space.

The identity matrix is formed from the tensor product of N TLSs.

Parameters

N: int The number of two-level systems.

Returns

identity: :class: qutip.Qobj The identity matrix.

isdiagonal(*mat*)

Check if the input matrix is diagonal.

Parameters

mat: ndarray/Qobj A 2D numpy array

Returns

diag: bool True/False depending on whether the input matrix is diagonal.

jspin(*N*, *op=None*, *basis='dicke'*)

Calculate the list of collective operators of the total algebra.

The Dicke basis $|j,m\rangle\langle j,m'|$ is used by default. Otherwise with "uncoupled" the operators are in a 2^N space.

Parameters

N: int Number of two-level systems.

op: str The operator to return 'x','y','z','+','-'. If no operator given, then output is the list of operators for ['x','y','z'].

basis: str The basis of the operators - "dicke" or "uncoupled" default: "dicke".

Returns

j_alg: list or :class: qutip.Qobj A list of *qutip.Qobj* representing all the operators in the "dicke" or "uncoupled" basis or a single operator requested.

m_degeneracy(*N*, *m*)

Calculate the number of Dicke states $|j,m\rangle$ with same energy.

Parameters

N: int The number of two-level systems.

m: float Total spin z-axis projection eigenvalue (proportional to the total energy).

Returns

degeneracy: int The m-degeneracy.

num_dicke_ladders(*N*)

Calculate the total number of ladders in the Dicke space.

For a collection of N two-level systems it counts how many different "j" exist or the number of blocks in the block-diagonal matrix.

Parameters

N: int The number of two-level systems.

Returns

Nj: int The number of Dicke ladders.

num_dicke_states(*N*)

Calculate the number of Dicke states.

Parameters

N: int The number of two-level systems.

Returns

nds: int The number of Dicke states.

num_tls(*nds*)

Calculate the number of two-level systems.

Parameters

nds: int The number of Dicke states.

Returns

N: int The number of two-level systems.

purity_dicke(*rho*)

Calculate purity of a density matrix in the Dicke basis. It accounts for the degenerate blocks in the density matrix.

Parameters

rho [*qutip.Qobj*] Density matrix in the Dicke basis of qutip.piqs.jspin(N), for N spins.

Returns

purity [float] The purity of the quantum state. It's 1 for pure states, 0<=purity<1 for mixed states.

spin_algebra(*N*, *op=None*)

Create the list [sx, sy, sz] with the spin operators.

The operators are constructed for a collection of N two-level systems (TLSs). Each element of the list, i.e., sx, is a vector of *qutip.Qobj* objects (spin matrices), as it cointains the list of the SU(2) Pauli matrices for the N TLSs. Each TLS operator sx[i], with i = 0, ..., (N-1), is placed in a 2^N-dimensional Hilbert space.

Parameters

N: int The number of two-level systems.

Returns

spin_operators: list or :class: qutip.Qobj A list of *qutip.Qobj* operators - [sx, sy, sz] or the requested operator.

Notes

sx[i] is $\frac{\sigma_x}{2}$ in the composite Hilbert space.

state_degeneracy(*N*, *j*)
Calculate the degeneracy of the Dicke state.

Each state $|j, m\rangle$ includes D(N,j) irreducible representations $|j, m, \alpha\rangle$.

Uses Decimals to calculate higher numerator and denominators numbers.

Parameters

N: int The number of two-level systems.

j: float Total spin eigenvalue (cooperativity).

Returns

degeneracy: int The state degeneracy.

superradiant(*N*, *basis='dicke'*)
Generate the density matrix of the superradiant state.

This state is given by (N/2, 0) or (N/2, 0.5) in the Dicke basis. If the argument *basis* is "uncoupled" then it generates the state in a 2**N dim Hilbert space.

Parameters

N: int The number of two-level systems.

basis: str The basis to use. Either "dicke" or "uncoupled".

Returns

state: :class: qutip.Qobj The superradiant state density matrix in the requested basis.

tau_column(*tau*, *k*, *j*)
Determine the column index for the non-zero elements of the matrix for a particular row *k* and the value of *j* from the Dicke space.

Parameters

tau: str The tau function to check for this *k* and *j*.

k: int The row of the matrix M for which the non zero elements have to be calculated.

j: float The value of *j* for this row.

5.2.5 Lattice

Lattice Properties

cell_structures(*val_s=None*, *val_t=None*, *val_u=None*)
Returns two matrices H_cell and cell_T to help the user form the inputs for defining an instance of Lattice1d and Lattice2d classes. The two matrices are the intra and inter cell Hamiltonians with the tensor structure of the specified site numbers and/or degrees of freedom defined by the user.

Parameters

val_s [list of str/str] The first list of str's specifying the sites/degrees of freedom in the unitcell

val_t [list of str/str] The second list of str's specifying the sites/degrees of freedom in the unitcell

val_u [list of str/str] The third list of str's specifying the sites/degrees of freedom in the unitcell

Returns

H_cell_s [list of list of str] tensor structure of the cell Hamiltonian elements

T_inter_cell_s [list of list of str] tensor structure of the inter cell Hamiltonian elements

H_cell [Qobj] A Qobj initiated with all 0s with proper shape for an input as Hamiltonian_of_cell in Lattice1d.__init__()

T_inter_cell [Qobj] A Qobj initiated with all 0s with proper shape for an input as inter_hop in Lattice1d.__init__()

Topology

berry_curvature(*eigfs*)

Computes the discretized Berry curvature on the two dimensional grid of parameters. The function works well for cases with no band mixing.

Parameters

eigfs [numpy ndarray] 4 dimensional numpy ndarray where the first two indices are for the two discrete values of the two parameters and the third is the index of the occupied bands. The fourth dimension holds the eigenfunctions.

Returns

b_curv [numpy ndarray] A two dimensional array of the discretized Berry curvature defined for the values of the two parameters defined in the eigfs.

plot_berry_curvature(*eigfs*)

Plots the discretized Berry curvature on the two dimensional grid of parameters. The function works well for cases with no band mixing.

5.2.6 Visualization

Pseudoprobability Functions

qfunc(*state: qutip.qobj.Qobj*, *xvec*, *yvec*, *g: float = 1.4142135623730951*, *precompute_memory: float = 1024*)

Husimi-Q function of a given state vector or density matrix at phase-space points `0.5 * g * (xvec + i*yvec)`.

Parameters

state [*Qobj*] A state vector or density matrix. This cannot have tensor-product structure.

xvec, yvec [array_like] x- and y-coordinates at which to calculate the Husimi-Q function.

g [float, default sqrt(2)] Scaling factor for `a = 0.5 * g * (x + iy)`. The value of g is related to the value of $\hbar$ in the commutation relation $[x,\ y] = i\hbar$ via $\hbar = 2/g^2$, so the default corresponds to $\hbar = 1$.

precompute_memory [real, default 1024] Size in MB that may be used during calculations as working space when dealing with density-matrix inputs. This is ignored for state-vector inputs. The bound is not quite exact due to other, order-of-magnitude smaller, intermediaries being necessary, but is a good approximation. If you want to use the same iterative algorithm for density matrices that is used for single kets, set `precompute_memory=None`.

Returns

ndarray Values representing the Husimi-Q function calculated over the specified range `[xvec, yvec]`.

See also:

QFunc a class-based version, more efficient if you want to calculate the Husimi-Q function for several states over the same coordinates.

spin_q_function(*rho*, *theta*, *phi*)

The Husimi Q function for spins is defined as `Q(theta, phi) = SCS.dag() * rho * SCS` for the spin coherent state `SCS = spin_coherent( j, theta, phi)` where j is the spin length. The implementation here is more efficient as it doesn't generate all of the SCS at theta and phi (see references).

The spin Q function is normal when integrated over the surface of the sphere

$$\frac{4\pi}{2j+1}\int_\phi\int_\theta Q(\theta,\phi)\sin(\theta)d\theta d\phi = 1$$

Parameters

state [qobj] A state vector or density matrix for a spin-j quantum system.

theta [array_like] Polar (colatitude) angle at which to calculate the Husimi-Q function.

phi [array_like] Azimuthal angle at which to calculate the Husimi-Q function.

Returns

Q, THETA, PHI [2d-array] Values representing the spin Husimi Q function at the values specified by THETA and PHI.

References

[1] Lee Loh, Y., & Kim, M. (2015). American J. of Phys., 83(1), 30–35. https://doi.org/10.1119/1.4898595

spin_wigner(*rho*, *theta*, *phi*)

Wigner function for a spin-j system.

The spin W function is normal when integrated over the surface of the sphere

$$\sqrt{\frac{4\pi}{2j+1}}\int_\phi\int_\theta W(\theta,\phi)\sin(\theta)d\theta d\phi = 1$$

Parameters

state [qobj] A state vector or density matrix for a spin-j quantum system.

theta [array_like] Polar (colatitude) angle at which to calculate the W function.

phi [array_like] Azimuthal angle at which to calculate the W function.

Returns

W, THETA, PHI [2d-array] Values representing the spin Wigner function at the values specified by THETA and PHI.

References

[1] Agarwal, G. S. (1981). Phys. Rev. A, 24(6), 2889–2896. https://doi.org/10.1103/PhysRevA.24.2889

[2] Dowling, J. P., Agarwal, G. S., & Schleich, W. P. (1994). Phys. Rev. A, 49(5), 4101–4109. https://doi.org/10.1103/PhysRevA.49.4101

[3] Conversion between Wigner 3-j symbol and Clebsch-Gordan coefficients taken from Wikipedia (https://en.wikipedia.org/wiki/3-j_symbol)

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)*. The value of *g* is related to the value of *hbar* in the commutation relation *[x, y] = i * hbar* via *hbar=2/g^2* giving the default value *hbar=1*.

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*, *color_style='scaled'*)

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.

color_style [string] Determines how colors are assigned to each square:

- If set to `"scaled"` (default), each color is chosen by passing the absolute value of the corresponding matrix element into *cmap* with the sign of the real part.
- If set to `"threshold"`, each square is plotted as the maximum of *cmap* for the positive real part and as the minimum for the negative part of the matrix element; note that this generalizes *"threshold"* to complex numbers.
- If set to `"phase"`, each color is chosen according to the angle of the corresponding matrix element.

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.

Examples

```
>>> import qutip
>>>
>>> dm = qutip.rand_dm(4)
>>> fig, ax = qutip.hinton(dm)
>>> fig.show()
>>>
>>> qutip.settings.colorblind_safe = True
>>> fig, ax = qutip.hinton(dm, color_style="threshold")
>>> fig.show()
>>> qutip.settings.colorblind_safe = False
>>>
>>> fig, ax = qutip.hinton(dm, color_style="phase")
>>> fig.show()
```

matrix_histogram(*M*, *xlabels=None*, *ylabels=None*, *title=None*, *limits=None*, *colorbar=True*, *fig=None*, *ax=None*, *options=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.

colorbar [bool (default: True)] show colorbar

options [dict] A dictionary containing extra options for the plot. The names (keys) and values of the options are described below:

'zticks' [list of numbers] A list of z-axis tick locations.

'cmap' [string (default: 'jet')] The name of the color map to use.

'cmap_min' [float (default: 0.0)] The lower bound to truncate the color map at. A value in range 0 - 1. The default, 0, leaves the lower bound of the map unchanged.

'cmap_max' [float (default: 1.0)] The upper bound to truncate the color map at. A value in range 0 - 1. The default, 1, leaves the upper bound of the map unchanged.

'bars_spacing' [float (default: 0.1)] spacing between bars.

'bars_alpha' [float (default: 1.)] transparency of bars, should be in range 0 - 1

'bars_lw' [float (default: 0.5)] linewidth of bars' edges.

'bars_edgecolor' [color (default: 'k')] The colors of the bars' edges. Examples: 'k', (0.1, 0.2, 0.5) or '#0f0f0f80'.

'shade' [bool (default: True)] Whether to shade the dark sides of the bars (True) or not (False). The shading is relative to plot's source of light.

'azim' [float] The azimuthal viewing angle.

'elev' [float] The elevation viewing angle.

'proj_type' [string (default: 'ortho' if ax is not passed)] The type of projection ('ortho' or 'persp')

'stick' [bool (default: False)] Changes xlim and ylim in such a way that bars next to XZ and YZ planes will stick to those planes. This option has no effect if `ax` is passed as a parameter.

'cbar_pad' [float (default: 0.04)] The fraction of the original axes between the colorbar and the new image axes. (i.e. the padding between the 3D figure and the colorbar).

'cbar_to_z' [bool (default: False)] Whether to set the color of maximum and minimum z-values to the maximum and minimum colors in the colorbar (True) or not (False).

'figsize' [tuple of two numbers] The size of the figure.

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_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_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*] 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_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 2k particles and the rest.

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.

Notes

See also [1].

References

[1]

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_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.

plot_wigner(*rho*, *fig=None*, *ax=None*, *figsize=(6, 6)*, *cmap=None*, *alpha_max=7.5*, *colorbar=False*, *method='clenshaw'*, *projection='2d'*)

Plot the the Wigner function for a density matrix (or ket) that describes an oscillator mode.

Parameters

rho [*qutip.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 {'clenshaw', '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_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*] 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_sphere(*fig*, *ax*, *wigner*, *reflections*)

Plots a coloured Bloch sphere.

Parameters

fig [`matplotlib.figure.Figure`] An instance of `Figure`.

ax [`matplotlib.axes.Axes`] An axes instance in the given figure.

wigner [list of float] The wigner transformation at *steps* different theta and phi.

reflections [bool] If the reflections of the sphere should be plotted as well.

Notes

Special thanks to Russell P Rundle for writing this function.

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.

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. The first entry of the ket defines the coefficient c_{l,-l}, while the last entry of the ket defines the coefficient c_{l, l}.

Parameters

theta [int/float/list/array] Polar angles in [0, pi]

phi [int/float/list/array] Azimuthal angles in [0, 2*pi]

args [list/array] `list` of ket vectors.

Returns

array for angular wave function evaluated at all possible combinations of theta and phi

This module contains utility functions that enhance Matplotlib in one way or another.

complex_phase_cmap()
Create a cyclic colormap for representing the phase of complex variables

Returns

cmap : A matplotlib linear segmented colormap.

wigner_cmap(*W*, *levels=1024*, *shift=0*, *max_color='#09224F'*, *mid_color='#FFFFFF'*, *min_color='#530017'*, *neg_color='#FF97D4'*, *invert=False*)
A custom colormap that emphasizes negative values by creating a nonlinear colormap.

Parameters

W [array] Wigner function array, or any array.

levels [int] Number of color levels to create.

shift [float] Shifts the value at which Wigner elements are emphasized. This parameter should typically be negative and small (i.e -1e-5).

max_color [str] String for color corresponding to maximum value of data. Accepts any string format compatible with the Matplotlib.colors.ColorConverter.

mid_color [str] Color corresponding to zero values. Accepts any string format compatible with the Matplotlib.colors.ColorConverter.

min_color [str] Color corresponding to minimum data values. Accepts any string format compatible with the Matplotlib.colors.ColorConverter.

neg_color [str] Color that starts highlighting negative values. Accepts any string format compatible with the Matplotlib.colors.ColorConverter.

invert [bool] Invert the color scheme for negative values so that smaller negative values have darker color.

Returns

Returns a Matplotlib colormap instance for use in plotting.

Notes

The 'shift' parameter allows you to vary where the colormap begins to highlight negative colors. This is beneficial in cases where there are small negative Wigner elements due to numerical round-off and/or truncation.

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.

5.2.7 Quantum Information Processing

Gates

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]]
```

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]]
```

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.

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.

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]]
```

expand_operator (*oper*, *N*, *targets*, *dims=None*, *cyclic_permutation=False*)
Expand a qubits operator to one that acts on a N-qubit system.

Parameters

oper [*qutip.Qobj*] An operator acts on qubits, the type of the *qutip.Qobj* has to be an operator and the dimension matches the tensored qubit Hilbert space e.g. dims = `[[2, 2, 2], [2, 2, 2]]`

N [int] The number of qubits in the system.

targets [int or list of int] The indices of qubits that are acted on.

dims [list, optional] A list of integer for the dimension of each composite system. E.g `[2, 2, 2, 2, 2]` for 5 qubits system. If None, qubits system will be the default option.

cyclic_permutation [boolean, optional] Expand for all cyclic permutation of the targets. E.g. if `N=3` and *oper* is a 2-qubit operator, the result will be a list of three operators, each acting on qubits 0 and 1, 1 and 2, 2 and 0.

Returns

expanded_oper [*qutip.Qobj*] The expanded qubits operator acting on a system with N qubits.

Notes

This is equivalent to gate_expand_1toN, gate_expand_2toN, gate_expand_3toN in `qutip.qip.gate.py`, but works for any dimension.

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]]
```

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.

gate_sequence_product(*U_list*, *left_to_right=True*, *inds_list=None*, *expand=False*)
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, optional Check if multiplication is to be done from left to right.

inds_list: list of list of int, optional If expand=True, list of qubit indices corresponding to U_list to which each unitary is applied.

expand: Boolean, optional Check if the list of unitaries need to be expanded to full dimension.

Returns

U_overall [qobj] Unitary matrix corresponding to U_list.

overall_inds [list of int, optional] List of qubit indices on which U_overall applies.

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.

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]]
```

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]]
```

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.

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.

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]]
```

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]]
```

sqrtnot (*N=None*, *target=0*)

Single-qubit square root NOT gate.

Returns

result [qobj] Quantum object for operator describing the square root NOT gate.

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

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]]
```

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]]
```

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]]
```

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_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.

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.

Circuit

circuit_to_qasm_str(*qc*)
Return QASM output of circuit object as string

Parameters

qc: :class:`.QubitCircuit` circuit object to produce QASM output for.

Returns

output: str string corresponding to QASM output.

print_qasm(*qc*)
Print QASM output of circuit object.

Parameters

qc: :class:`.QubitCircuit` circuit object to produce QASM output for.

read_qasm(*qasm_input*, *mode='qiskit'*, *version='2.0'*, *strmode=False*)
Read OpenQASM intermediate representation (https://github.com/Qiskit/openqasm) and return a *QubitCircuit* and state inputs as specified in the QASM file.

Parameters

qasm_input [str] File location or String Input for QASM file to be imported. In case of string input, the parameter strmode must be True.

mode [str] QASM mode to be read in. When mode is "qiskit", the "qelib1.inc" include is automatically included, without checking externally. Otherwise, each include is processed.

version [str] QASM version of the QASM file. Only version 2.0 is currently supported.

strmode [bool] if specified as True, indicates that qasm_input is in string format rather than from file.

Returns

qc [*QubitCircuit*] Returns a *QubitCircuit* object specified in the QASM file.

save_qasm(*qc*, *file_loc*)
Save QASM output of circuit object to file.

Parameters

qc: :class:`.QubitCircuit` circuit object to produce QASM output for.

5.2.8 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.transfertensor.TTMSolverOptions*.

Returns

output: ***qutip.solver.Result*** An instance of the class *qutip.solver.Result*.

5.2.9 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).

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 input 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 dervived 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.

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 PropagatorComputer, 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 dervived 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 *qutip.control.dynamics.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, default None] Type / shape of pulse(s) used modulate the control amplitudes. Options include: RND, LIN, ZERO, SINE, SQUARE, TRIANGLE, SAW, GAUSSIAN.

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, default 'MODULATE'] Determines how the guess pulse is applied to the pulse generated by the basis expansion. Options are: MODULATE, ADD.

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 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

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 performance 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.minimize` 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, optional] Type / shape of pulse(s) used modulate the control amplitudes. Options include: RND, LIN, ZERO, SINE, SQUARE, TRIANGLE, SAW, GAUSSIAN.

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, 'MODULATE'] Determines how the guess pulse is applied to the pulse generated by the basis expansion. Options are: MODULATE, ADD.

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.

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 initial (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 input 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 compatibility 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 propagators and propagator 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 control amplitudes. Options (GRAPE) include: RND, LIN, ZERO, SINE, SQUARE, TRIANGLE, SAW. Default 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 initial (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 input 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 compatibility reasons. Supplying `DEF` will given algorithm-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 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.

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.

5.2.10 Utility 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).

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).

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).

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.

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.

n_thermal(*w*, *w_th*)
Return the number of photons in thermal equilibrium for an harmonic oscillator mode with frequency 'w', at the temperature described by 'w_th' where $\omega_{\rm th} = k_B T/\hbar$.

Parameters

w [*float* or *array*] Frequency of the oscillator.

w_th [*float*] The temperature in units of frequency (or the same units as *w*).

Returns

n_avg [*float* or *array*] Return the number of average photons in thermal equilibrium for a an oscillator with the given frequency and temperature.

File I/O Functions

file_data_read(*filename*, *sep=None*)

Retrieves an array of data from the requested file.

Parameters

filename [str or pathlib.Path] 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 or pathlib.Path] 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 or pathlib.Path] 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 or pathlib.Path] Name of output data file.

Parallelization

This function provides functions for parallel execution of loops and function mappings, using the builtin Python module multiprocessing.

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`.

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.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*.

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.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`.

Semidefinite Programming

This module implements internal-use functions for semidefinite programming.

IPython Notebook Tools

This module contains utility functions for using QuTiP with IPython notebooks.

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`.

parfor(*task*, *task_vec*, *args=None*, *client=None*, *view=None*, *show_scheduling=False*, *show_progressbar=False*)

Call the function `tast` for each value in `task_vec` using a cluster of IPython engines. The function `task` should have the signature `task(value, args)` or `task(value)` if `args=None`.

The `client` and `view` are the IPython.parallel client and load-balanced view that will be used in the parfor execution. If these are `None`, new instances will be created.

Parameters

task: a Python function The function that is to be called for each value in `task_vec`.

task_vec: array / list The list or array of values for which the `task` function is to be evaluated.

args: list / dictionary The optional additional argument to the `task` function. For example a dictionary with parameter values.

client: IPython.parallel.Client The IPython.parallel Client instance that will be used in the parfor execution.

view: a IPython.parallel.Client view The view that is to be used in scheduling the tasks on the IPython cluster. Preferably a load-balanced view, which is obtained from the IPython.parallel.Client instance client by calling, view = client.load_balanced_view().

show_scheduling: bool {False, True}, default False Display a graph showing how the tasks (the evaluation of `task` for for the value in `task_vec1`) was scheduled on the IPython engine cluster.

show_progressbar: bool {False, True}, default False Display a HTML-based progress bar duing the execution of the parfor loop.

Returns

result [list] The result list contains the value of `task(value, args)` for each value in `task_vec`, that is, it should be equivalent to `[task(v, args) for v in task_vec]`.

version_table(*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: bool = True*, *, *tol: float = 1e-14*, *safe_mode: bool = True*)

Simultaneous diagonalization of commuting Hermitian matrices.

Parameters

ops [list/array] `list` or `array` of qobjs representing commuting Hermitian operators.

evals [bool [True]] Whether to return the eigenvalues for each ops and eigenvectors or just the eigenvectors.

tol [float [1e-14]] Tolerance for detecting degenerate eigenstates.

safe_mode [bool [True]] Whether to check that all ops are Hermitian and commuting. If set to `False` and operators are not commuting, the eigenvectors returned will often be eigenvectors of only the first operator.

Returns

eigs [tuple] Tuple of arrays representing eigvecs and eigvals of quantum objects corresponding to simultaneous eigenvectors and eigenvalues for each operator.

Chapter 6

Change Log

6.1 Version 4.7.1 (December 11, 2022)

This is a bugfix release for QuTiP 4.7.X. In addition to the minor fixes listed below, the release adds builds for Python 3.11 and support for packaging 22.0.

6.1.1 Features

- Improve qutip import times by setting logger names explicitly. (#1980)

6.1.2 Bug Fixes

- Change floquet_master_equation_rates(...) to use an adaptive number of time steps scaled by the number of sidebands, kmax. (#1961)
- Change fidelity(A, B) to use the reduced fidelity formula for pure states which is more numerically efficient and accurate. (#1964)
- Change *brmesolve* to raise an exception when ode integration is not successful. (#1965)
- Backport fix for IPython helper Bloch._repr_svg_ from dev.major. Previously the print_figure function returned bytes, but since ipython/ipython#5452 (in 2014) it returns a Unicode string. This fix updates QuTiP's helper to match. (#1970)
- Fix correlation for case where only the collapse operators are time dependent. (#1979)
- Fix the hinton visualization method to plot the matrix instead of its transpose. (#2011)
- Fix the hinton visualization method to take into account all the matrix coefficients to set the squares scale, instead of only the diagonal coefficients. (#2012)
- Fix parsing of package versions in setup.py to support packaging 22.0. (#2037)
- Add back .qu suffix to objects saved with qsave and loaded with qload. The suffix was accidentally removed in QuTiP 4.7.0. (#2038)
- Add a default max_step to processors. (#2040)

6.1.3 Documentation

- Add towncrier for managing the changelog. (#1927)
- Update the version of numpy used to build documentation to 1.22.0. (#1940)
- Clarify returned objects from bloch_redfield_tensor(). (#1950)
- Update Floquet Markov solver docs. (#1958)
- Update the roadmap and ideas to show completed work as of August 2022. (#1967)

6.1.4 Miscellaneous

- Return TypeError instead of Exception for type error in sesolve argument. (#1924)
- Add towncrier draft build of changelog to CI tests. (#1946)
- Add Python 3.11 to builds. (#2041)
- Simplify version parsing by using packaging.version.Version. (#2043)
- Update builds to use cibuildwheel 2.11, and to build with manylinux2014 on Python 3.8 and 3.9, since numpy and SciPy no longer support manylinux2010 on those versions of Python. (#2047)

6.2 Version 4.7.0 (April 13, 2022)

This release sees the addition of two new solvers – `qutip.krylovsolve` based on the Krylov subspace approximation and `qutip.nonmarkov.heom` that reimplements the BoFiN HEOM solver.

Bloch sphere rendering gained support for drawing arcs and lines on the sphere, and for setting the transparency of rendered points and vectors, Hinton plots gained support for specifying a coloring style, and matrix histograms gained better default colors and more flexible styling options.

Other significant improvements include better scaling of the Floquet solver, support for passing `Path` objects when saving and loading files, support for passing callable functions as `e_ops` to `mesolve` and `sesolve`, and faster state number enumeration and Husimi Q functions.

Import bugfixes include some bugs affecting plotting with matplotlib 3.5 and fixing support for qutrits (and other non-qubit) quantum circuits.

The many other small improvements, bug fixes, documentation enhancements, and behind the scenese development changes are included in the list below.

QuTiP 4.7.X will be the last series of releases for QuTiP 4. Patch releases will continue for the 4.7.X series but the main development effort will move to QuTiP 5.

The many, many contributors who filed issues, submitted or reviewed pull requests, and improved the documentation for this release are listed next to their contributions below. Thank you to all of you.

6.2.1 Improvements

- **MAJOR** Added krylovsolve as a new solver based on krylov subspace approximation. (#1739 by Emiliano Fortes)
- **MAJOR** Imported BoFiN HEOM (https://github.com/tehruhn/bofin/) into QuTiP and replaced the HEOM solver with a compatibility wrapper around BoFiN bosonic solver. (#1601, #1726, and #1724 by Simon Cross, Tarun Raheja and Neill Lambert)
- **MAJOR** Added support for plotting lines and arcs on the Bloch sphere. (#1690 by Gaurav Saxena, Asier Galicia and Simon Cross)

- Added transparency parameter to the add_point, add_vector and add_states methods in the Bloch and Bloch3d classes. (#1837 by Xavier Spronken)
- Support `Path` objects in `qutip.fileio`. (#1813 by Adrià Labay)
- Improved the weighting in steadystate solver, so that the default weight matches the documented behaviour and the dense solver applies the weights in the same manner as the sparse solver. (#1275 and #1802 by NS2 Group at LPS and Simon Cross)
- Added a `color_style` option to the `hinton` plotting function. (#1595 by Cassandra Granade)
- Improved the scaling of `floquet_master_equation_rates` and `floquet_master_equation_tensor` and fixed transposition and basis change errors in `floquet_master_equation_tensor` and `floquet_markov_mesolve`. (#1248 by Camille Le Calonnec, Jake Lishman and Eric Giguère)
- Removed `linspace_with` and `view_methods` from `qutip.utilities`. For the former it is far better to use `numpy.linspace` and for the later Python's in-built `help` function or other tools. (#1680 by Eric Giguère)
- Added support for passing callable functions as `e_ops` to `mesolve` and `sesolve`. (#1655 by Marek Narożniak)
- Added the function `steadystate_floquet`, which returns the "effective" steadystate of a periodic driven system. (#1660 by Alberto Mercurio)
- Improved mcsolve memory efficiency by not storing final states when they are not needed. (#1669 by Eric Giguère)
- Improved the default colors and styling of matrix_histogram and provided additional styling options. (#1573 and #1628 by Mahdi Aslani)
- Sped up `state_number_enumerate`, `state_number_index`, `state_index_number`, and added some error checking. `enr_state_dictionaries` now returns a list for `idx2state`. (#1604 by Johannes Feist)
- Added new Husimi Q algorithms, improving the speed for density matrices, and giving a near order-of-magnitude improvement when calculating the Q function for many different states, using the new `qutip.QFunc` class, instead of the `qutip.qfunc` function. (#934 and #1583 by Daniel Weigand and Jake Lishman)
- Updated licence holders with regards to new governance model, and remove extraneous licensing information from source files. (#1579 by Jake Lishman)
- Removed the vendored copy of LaTeX's qcircuit package which is GPL licensed. We now rely on the package being installed by user. It is installed by default with TexLive. (#1580 by Jake Lishman)
- The signatures of rand_ket and rand_ket_haar were changed to allow N (the size of the random ket) to be determined automatically when dims are specified. (#1509 by Purva Thakre)

6.2.2 Bug Fixes

- Fix circuit index used when plotting circuits with non-reversed states. (#1847 by Christian Staufenbiel)
- Changed implementation of `qutip.orbital` to use `scipy.special.spy_harm` to remove bugs in angle interpretation. (#1844 by Christian Staufenbiel)
- Fixed `QobjEvo.tidyup` to use `settings.auto_tidyup_atol` when removing small elements in sparse matrices. (#1832 by Eric Giguère)
- Ensured that tidyup's default tolerance is read from settings at each call. (#1830 by Eric Giguère)
- Fixed `scipy.sparse` deprecation warnings raised by `qutip.fast_csr_matrix`. (#1827 by Simon Cross)
- Fixed rendering of vectors on the Bloch sphere when using matplotlib 3.5 and above. (#1818 by Simon Cross)

- Fixed the displaying of `Lattice1d` instances and their unit cells. Previously calling them raised exceptions in simple cases. (#1819, #1697 and #1702 by Simon Cross and Saumya Biswas)
- Fixed the displaying of the title for `hinton` and `matrix_histogram` plots when a title is given. Previously the supplied title was not displayed. (#1707 by Vladimir Vargas-Calderón)
- Removed an incorrect check on the initial state dimensions in the `QubitCircuit` constructor. This allows, for example, the construction of qutrit circuits. (#1807 by Boxi Li)
- Fixed the checking of `method` and `offset` parameters in `coherent` and `coherent_dm`. (#1469 and #1741 by Joseph Fox-Rabinovitz and Simon Cross)
- Removed the Hamiltonian saved in the `sesolve` solver results. (#1689 by Eric Giguère)
- Fixed a bug in rand_herm with `pos_def=True` and `density>0.5` where the diagonal was incorrectly filled. (#1562 by Eric Giguère)

6.2.3 Documentation Improvements

- Added contributors image to the documentation. (#1828 by Leonard Assis)
- Fixed the Theory of Quantum Information bibliography link. (#1840 by Anto Luketina)
- Fixed minor grammar errors in the dynamics guide. (#1822 by Victor Omole)
- Fixed many small documentation typos. (#1569 by Ashish Panigrahi)
- Added Pulser to the list of libraries that use QuTiP. (#1570 by Ashish Panigrahi)
- Corrected typo in the states and operators guide. (#1567 by Laurent Ajdnik)
- Converted http links to https. (#1555 by Jake Lishamn)

6.2.4 Developer Changes

- Add GitHub actions test run on windows-latest. (#1853 and #1855 by Simon Cross)
- Bumped the version of pillow used to build documentation from 9.0.0 to 9.0.1. (#1835 by dependabot)
- Migrated the `qutip.superop_reps` tests to pytest. (#1825 by Felipe Bivort Haiek)
- Migrated the `qutip.steadystates` tests to pytest. (#1679 by Eric Giguère)
- Changed the README.md CI badge to the GitHub Actions badge. (#1581 by Jake Lishman)
- Updated CodeClimate configuration to treat our Python source files as Python 3. (#1577 by Jake Lishman)
- Reduced cyclomatic complexity in `qutip._mkl`. (#1576 by Jake Lishman)
- Fixed PEP8 warnings in `qutip.control`, `qutip.mcsolve`, `qutip.random_objects`, and `qutip.stochastic`. (#1575 by Jake Lishman)
- Bumped the version of urllib3 used to build documentation from 1.26.4 to 1.26.5. (#1563 by dependabot)
- Moved tests to GitHub Actions. (#1551 by Jake Lishman)
- The GitHub contributing guidelines were re-added and updated to point to the more complete guidelines in the documentation. (#1549 by Jake Lishman)
- The release documentation was reworked after the initial 4.6.1 to match the actual release process. (#1544 by Jake Lishman)

6.3 Version 4.6.3 (February 9, 2022)

This minor release adds support for numpy 1.22 and Python 3.10 and removes some blockers for running QuTiP on the Apple M1.

The performance of the `enr_destroy`, `state_number_enumerate` and `hadamard_transform` functions was drastically improved (up to 70x or 200x faster in some common cases), and support for the drift Hamiltonian was added to the `qutip.qip` `Processor`.

The `qutip.hardware_info` module was removed as part of adding support for the Apple M1. We hope the removal of this little-used module does not adversely affect many users – it was largely unrelated to QuTiP's core functionality and its presence was a continual source of blockers to importing `qutip` on new or changed platforms.

A new check on the dimensions of `Qobj`'s were added to prevent segmentation faults when invalid shape and dimension combinations were passed to Cython code.

In addition, there were many small bugfixes, documentation improvements, and improvements to our building and testing processes.

6.3.1 Improvements

- The `enr_destroy` function was made ~200x faster in many simple cases. (#1593 by Johannes Feist)
- The `state_number_enumerate` function was made significantly faster. (#1594 by Johannes Feist)
- Added the missing drift Hamiltonian to the method run_analytically of `Processor`. (#1603 Boxi Li)
- The `hadamard_transform` was made much faster, e.g., ~70x faster for N=10. (#1688 by Asier Galicia)
- Added support for computing the power of a scalar-like Qobj. (#1692 by Asier Galicia)
- Removed the `hardware_info` module. This module wasn't used inside QuTiP and regularly broke when new operating systems were released, and in particular prevented importing QuTiP on the Apple M1. (#1754, #1758 by Eric Giguère)

6.3.2 Bug Fixes

- Fixed support for calculating the propagator of a density matrix with collapse operators. QuTiP 4.6.2 introduced extra sanity checks on the dimensions of inputs to mesolve (Fix mesolve segfault with bad initial state #1459), but the propagator function's calls to mesolve violated these checks by supplying initial states with the dimensions incorrectly set. `propagator` now calls mesolve with the correct dimensions set on the initial state. (#1588 by Simon Cross)
- Fixed support for calculating the propagator for a superoperator without collapse operators. This functionality was not tested by the test suite and appears to have broken sometime during 2019. Tests have now been added and the code breakages fixed. (#1588 by Simon Cross)
- Fixed the ignoring of the random number seed passed to `rand_dm` in the case where `pure` was set to true. (#1600 Pontus Wikståhl)
- Fixed qutip.control.optimize_pulse support for sparse eigenvector decomposition with the Qobj oper_dtype (the Qobj oper_dtype is the default for large systems). (#1621 by Simon Cross)
- Removed qutip.control.optimize_pulse support for scipy.sparse.csr_matrix and generic ndarray-like matrices. Support for these was non-functional. (#1621 by Simon Cross)
- Fixed errors in the calculation of the Husimi spin_q_function and spin_wigner functions and added tests for them. (#1632 by Mark Johnson)
- Fixed setting of OpenMP compilation flag on Linux. Previously when compiling the OpenMP functions were compiled without parallelization. (#1693 by Eric Giguère)

- Fixed tracking the state of the Bloch sphere figure and axes to prevent exceptions during rendering. (#1619 by Simon Cross)
- Fixed compatibility with numpy configuration in numpy's 1.22.0 release. (#1752 by Matthew Treinish)
- Added dims checks for e_ops passed to solvers to prevent hanging the calling process when e_ops of the wrong dimensions were passed. (#1778 by Eric Giguère)
- Added a check in Qobj constructor that the respective members of data.shape cannot be larger than what the corresponding dims could contain to prevent a segmentation fault caused by inconsistencies between dims and shapes. (#1783, #1785, #1784 by Lajos Palanki & Eric Giguère)

6.3.3 Documentation Improvements

- Added docs for the num_cbits parameter of the QubitCircuit class. (#1652 by Jon Crall)
- Fixed the parameters in the call to fsesolve in the Floquet guide. (#1675 by Simon Cross)
- Fixed the description of random number usage in the Monte Carlo solver guide. (#1677 by Ian Thorvaldson)
- Fixed the rendering of equation numbers in the documentation (they now appear on the right as expected, not above the equation). (#1678 by Simon Cross)
- Updated the installation requirements in the documentation to match what is specified in setup.py. (#1715 by Asier Galicia)
- Fixed a typo in the `chi_to_choi` documentation. Previously the documentation mixed up chi and choi. (#1731 by Pontus Wikståhl)
- Improved the documentation for the stochastic equation solvers. Added links to notebooks with examples, API doumentation and external references. (#1743 by Leonardo Assis)
- Fixed a typo in `qutip.settings` in the settings guide. (#1786 by Mahdi Aslani)
- Made numerous small improvements to the text of the QuTiP basics guide. (#1768 by Anna Naden)
- Made a small phrasing improvement to the README. (#1790 by Rita Abani)

6.3.4 Developer Changes

- Improved test coverage of states and operators functions. (#1578 by Eric Giguère)
- Fixed test_interpolate mcsolve use (#1645 by Eric Giguère)
- Ensured figure plots are explicitly closed during tests so that the test suite passes when run headless under Xvfb. (#1648 by Simon Cross)
- Bumped the version of pillow used to build documentation from 8.2.0 to 9.0.0. (#1654, #1760 by dependabot)
- Bumped the version of babel used to build documentation from 2.9.0 to 2.9.1. (#1695 by dependabot)
- Bumped the version of numpy used to build documentation from 1.19.5 to 1.21.0. (#1767 by dependabot)
- Bumped the version of ipython used to build documentation from 7.22.0 to 7.31.1. (#1780 by dependabot)
- Rename qutip.bib to CITATION.bib to enable GitHub's citation support. (#1662 by Ashish Panigrahi)
- Added tests for simdiags. (#1681 by Eric Giguère)
- Added support for specifying the numpy version in the CI test matrix. (#1696 by Simon Cross)
- Fixed the skipping of the dnorm metric tests if cvxpy is not installed. Previously all metrics tests were skipped by accident. (#1704 by Florian Hopfmueller)
- Added bug report, feature request and other options to the GitHub issue reporting template. (#1728 by Aryaman Kolhe)

- Updated the build process to support building on Python 3.10 by removing the build requirement for numpy < 1.20 and replacing it with a requirement on oldest-supported-numpy. (#1747 by Simon Cross)
- Updated the version of cibuildwheel used to build wheels to 2.3.0. (#1747, #1751 by Simon Cross)
- Added project urls to linking to the source repository, issue tracker and documentation to setup.cfg. (#1779 by Simon Cross)
- Added a numpy 1.22 and Python 3.10 build to the CI test matrix. (#1777 by Simon Cross)
- Ignore deprecation warnings from SciPy 1.8.0 scipy.sparse.X imports in CI tests. (#1797 by Simon Cross)
- Add building of wheels for Python 3.10 to the cibuildwheel job. (#1796 by Simon Cross)

6.4 Version 4.6.2 (June 2, 2021)

This minor release adds a function to calculate the quantum relative entropy, fixes a corner case in handling time-dependent Hamiltonians in `mesolve` and adds back support for a wider range of matplotlib versions when plotting or animating Bloch spheres.

It also adds a section in the README listing the papers which should be referenced while citing QuTiP.

6.4.1 Improvements

- Added a "Citing QuTiP" section to the README, containing a link to the QuTiP papers. (#1554)
- Added `entropy_relative` which returns the quantum relative entropy between two density matrices. (#1553)

6.4.2 Bug Fixes

- Fixed Bloch sphere distortion when using Matplotlib >= 3.3.0. (#1496)
- Removed use of integer-like floats in math.factorial since it is deprecated as of Python 3.9. (#1550)
- Simplified call to ffmpeg used in the the Bloch sphere animation tutorial to work with recent versions of ffmpeg. (#1557)
- Removed blitting in Bloch sphere FuncAnimation example. (#1558)
- Added a version checking condition to handle specific functionalities depending on the matplotlib version. (#1556)
- Fixed `mesolve` handling of time-dependent Hamiltonian with a custom tlist and `c_ops`. (#1561)

6.4.3 Developer Changes

- Read documentation version and release from the VERSION file.

6.5 Version 4.6.1 (May 4, 2021)

This minor release fixes bugs in QIP gate definitions, fixes building from the source tarball when git is not installed and works around an MKL bug in versions of SciPy <= 1.4.

It also adds the `[full]` pip install target so that `pip install qutip[full]` installs qutip and all of its optional and developer dependencies.

6.5.1 Improvements

- Add the `[full]` pip install target (by **Jake Lishman**)

6.5.2 Bug Fixes

- Work around pointer MKL eigh bug in SciPy <= 1.4 (by **Felipe Bivort Haiek**)
- Fix berkeley, swapalpha and cz gate operations (by **Boxi Li**)
- Expose the CPHASE control gate (by **Boxi Li**)
- Fix building from the sdist when git is not installed (by **Jake Lishman**)

6.5.3 Developer Changes

- Move the qutip-doc documentation into the qutip repository (by **Jake Lishman**)
- Fix warnings in documentation build (by **Jake Lishman**)
- Fix warnings in pytest runs and make pytest treat warnings as errors (by **Jake Lishman**)
- Add Simon Cross as author (by **Simon Cross**)

6.6 Version 4.6.0 (April 11, 2021)

This release brings improvements for qubit circuits, including a pulse scheduler, measurement statistics, reading/writing OpenQASM and optimisations in the circuit simulations.

This is the first release to have full binary wheel releases on pip; you can now do `pip install qutip` on almost any machine to get a correct version of the package without needing any compilers set up. The support for Numpy 1.20 that was first added in QuTiP 4.5.3 is present in this version as well, and the same build considerations mentioned there apply here too. If building using the now-supported PEP 517 mechanisms (e.g. `python -mbuild /path/to/qutip`), all build dependencies will be correctly satisfied.

6.6.1 Improvements

- **MAJOR** Add saving, loading and resetting functionality to `qutip.settings` for easy re-configuration. (by **Eric Giguère**)
- **MAJOR** Add a quantum gate scheduler in `qutip.qip.scheduler`, to help parallelise the operations of quantum gates. This supports two scheduling modes: as late as possible, and as soon as possible. (by **Boxi Li**)
- **MAJOR** Improved qubit circuit simulators, including OpenQASM support and performance optimisations. (by **Sidhant Saraogi**)
- **MAJOR** Add tools for quantum measurements and their statistics. (by **Simon Cross** and **Sidhant Saraogi**)

- Add support for Numpy 1.20. QuTiP should be compiled against a version of Numpy `>= 1.16.6` and `< 1.20` (note: does _not_ include 1.20 itself), but such an installation is compatible with any modern version of Numpy. Source installations from `pip` understand this constraint.
- Improve the error message when circuit plotting fails. (by **Boxi Li**)
- Add support for parsing M1 Mac hardware information. (by **Xiaoliang Wu**)
- Add more single-qubit gates and controlled gates. (by **Mateo Laguna** and **Martín Sande Costa**)
- Support decomposition of `X`, `Y` and `Z` gates in circuits. (by **Boxi Li**)
- Refactor `QubitCircuit.resolve_gate()` (by **Martín Sande Costa**)

6.6.2 Bug Fixes

- Fix `dims` in the returns from `Qobj.eigenstates` on superoperators. (by **Jake Lishman**)
- Calling Numpy ufuncs on `Qobj` will now correctly raise a `TypeError` rather than returning a nonsense `ndarray`. (by **Jake Lishman**)
- Convert segfault into Python exception when creating too-large tensor products. (by **Jake Lishman**)
- Correctly set `num_collapse` in the output of `mesolve`. (by **Jake Lishman**)
- Fix `ptrace` when all subspaces are being kept, or the subspaces are passed in order. (by **Jake Lishman**)
- Fix sorting bug in `Bloch3d.add_points()`. (by **pschindler**)
- Fix invalid string literals in docstrings and some unclosed files. (by **Élie Gouzien**)
- Fix Hermicity tests for matrices with values that are within the tolerance of 0. (by **Jake Lishman**)
- Fix the trace norm being incorrectly reported as 0 for small matrices. (by **Jake Lishman**)
- Fix issues with `dnorm` when using CVXPy 1.1 with sparse matrices. (by **Felipe Bivort Haiek**)
- Fix segfaults in `mesolve` when passed a bad initial `Qobj` as the state. (by **Jake Lishman**)
- Fix sparse matrix construction in PIQS when using Scipy 1.6.1. (by **Drew Parsons**)
- Fix `zspmv_openmp.cpp` missing from the pip sdist. (by **Christoph Gohlke**)
- Fix correlation functions throwing away imaginary components. (by **Asier Galicia Martinez**)
- Fix `QubitCircuit.add_circuit()` for SWAP gate. (by **Canoming**)
- Fix the broken LaTeX image conversion. (by **Jake Lishman**)
- Fix gate resolution of the FREDKIN gate. (by **Bo Yang**)
- Fix broken formatting in docstrings. (by **Jake Lishman**)

6.6.3 Deprecations

- `eseries`, `essolve` and `ode2es` are all deprecated, pending removal in QuTiP 5.0. These are legacy functions and classes that have been left unmaintained for a long time, and their functionality is now better achieved with `QobjEvo` or `mesolve`.

6.6.4 Developer Changes

- **MAJOR** Overhaul of setup and packaging code to make it satisfy PEP 517, and move the build to a matrix on GitHub Actions in order to release binary wheels on pip for all major platforms and supported Python versions. (by **Jake Lishman**)
- Default arguments in `Qobj` are now `None` rather than mutable types. (by **Jake Lishman**)
- Fixed comsumable iterators being used to parametrise some tests, preventing the testing suite from being re-run within the same session. (by **Jake Lishman**)
- Remove unused imports, simplify some floats and remove unnecessary list conversions. (by **jakobjakobson13**)
- Improve Travis jobs matrix for specifying the testing containers. (by **Jake Lishman**)
- Fix coverage reporting on Travis. (by **Jake Lishman**)
- Added a `pyproject.toml` file. (by **Simon Humpohl** and **Eric Giguère**)
- Add doctests to documentation. (by **Sidhant Saraogi**)
- Fix all warnings in the documentation build. (by **Jake Lishman**)

6.7 Version 4.5.3 (February 19, 2021)

This patch release adds support for Numpy 1.20, made necessary by changes to how array-like objects are handled. There are no other changes relative to version 4.5.2.

Users building from source should ensure that they build against Numpy versions >= 1.16.6 and < 1.20 (not including 1.20 itself), but after that or for those installing from conda, an installation will support any current Numpy version >= 1.16.6.

6.7.1 Improvements

- Add support for Numpy 1.20. QuTiP should be compiled against a version of Numpy `>= 1.16.6` and `< 1.20` (note: does _not_ include 1.20 itself), but such an installation is compatible with any modern version of Numpy. Source installations from `pip` understand this constraint.

6.8 Version 4.5.2 (July 14, 2020)

This is predominantly a hot-fix release to add support for Scipy 1.5, due to changes in private sparse matrix functions that QuTiP also used.

6.8.1 Improvements

- Add support for Scipy 1.5. (by **Jake Lishman**)
- Improved speed of `zcsr_inner`, which affects `Qobj.overlap`. (by **Jake Lishman**)
- Better error messages when installation requirements are not satisfied. (by **Eric Giguère**)

6.8.2 Bug Fixes

- Fix `zcsr_proj` acting on matrices with unsorted indices. (by **Jake Lishman**)
- Fix errors in Milstein's heterodyne. (by **Eric Giguère**)
- Fix datatype bug in `qutip.lattice` module. (by **Boxi Li**)
- Fix issues with `eigh` on Mac when using OpenBLAS. (by **Eric Giguère**)

6.8.3 Developer Changes

- Converted more of the codebase to PEP 8.
- Fix several instances of unsafe mutable default values and unsafe `is` comparisons.

6.9 Version 4.5.1 (May 15, 2020)

6.9.1 Improvements

- `husimi` and `wigner` now accept half-integer spin (by **maij**)
- Better error messages for failed string coefficient compilation. (issue raised by **nohchangsuk**)

6.9.2 Bug Fixes

- Safer naming for temporary files. (by **Eric Giguère**)
- Fix `clebsch` function for half-integer (by **Thomas Walker**)
- Fix `randint`'s dtype to `uint32` for compatibility with Windows. (issue raised by **Boxi Li**)
- Corrected stochastic's heterodyne's m_ops (by **eliegenois**)
- Mac pool use spawn. (issue raised by **goerz**)
- Fix typos in `QobjEvo._shift`. (by **Eric Giguère**)
- Fix warning on Travis CI. (by **Ivan Carvalho**)

6.9.3 Deprecations

- `qutip.graph` functions will be deprecated in QuTiP 5.0 in favour of `scipy.sparse.csgraph`.

6.9.4 Developer Changes

- Add Boxi Li to authors. (by **Alex Pitchford**)
- Skip some tests that cause segfaults on Mac. (by **Nathan Shammah** and **Eric Giguère**)
- Use Python 3.8 for testing on Mac and Linux. (by **Simon Cross** and **Eric Giguère**)

6.10 Version 4.5.0 (January 31, 2020)

6.10.1 Improvements

- **MAJOR FEATURE**: Added *qip.noise*, a module with pulse level description of quantum circuits allowing to model various types of noise and devices (by **Boxi Li**).
- **MAJOR FEATURE**: Added *qip.lattice*, a module for the study of lattice dynamics in 1D (by **Saumya Biswas**).
- Migrated testing from Nose to PyTest (by **Tarun Raheja**).
- Optimized testing for PyTest and removed duplicated test runners (by **Jake Lishman**).
- Deprecated importing *qip* functions to the qutip namespace (by **Boxi Li**).
- Added the possibility to define non-square superoperators relevant for quantum circuits (by **Arne Grimsmo** and **Josh Combes**).
- Implicit tensor product for *qeye*, *qzero* and *basis* (by **Jake Lishman**).
- QObjEvo no longer requires Cython for string coefficient (by **Eric Giguère**).
- Added marked tests for faster tests in *testing.run()* and made faster OpenMP benchmarking in CI (by **Eric Giguère**).
- Added entropy and purity for Dicke density matrices, refactored into more general dicke_trace (by **Nathan Shammah**).
- Added option for specifying resolution in Bloch.save function (by **Tarun Raheja**).
- Added information related to the value of hbar in *wigner* and *continuous_variables* (by **Nicolas Quesada**).
- Updated requirements for *scipy 1.4* (by **Eric Giguère**).
- Added previous lead developers to the qutip.about() message (by **Nathan Shammah**).
- Added improvements to *Qobj* introducing the *inv* method and making the partial trace, *ptrace*, faster, keeping both sparse and dense methods (by **Eric Giguère**).
- Allowed general callable objects to define a time-dependent Hamiltonian (by **Eric Giguère**).
- Added feature so that *QobjEvo* no longer requires Cython for string coefficients (by **Eric Giguère**).
- Updated authors list on Github and added *my binder* link (by **Nathan Shammah**).

6.10.2 Bug Fixes

- Fixed *PolyDataMapper* construction for *Bloch3d* (by **Sam Griffiths**).
- Fixed error checking for null matrix in essolve (by **Nathan Shammah**).
- Fixed name collision for parallel propagator (by **Nathan Shammah**).
- Fixed dimensional incongruence in *propagator* (by **Nathan Shammah**)
- Fixed bug by rewriting clebsch function based on long integer fraction (by **Eric Giguère**).
- Fixed bugs in QobjEvo's args depending on state and added solver tests using them (by **Eric Giguère**).
- Fixed bug in *sesolve* calculation of average states when summing the timeslot states (by **Alex Pitchford**).
- Fixed bug in *steadystate* solver by removing separate arguments for MKL and Scipy (by **Tarun Raheja**).
- Fixed *Bloch.add_ponts* by setting *edgecolor* = *None* in *plot_points* (by **Nathan Shammah**).
- Fixed error checking for null matrix in *essolve* solver affecting also *ode2es* (by **Peter Kirton**).
- Removed unnecessary shebangs in .pyx and .pxd files (by **Samesh Lakhotia**).

- Fixed *sesolve* and import of *os* in *codegen* (by **Alex Pitchford**).
- Updated *plot_fock_distribution* by removing the offset value 0.4 in the plot (by **Rajiv-B**).

6.11 Version 4.4.1 (August 29, 2019)

6.11.1 Improvements

- QobjEvo do not need to start from 0 anymore (by **Eric Giguère**).
- Add a quantum object purity function (by **Nathan Shammah** and **Shahnawaz Ahmed**).
- Add step function interpolation for array time-coefficient (by **Boxi Li**).
- Generalize expand_oper for arbitrary dimensions, and new method for cyclic permutations of given target cubits (by **Boxi Li**).

6.11.2 Bug Fixes

- Fixed the pickling but that made solver unable to run in parallel on Windows (Thank **lrunze** for reporting)
- Removed warning when mesolve fall back on sesolve (by **Michael Goerz**).
- Fixed dimension check and confusing documentation in random ket (by **Yariv Yanay**).
- Fixed Qobj isherm not working after using Qobj.permute (Thank **llorz1207** for reporting).
- Correlation functions call now properly handle multiple time dependant functions (Thank **taw181** for reporting).
- Removed mutable default values in mesolve/sesolve (by **Michael Goerz**).
- Fixed simdiag bug (Thank **Croydon-Brixton** for reporting).
- Better support of constant QobjEvo (by **Boxi Li**).
- Fixed potential cyclic import in the control module (by **Alexander Pitchford**).

6.12 Version 4.4.0 (July 03, 2019)

6.12.1 Improvements

- **MAJOR FEATURE**: Added methods and techniques to the stochastic solvers (by **Eric Giguère**) which allows to use a much broader set of solvers and much more efficiently.
- **MAJOR FEATURE**: Optimization of the montecarlo solver (by **Eric Giguère**). Computation are faster in many cases. Collapse information available to time dependant information.
- Added the QObjEvo class and methods (by **Eric Giguère**), which is used behind the scenes by the dynamical solvers, making the code more efficient and tidier. More built-in function available to string coefficients.
- The coefficients can be made from interpolated array with variable timesteps and can obtain state information more easily. Time-dependant collapse operator can have multiple terms.
- New wigner_transform and plot_wigner_sphere function. (by **Nithin Ramu**).
- ptrace is faster and work on bigger systems, from 15 Qbits to 30 Qbits.
- QIP module: added the possibility for user-defined gates, added the possibility to remove or add gates in any point of an already built circuit, added the molmer_sorensen gate, and fixed some bugs (by **Boxi Li**).
- Added the quantum Hellinger distance to qutip.metrics (by **Wojciech Rzadkowski**).

- Implemented possibility of choosing a random seed (by **Marek Marekyggdrasil**).
- Added a code of conduct to Github.

6.12.2 Bug Fixes

- Fixed bug that made QuTiP incompatible with SciPy 1.3.

6.13 Version 4.3.0 (July 14, 2018)

6.13.1 Improvements

- **MAJOR FEATURE**: Added the Permutational Invariant Quantum Solver (PIQS) module (by **Nathan Shammah** and **Shahnawaz Ahmed**) which allows the simluation of large TLSs ensembles including collective and local Lindblad dissipation. Applications range from superradiance to spin squeezing.
- **MAJOR FEATURE**: Added a photon scattering module (by **Ben Bartlett**) which can be used to study scattering in arbitrary driven systems coupled to some configuration of output waveguides.
- Cubic_Spline functions as time-dependent arguments for the collapse operators in mesolve are now allowed.
- Added a faster version of bloch_redfield_tensor, using components from the time-dependent version. About 3x+ faster for secular tensors, and 10x+ faster for non-secular tensors.
- Computing Q.overlap() [inner product] is now ~30x faster.
- Added projector method to Qobj class.
- 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.
- Unitary operator evolution added to sesolve
- Use OPENMP for tidyup if installed.

6.13.2 Bug Fixes

- Fixed bug that stopped simdiag working for python 3.
- Fixed semidefinite cvxpy Variable and Parameter.
- Fixed iterative lu solve atol keyword issue.
- Fixed unitary op evolution rhs matrix in ssesolve.
- Fixed interpolating function to return zero outside range.
- Fixed dnorm complex casting bug.
- Fixed control.io path checking issue.
- Fixed ENR fock dimension.
- Fixed hard coded options in propagator 'batch' mode
- Fixed bug in trace-norm for non-Hermitian operators.
- Fixed bug related to args not being passed to coherence_function_g2
- Fixed MKL error checking dict key error

6.14 Version 4.2.0 (July 28, 2017)

6.14.1 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.

6.14.2 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.

6.15 Version 4.1.0 (March 10, 2017)

6.15.1 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.

6.15.2 Bug Fixes

- Cleanup of temp. Cython files now more robust and working under Windows.

6.16 Version 4.0.2 (January 5, 2017)

6.16.1 Bug Fixes

- td files no longer left behind by correlation tests
- Various fast sparse fixes

6.17 Version 4.0.0 (December 22, 2016)

6.17.1 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

6.17.2 Bug Fixes

- recompilation of pyx files on first import removed
- tau array in control.pulseoptim funcs now works

6.18 Version 3.2.0 (Never officially released)

6.18.1 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.

6.18.2 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.

6.18.3 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`.

6.19 Version 3.1.0 (January 1, 2015)

6.19.1 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.

6.19.2 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.

6.20 Version 3.0.1 (Aug 5, 2014)

6.20.1 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.
-

6.21 Version 3.0.0 (July 17, 2014)

6.21.1 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.

6.21.2 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).

6.22 Version 2.2.0 (March 01, 2013)

6.22.1 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.

6.22.2 Bug Fixes

- Function based time-dependent Hamiltonians now keep the correct phase.
- mcsolve no longer prints to the command line if ntraj=1.

6.23 Version 2.1.0 (October 05, 2012)

6.23.1 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.

6.23.2 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.

6.24 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:

6.24.1 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.

6.25 Version 1.1.4 (May 28, 2012)

6.25.1 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+

6.26 Version 1.1.3 (November 21, 2011)

6.26.1 New Functions

- Allow custom naming of Bloch sphere.

6.26.2 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.

6.27 Version 1.1.2 (October 27, 2011)

6.27.1 Bug Fixes

- Fixed issue where Monte Carlo states were not output properly.

6.28 Version 1.1.1 (October 25, 2011)

THIS POINT-RELEASE INCLUDES VASTLY IMPROVED TIME-INDEPENDENT MCSOLVE AND ODESOLVE PERFORMANCE

6.28.1 New Functions

- Added linear entropy function.
- Number of CPU's can now be changed.

6.28.2 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.

6.29 Version 1.1.0 (October 04, 2011)

THIS RELEASE NOW REQUIRES THE GCC COMPILER TO BE INSTALLED

6.29.1 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.

6.29.2 Bug Fixes

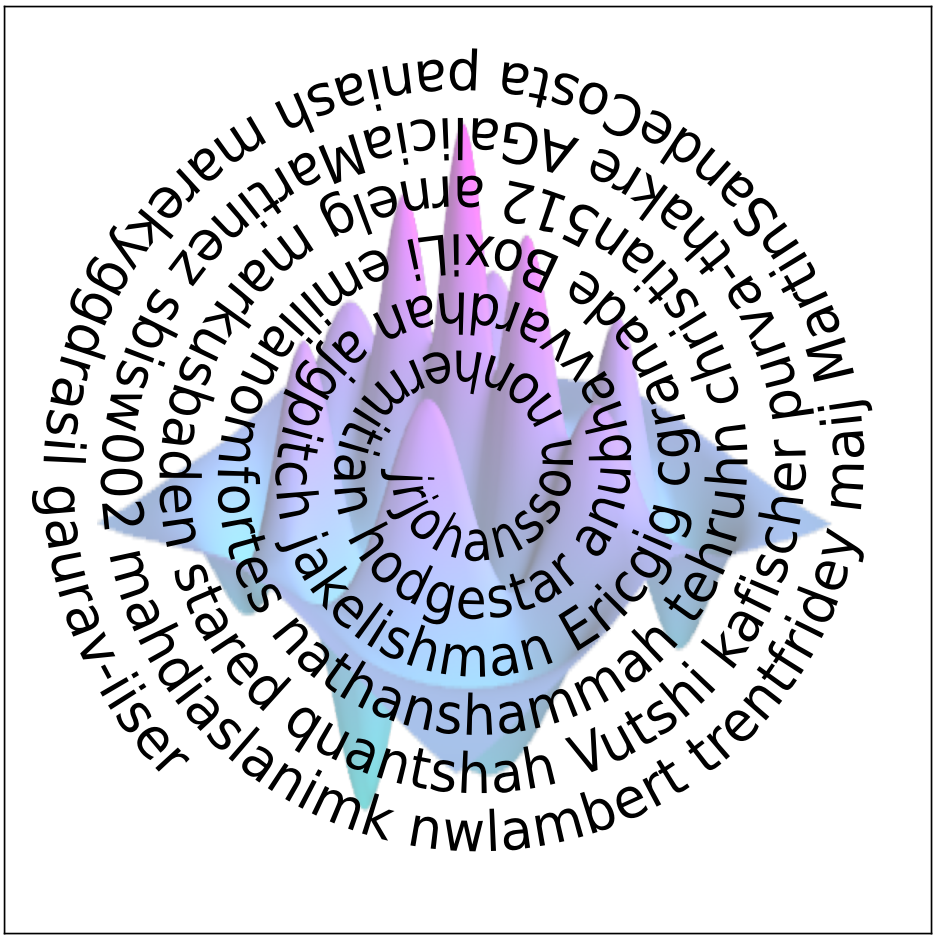
- Switched Examples to new Signals method used in PySide 1.0.6+.
- Switched ProgressBar to new Signals method.
- Fixed memory issue in expm functions.
- Fixed memory bug in isherm.
- Made all Qobj data complex by default.
- Reduced ODE tolerance levels in Odeoptions.
- Fixed bug in ptrace where dense 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).

6.30 Version 1.0.0 (July 29, 2011)

- **Initial release.**

Chapter 7

Developers



7.1 Lead Developers

- Alex Pitchford
- Nathan Shammah
- Shahnawaz Ahmed
- Neill Lambert
- Eric Giguère
- Boxi Li
- Jake Lishman
- Simon Cross

7.2 Past Lead Developers

- Robert Johansson (RIKEN)
- Paul Nation (Korea University)
- Chris Granade
- Arne Grimsmo

7.3 Contributors

Note: Anyone is welcome to contribute to QuTiP. If you are interested in helping, please let us know!

- Abhisek Upadhyaya
- Adriaan
- Alexander Pitchford
- Alexios-xi
- Amit
- Anubhav Vardhan
- Arie van Deursen
- Arne Grimsmo
- Arne Hamann
- Asier Galicia Martinez
- Ben Bartlett
- Ben Criger
- Ben Jones
- Bo Yang
- Boxi Li
- Canoming
- Christoph Gohlke

- Christopher Granade
- Craig Gidney
- Denis Vasilyev
- Dominic Meiser
- Drew Parsons
- Eric Giguère
- Eric Hontz
- Felipe Bivort Haiek
- Florestan Ziem
- Gilbert Shih
- Harry Adams
- Ivan Carvalho
- Jake Lishman
- Jevon Longdell
- Johannes Feist
- Jonas Hoersch
- Jonas Neergaard-Nielsen
- Jonathan A. Gross
- Julian Iacoponi
- Kevin Fischer
- Laurence Stant
- Louis Tessler
- Lucas Verney
- Marco David
- Marek Narozniak
- Markus Baden
- Martín Sande
- Mateo Laguna
- Matthew O'Brien
- Michael Goerz
- Michael V. DePalatis
- Moritz Oberhauser
- Nathan Shammah
- Neill Lambert
- Nicolas Quesada
- Nikolas Tezak
- Nithin Ramu
- Paul Nation
- Peter Kirton

- Philipp Schindler
- Piotr Migdal
- Rajiv-B
- Ray Ganardi
- Reinier Heeres
- Richard Brierley
- Robert Johansson
- Sam Griffiths
- Samesh Lakhotia
- Sebastian Krämer
- Shahnawaz Ahmed
- Sidhant Saraogi
- Simon Cross
- Simon Humpohl
- Simon Whalen
- Stefan Krastanov
- Tarun Raheja
- Thomas Walker
- Viacheslav Ostroukh
- Vlad Negnevitsky
- Wojciech Rzadkowski
- Xiaodong Qi
- Xiaoliang Wu
- Yariv Yanay
- YouWei Zhao
- alex
- eliegenois
- essence-of-waqf
- fhenneke
- gecrooks
- jakobjakobson13
- maij
- sbisw002
- yuri@FreeBSD
- Élie Gouzien

Chapter 8

Development Documentation

This chapter covers the development of QuTiP and its subpackages, including a roadmap for upcoming releases and ideas for future improvements.

8.1 Contributing to QuTiP Development

8.1.1 Quick Start

QuTiP is developed through wide collaboration using the `git` version-control system, with the main repositories hosted in the qutip organisation on GitHub. You will need to be familiar with `git` as a tool, and the GitHub Flow workflow for branching and making pull requests. The exact details of environment set-up, build process and testing vary by repository and are discussed below, however in overview, the steps to contribute are:

1. Consider creating an issue on the GitHub page of the relevant repository, describing the change you think should be made and why, so we can discuss details with you and make sure it is appropriate.
2. (If this is your first contribution.) Make a fork of the relevant repository on GitHub and clone it to your local computer. Also add our copy as a remote (`git remote add qutip https://github.com/qutip/<repo>`)
3. Begin on the `master` branch (`git checkout master`), and pull in changes from the main QuTiP repository to make sure you have an up-to-date copy (`git pull qutip master`).
4. Switch to a new `git` branch (`git checkout -b <branch-name>`).
5. Make the changes you want to make, then create some commits with short, descriptive names (`git add <files>` then `git commit`).
6. Follow the build process for this repository to build the final result so you can check your changes work sensibly.
7. Run the tests for the repository (if it has them).
8. Push the changes to your fork (`git push -u origin <branch-name>`). You won't be able to push to the main QuTiP repositories directly.
9. Go to the GitHub website for the repository you are contributing to, click on the "Pull Requests" tab, click the "New Pull Request" button, and follow the instructions there.

Once the pull request is created, some members of the QuTiP admin team will review the code to make sure it is suitable for inclusion in the library, to check the programming, and to ensure everything meets our standards. For some repositories, several automated tests will run whenever you create or modify a pull request; in general these will be the same tests you can run locally, and all tests are required to pass online before your changes are merged. There may be some feedback and possibly some requested changes. You can add more commits to address these, and push them to the relevant branch of your fork to update the pull request.

The rest of this document covers programming standards, and particular considerations for some of the more complicated repositories.

8.1.2 Core Library: qutip/qutip

The core library is in the qutip/qutip repository on GitHub.

Building

Building the core library from source is typically a bit more difficult than simply installing the package for regular use. You will most likely want to do this in a clean Python environment so that you do not compromise a working installation of a release version, for example by starting from

```
conda create -n qutip-dev python
```

Complete instructions for the build are elsewhere in this guide, however beware that you will need to follow the *installation from source using setuptools section*, not the general installation. You will need all the *build* and *tests* "optional" requirements for the package. The build requirements can be found in the `pyproject.toml` file, and the testing requirements are in the `tests` key of the `options.extras_require` section of `setup.cfg`. You will also need the requirements for any optional features you want to test as well.

Refer to the main instructions for the most up-to-date version, however as of version 4.6 the requirements can be installed into a conda environment with

```
conda install setuptools wheel 'numpy>=1.16.6,<1.20' 'scipy>=1.0' 'cython>=0.29.20
↪' packaging 'pytest>=5.2' pytest-rerunfailures
```

Note that `qutip` should *not* be installed with `conda install`.

Note: If you prefer, you can also use `pip` to install all the dependencies. We typically recommend `conda` when doing main-library development because it is easier to switch low-level packages around like BLAS implementations, but if this doesn't mean anything to you, feel free to use `pip`.

You will need to make sure you have a functioning C++ compiler to build QuTiP. If you are on Linux or Mac, this is likely already done for you, however if you are on Windows, refer to the *Windows installation* section of the installation guide.

The command to build QuTiP in editable mode is

```
python setup.py develop
```

from the repository directory. If you now load up a Python interpreter, you should be able to `import qutip` from anywhere as long as the correct Python environment is active. Any changes you make to the Python files in the git repository should be immediately present if you restart your Python interpreter and re-import `qutip`.

On the first run, the setup command will compile many C++ extension modules built from Cython sources (files ending `.pxd` and `.pyx`). Generally the low-level linear algebra routines that QuTiP uses are written in these files, not in pure Python. Unlike Python files, changes you make to Cython files will not appear until you run `python setup.py develop` again; you will only need to re-run this if you are changing Cython files. Cython will detect and compile only the files that have been changed, so this command will be faster on subsequent runs.

Note: When undertaking Cython development, the reason we use `python setup.py develop` instead of `pip install -e .` is because Cython's changed-file detection does not reliably work in the latter. `pip` tends to build in temporary virtual environments, which often makes Cython think its core library files have been updated, triggering a complete, slow rebuild of everything.

Code Style

The biggest concern you should always have is to make it easy for your code to be read and understood by the person who comes next.

All new contributions must follow PEP 8 style; all pull requests will be passed through a linter that will complain if you violate it. You should use the `pycodestyle` package locally (available on `pip`) to test you satisfy the requirements before you push your commits, since this is rather faster than pushing 10 different commits trying to fix minor niggles. Keep in mind that there is quite a lot of freedom in this style, especially when it comes to line breaks. If a line is too long, consider the *best* way to split it up with the aim of making the code readable, not just the first thing that doesn't generate a warning.

Try to stay consistent with the style of the surrounding code. This includes using the same variable names, especially if they are function arguments, even if these "break" PEP 8 guidelines. *Do not* change existing parameter, attribute or method names to "match" PEP 8; these are breaking user-facing changes, and cannot be made except in a new major release of QuTiP.

Other than this, general "good-practice" Python standards apply: try not to duplicate code; try to keep functions short, descriptively-named and side-effect free; provide a docstring for every new function; and so on.

Documenting

When you make changes in the core library, you should update the relevant documentation if needed. If you are making a bug fix, or other relatively minor changes, you will probably only need to make sure that the docstrings of the modified functions and classes are up-to-date; changes here will propagate through to the documentation the next time it is built. Be sure to follow the Numpy documentation standards (`numpydoc`) when writing docstrings. All docstrings will be parsed as reStructuredText, and will form the API documentation section of the documentation.

Testing

We use `pytest` as our test runner. The base way to run every test is

```
pytest /path/to/repo/qutip/tests
```

This will take around 10 to 30 minutes, depending on your computer and how many of the optional requirements you have installed. It is normal for some tests to be marked as "skip" or "xfail" in yellow; these are not problems. True failures will appear in red and be called "fail" or "error".

While prototyping and making changes, you might want to use some of the filtering features of `pytest`. Instead of passing the whole `tests` directory to the `pytest` command, you can also pass a list of files. You can also use the `-k` selector to only run tests whose names include a particular pattern, for example

```
pytest qutip/tests/test_qobj.py -k "expm"
```

to run the tests of `Qobj.expm`.

Changelog Generation

We use `towncrier` for tracking changes and generating a changelog. When making a pull request, we require that you add a towncrier entry along with the code changes. You should create a file named `<PR number>.<change type>` in the `doc/changes` directory, where the PR number should be substituted for `<PR number>`, and `<change type>` is either `feature`, `bugfix`, `doc`, `removal`, `misc`, or `deprecation`, depending on the type of change included in the PR.

You can also create this file by installing `towncrier` and running

    towncrier create <PR number>.<change type>

Running this will create a file in the `doc/changes` directory with a filename corresponding to the argument you passed to `towncrier create`. In this file, you should add a short description of the changes that the PR introduces.

8.1.3 Documentation: qutip/qutip (doc directory)

The core library is in the qutip/qutip repository on GitHub, inside the doc directory.

Building

The documentation is built using `sphinx`, `matplotlib` and `numpydoc`, with several additional extensions including `sphinx-gallery` and `sphinx-rtd-theme`. The most up-to-date instructions and dependencies will be in the `README.md` file of the documentation directory. You can see the rendered version of this file simply by going to the documentation GitHub page and scrolling down.

Building the documentation can be a little finnicky on occasion. You likely will want to keep a separate Python environment to build the documentation in, because some of the dependencies can have tight requirements that may conflict with your favourite tools for Python development. We recommend creating an empty `conda` environment containing only Python with

```
conda create -n qutip-doc python=3.8
```

and install all further dependencies with `pip`. There is a `requirements.txt` file in the repository root that fixes all package versions exactly into a known-good configuration for a completely empty environment, using

```
pip install -r requirements.txt
```

This known-good configuration was intended for Python 3.8, though in principle it is possible that other Python versions will work.

Note: We recommend you use `pip` to install dependencies for the documentation rather than `conda` because several necessary packages can be slower to update their `conda` recipes, so suitable versions may not be available.

The documentation build includes running many components of the main QuTiP library to generate figures and to test the output, and to generate all the API documentation. You therefore need to have a version of QuTiP available in the same Python environment. If you are only interested in updating the users' guide, you can use a release version of QuTiP, for example by running `pip install qutip`. If you are also modifying the main library, you need to make your development version accessible in this environment. See the *above section on building QuTiP* for more details, though the `requirements.txt` file will have already installed all the build requirements, so you should be able to simply run

```
python setup.py develop
```

in the main library repository.

The documentation is built by running the `make` command. There are several targets to build, but the most useful will be `html` to build the webpage documentation, `latexpdf` to build the PDF documentation (you will also need a full `pdflatex` installation), and `clean` to remove all built files. The most important command you will want to run is

```
make html
```

You should re-run this any time you make changes, and it should only update files that have been changed.

Important: The documentation build includes running almost all the optional features of QuTiP. If you get failure messages in red, make sure you have installed all of the optional dependencies for the main library.

The HTML files will be placed in the `_build/html` directory. You can open the file `_build/html/index.html` in your web browser to check the output.

Code Style

All user guide pages and docstrings are parsed by Sphinx using reStructuredText. There is a general Sphinx usage guide, which has a lot of information that can sometimes be a little tricky to follow. It may be easier just to look at other `.rst` files already in the documentation to copy the different styles.

Note: reStructuredText is a very different language to the Markdown that you might be familiar with. It's always worth checking your work in a web browser to make sure it's appeared the way you intended.

Testing

There are unfortunately no automated tests for the documentation. You should ensure that no errors appeared in red when you ran `make html`. Try not to introduce any new warnings during the build process. The main test is to open the HTML pages you have built (open `_build/html/index.html` in your web browser), and click through to the relevant pages to make sure everything has rendered the way you expected it to.

8.2 QuTiP Development Roadmap

8.2.1 Preamble

This document outlines plan and ideas for the current and future development of QuTiP. The document is maintained by the QuTiP Admim team. Contributuions from the QuTiP Community are very welcome.

In particular this document outlines plans for the next major release of qutip, which will be version 5. And also plans and dreams beyond the next major version.

There is lots of development going on in QuTiP that is not recorded in here. This a just an attempt at coordinated stragetgy and ideas for the future.

What is QuTiP?

The name QuTiP refers to a few things. Most famously, qutip is a Python library for simulating quantum dynamics. To support this, the library also contains various software tools (functions and classes) that have more generic applications, such as linear algebra components and visualisation utilities, and also tools that are specifically quantum related, but have applications beyond just solving dynamics (for instance partial trace computation).

QuTiP is also an organisation, in the Github sense, and in the sense of a group of people working collaboratively towards common objectives, and also a web presence qutip.org. The QuTiP Community includes all the people who have supported the project since in conception in 2010, including manager, funders, developers, maintainers and users.

These related, and overlapping, uses of the QuTiP name are of little consequence until one starts to consider how to organise all the software packages that are somehow related to QuTiP, and specifically those that are maintained by the QuTiP Admim Team. Herin QuTiP will refer to the project / organisation and qutip to the library for simulating quantum dyanmics.

Should we be starting again from scratch, then we would probably chose another name for the main qutip library, such as qutip-quantdyn. However, qutip is famous, and the name will stay.

8.2.2 Library package structure

With a name as general as Quantum Toolkit in Python, the scope for new code modules to be added to qutip is very wide. The library was becoming increasingly difficult to maintain, and in c. 2020 the QuTiP Admim Team decided to limit the scope of the 'main' (for want of a better name) qutip package. This scope is restricted to components for the simulation (solving) of the dynamics of quantum systems. The scope includes utilities to support this, including analysis and visualisation of output.

At the same time, again with the intention of easing maintence, a decision to limit dependences was agreed upon. Main qutip runtime code components should depend only upon Numpy and Scipy. Installation (from source) requires Cython, and some optional components also require Cython at runtime. Unit testing requires Pytest. Visualisation (optional) components require Matplotlib.

Due to the all encompassing nature of the plan to abstract the linear algebra data layer, this enhancement (developed as part of a GSoC project) was allowed the freedom (potential for non-backward compatibility) of requiring a major release. The timing of such allows for a restructuring of the qutip compoments, such that some that could be deemed out of scope could be packaged in a different way – that is, not installed as part of the main qutip package. Hence the proposal for different types of package described next. With reference to the *discussion above* on the name QuTiP/qutip, the planned restructuring suffers from confusing naming, which seems unavoidable without remaining either the organisation or the main package (neither of which are desirable).

QuTiP family packages The main qutip package already has sub-packages, which are maintained in the main qutip repo. Any packages maitained by the QuTiP organisation will be called QuTiP 'family' packages. Sub-packages within qutip main will be called 'integrated' sub-packages. Some packages will be maintained in their own repos and installed separately within the main qutip folder structure to provide backwards compatibility, these are (will be) called qutip optional sub-packages. Others will be installed in their own folders, but (most likely) have qutip as a dependency – these will just be called 'family' packages.

QuTiP affilliated packages Other packages have been developed by others outside of the QuTiP organisation that work with, and are complementary to, qutip. The plan is to give some recognition to those that we deem worthy of such [this needs clarification]. These packages will not be maintained by the QuTiP Team.

Family packages

qutip main

- **current package status**: family package *qutip*
- **planned package status**: family package *qutip*

The in-scope components of the main qutip package all currently reside in the base folder. The plan is to move some components into integrated subpackages as follows:

- *core* quantum objects and operations
- *solver* quantum dynamics solvers

What will remain in the base folder will be miscellaneous modules. There may be some opportunity for grouping some into a *visualisation* subpackage. There is also some potential for renaming, as some module names have underscores, which is unconventional.

Qtrl

- **current package status**: integrated sub-package *qutip.control*
- **planned package status**: family package *qtrl*

There are many OSS Python packages for quantum control optimisation. There are also many different algorithms. The current *control* integrated subpackage provides the GRAPE and CRAB algorithms. It is too ambitious for QuTiP to attempt (or want) to provide for all options. Control optimisation has been deemed out of scope and hence these components will be separated out into a family package called Qtrl.

Potentially Qtrl may be replaced by separate packages for GRAPE and CRAB, based on the QuTiP Control Framework.

QIP

- **current package status**: integrated sub-package *qutip.qip*
- **planned package status**: family package *qutip-qip*

The QIP subpackage has been deemed out of scope (feature-wise). It also depends on *qutip.control* and hence would be out of scope for dependency reasons. A separate repository has already been made for qutip-qip.

qutip-symbolic

- **current package status**: independent package *sympsi*
- **planned package status**: family package *qutip-symbolic*

Long ago Robert Johansson and Eunjong Kim developed Sympsi. It is a fairly coomplete library for quantum computer algebra (symbolic computation). It is primarily a quantum wrapper for Sympy.

It has fallen into unmaintained status. The latest version on the sympsi repo does not work with recent versions of Sympy. Alex Pitchford has a fork that does 'work' with recent Sympy versions – unit tests pass, and most examples work. However, some (important) examples fail, due to lack of respect for non-commuting operators in Sympy simplifcation functions (note this was true as of Nov 2019, may be fixed now).

There is a [not discussed with RJ & EK] plan to move this into the QuTiP family to allow the Admin Team to maintain, develop and promote it. The 'Sympsi' name is cute, but a little abstract, and qutip-symbolic is proposed as an alternative, as it is plainer and more distinct from Sympy.

Affilliated packages

qucontrol-krotov

- **code repository**: https://github.com/qucontrol/krotov

A package for quantum control optimisation using Krotov, developed mainly by Michael Goerz.

Generally accepted by the Admin Team as well developed and maintained. A solid candiate for affilliation.

8.2.3 Development Projects

Solver data layer integration

tag solve-dl

status development ongoing

admin lead Eric

main dev Eric

The new data layer gives opportunity for significantly improving performance of the qutip solvers. Eric has been revamping the solvers by deploying *QobjEvo* (the time-dependent quantum object) that he developed. *QobjEvo* will exploit the data layer, and the solvers in turn exploit *QobjEvo*.

Qtrl migration

tag qtrl-mig

status conceptualised

admin lead Alex

main dev TBA

The components currently packaged as an integrated subpackage of qutip main will be moved to separate package called Qtrl. This is the original codename of the package before it was integrated into qutip. Also changes to exploit the new data layer will be implemented.

QuTiP control framework

tag ctrl-fw

status conceptualised

admin lead Alex

main dev TBA

Create new package qutip-ctrlfw "QuTiP Control Framework". The aim is provide a common framework that can be adopted by control optimisation packages, such that different packages (algorithms) can be applied to the same problem.

Classes for defining a controlled system:

- named control parameters. Scalar and n-dim. Continuous and discrete variables
- mapping of control parameters to dynamics generator args
- masking for control parameters to be optimised

Classes for time-dependent variable parameterisation

- piecewise constant
- piecewise linear
- Fourier basis
- more

Classes for defining an optimisation problem:

- single and multiple objectives

QuTiP optimisation

tag qutip-optim

status conceptualised

admin lead Alex

main dev TBA

A wrapper for multi-variable optimisation functions. For instance those in *scipy.optimize* (Nelder-Mead, BFGS), but also others, such as Bayesian optimisation and other machine learning based approaches. Initially just providing a common interface for quantum control optimisation, but applicable more generally.

Sympsi migration

tag sympsi-mig

status conceptualised

admin lead Alex

main dev TBA

Create a new family package qutip-symbolic from ajgpitch fork of Sympy. Must gain permission from Robert Johansson and Eunjong Kim. Extended Sympy simplify to respect non-commuting operators. Produce user documentation.

Status messaging and recording

tag status-msg

status conceptualised

admin lead Alex

main dev TBA

QuTiP has various ways of recording and reporting status and progress.

- *ProgressBar* used by some solvers
- Python logging used in qutip.control
- *Dump* used in qutip.control
- heom records *solver.Stats*

Some consolidation of these would be good.

Some processes (some solvers, correlation, control optimisation) have many stages and many layers. *Dump* was initially developed to help with debugging, but it is also useful for recording data for analysis. qutip.logging_utils has been criticised for the way it uses Python logging. The output goes to stderr and hence the output looks like errors in Jupyter notebooks.

Clearly, storing process stage data is costly in terms of memory and cpu time, so any implementation must be able to be optionally switched on/off, and avoided completely in low-level processes (cythonized components).

Required features:

- optional recording (storing) of process stage data (states, operators etc)
- optionally write subsets to stdout
- maybe other graphical representations
- option to save subsets to file
- should ideally replace use of *ProgressBar*, Python logging, *control.Dump*, *solver.Stats*

qutip Interactive

status conceptualised

tag qutip-gui

admin lead Alex

main dev TBA

QuTiP is pretty simple to use at an entry level for anyone with basic Python skills. However, *some* Python skills are necessary. A graphical user interface (GUI) for some parts of qutip could help make qutip more accessible. This could be particularly helpful in education, for teachers and learners.

This would make an good GSoC project. It is independent and the scope is flexible.

The scope for this is broad and flexible. Ideas including, but not limited to:

Interactive Bloch sphere

Matplotlib has some interactive features (sliders, radio buttons, cmd buttons) that can be used to control parameters. They are a bit clunky to use, but they are there. Could maybe avoid these and develop our own GUI. An interactive Bloch sphere could have sliders for qubit state angles. Buttons to add states, toggle state evolution path.

Interactive solvers

Options to configure dynamics generators (Lindbladian / Hamiltonian args etc) and expectation operators. Then run solver and view state evolution.

Animated circuits

QIP circuits could be animated. Status lights showing evolution of states during the processing. Animated Bloch spheres for qubits.

8.2.4 Completed Development Projects

data layer abstraction

tag dl-abs

status completed

admin lead Eric

main dev Jake Lishman

Development completed as a GSoC project. Fully implemented in the dev.major branch. Currently being used by some research groups.

Abstraction of the linear algebra data from code qutip components, allowing for alternatives, such as sparse, dense etc. Difficult to summarize. Almost every file in qutip affected in some way. A major milestone for qutip. Significant performance improvements throughout qutip.

Some developments tasks remain, including providing full control over how the data-layer dispatchers choose the most appropriate output type.

qutip main reorganization

tag qmain-reorg

status completed

admin lead Eric

main dev Jake Lishman

Reorganise qutip main components to the structure *described above*.

qutip user docs migration

tag qmain-docs

status completed

admin lead Jake Lishman

main dev Jake Lishman

The qutip user documentation build files are to be moved to the qutip/qutip repo. This is more typical for an OSS package.

As part of the move, the plan is to reconstruct the Sphinx structure from scratch. Historically, there have been many issues with building the docs. Sphinx has come a long way since qutip docs first developed. The main source (rst) files will remain [pretty much] as they are, although there is a lot of scope to improve them.

The qutip-doc repo will afterwards just be used for documents, such as this one, pertaining to the QuTiP project.

QIP migration

tag qip-mig

status completed

admin lead Boxi

main dev Sidhant Saraogi

A separate package for qutip-qip was created during Sidhant's GSoC project. There is some fine tuning required, especially after qutip.control is migrated.

HEOM revamp

tag heom-revamp

status completed

admin lead Neill

main dev Simon Cross, Tarun Raheja

An overhaul of the HEOM solver, to incorporate the improvements pioneered in BoFiN.

8.2.5 QuTiP major release roadmap

QuTiP v.5

These Projects need to be completed for the qutip v.5 release.

- *data layer abstraction* (completed)
- *qutip main reorganization* (completed)
- *qutip user docs migration* (completed)
- *Solver data layer integration* (in-progress)
- *QIP migration* (completed)
- *Qtrl migration*
- *HEOM revamp* (completed)

The planned timeline for the release is:

- **alpha version, September 2022**. Core features packaged and available for experienced users to test.
- **beta version, November 2022**. All required features and documentation complete, packaged and ready for community testing.
- **full release, January 2023**. Full tested version released.

8.3 Ideas for future QuTiP development

Ideas for significant new features are listed here. For the general roadmap, see *QuTiP Development Roadmap*.

8.3.1 QuTiP Interactive

Contents

QuTiP is pretty simple to use at an entry level for anyone with basic Python skills. However, *some* Python skills are necessary. A graphical user interface (GUI) for some parts of qutip could help make qutip more accessible. This could be particularly helpful in education, for teachers and learners.

Ideally, interactive components could be embedded in web pages. Including, but not limited to, Jupyter notebooks.

The scope for this is broad and flexible. Ideas including, but not limited to:

Interactive Bloch sphere

QuTiP has a Bloch sphere virtualisation for qubit states. This could be made interactive through sliders, radio buttons, cmd buttons etc. An interactive Bloch sphere could have sliders for qubit state angles. Buttons to add states, toggle state evolution path. Potential for recording animations. Matplotlib has some interactive features (sliders, radio buttons, cmd buttons) that can be used to control parameters. that could potentially be used.

Interactive solvers

Options to configure dynamics generators (Lindbladian / Hamiltonian args etc) and expectation operators. Then run solver and view state evolution.

Animated circuits

QIP circuits could be animated. Status lights showing evolution of states during the processing. Animated Bloch spheres for qubits.

Expected outcomes

- Interactive graphical components for demonstrating quantum dynamics
- Web pages for qutip.org or Jupyter notebooks introducing quantum dynamics using the new components

Skills

- Git, Python and familiarity with the Python scientific computing stack
- elementary understanding of quantum dynamics

Difficulty

- Variable

Mentors

- Nathan Shammah (nathan.shammah@gmail.com)
- Alex Pitchford (alex.pitchford@gmail.com)
- Simon Cross (hodgestar@gmail.com)
- Boxi Li (etamin1201@gmail.com) [QuTiP GSoC 2019 graduate]

8.3.2 Pulse level description of quantum circuits

Contents

The aim of this proposal is to enhance QuTiP quantum-circuit compilation features with regard to quantum information processing. While QuTiP core modules deal with dynamics simulation, there is also a module for quantum circuits simulation. The two subsequent Google Summer of Code projects, in 2019 and 2020, enhanced them in capabilities and features, allowing the simulation both at the level of gates and at the level of time evolution. To connect them, a compiler is implemented to compile quantum gates into the Hamiltonian model. We would like to further enhance this feature in QuTiP and the connection with other libraries.

Expected outcomes

- APIs to import and export pulses to other libraries. Quantum compiler is a current research topic in quantum engineering. Although QuTiP has a simple compiler, many may want to try their own compiler which is more compatible with their quantum device. Allowing importation and exportation of control pulses will make this much easier. This will include a study of existing libraries, such as *qiskit.pulse* and *OpenPulse*[1], comparing them with *qutip.qip.pulse* module and building a more general and comprehensive description of the pulse.
- More examples of quantum system in the *qutip.qip.device* module. The circuit simulation and compilation depend strongly on the physical system. At the moment, we have two models: spin chain and cavity QED. We would like to include some other commonly used planform such as Superconducting system[2], Ion trap system[3] or silicon system. Each model will need a new set of control Hamiltonian and a compiler that finds the control pulse of a quantum gate. More involved noise models can also be added based on the physical system. This part is going to involve some physics and study of commonly used hardware platforms. The related code can be found in *qutip.qip.device* and *qutip.qip.compiler*.

Skills

- Git, Python and familiarity with the Python scientific computing stack
- quantum information processing and quantum computing (quantum circuit formalism)

Difficulty

- Medium

Mentors

- Boxi Li (etamin1201@gmail.com) [QuTiP GSoC 2019 graduate]
- Nathan Shammah (nathan.shammah@gmail.com)
- Alex Pitchford (alex.pitchford@gmail.com)

[1] McKay D C, Alexander T, Bello L, et al. Qiskit backend specifications for openqasm and openpulse experiments[J]. arXiv preprint arXiv:1809.03452, 2018.

[2] Häffner H, Roos C F, Blatt R, **Quantum computing with trapped ions**, Physics reports, 2008, 469(4): 155-203.

[3] Krantz P, Kjaergaard M, Yan F, et al. **A quantum engineer's guide to superconducting qubits**, Applied Physics Reviews, 2019, 6(2): 021318.

References

8.3.3 Quantum Error Mitigation

Contents

From the QuTiP 4.5 release, the qutip.qip module now contains the noisy quantum circuit simulator (which was a GSoC project) providing enhanced features for a pulse-level description of quantum circuits and noise models. A new class *Processor* and several subclasses are added to represent different platforms for quantum computing. They can transfer a quantum circuit into the corresponding control sequence and simulate the dynamics with QuTiP solvers. Different noise models can be added to *qutip.qip.noise* to simulate noise in a quantum device.

This module is still young and many features can be improved, including new device models, new noise models and integration with the existing general framework for quantum circuits (*qutip.qip.circuit*). There are also possible applications such as error mitigation techniques ([1],[2],[3]).

The tutorial notebooks can be found at https://qutip.org/tutorials.html#nisq. A recent presentation on the FOSDEM conference may help you get an overview (https://fosdem.org/2020/schedule/event/quantum_qutip/). See also the Github Project page for a collection of related issues and ongoing Pull Requests.

Expected outcomes

- Make an overview of existing libraries and features in error mitigation, similarly to a literature survey for a research article, but for a code project (starting from Refs.[4],[5]). This is done in order to best integrate the features in QuTiP with existing libraries and avoid reinventing the wheel.
- Features to perform error mitigation techniques in QuTiP, such as zero-noise extrapolation by pulse stretching.
- Tutorials implementing basic quantum error mitigation protocols
- Possible integration with Mitiq[6]

[1] Kristan Temme, Sergey Bravyi, Jay M. Gambetta, **Error mitigation for short-depth quantum circuits**, Phys. Rev. Lett. 119, 180509 (2017)

[2] Abhinav Kandala, Kristan Temme, Antonio D. Corcoles, Antonio Mezzacapo, Jerry M. Chow, Jay M. Gambetta, **Extending the computational reach of a noisy superconducting quantum processor**, Nature *567*, 491 (2019)

[3] S. Endo, S.C. Benjamin, Y. Li, **Practical quantum error mitigation for near-future applications**, Physical Review X *8*, 031027 (2018)

[4] Boxi Li's blog on the GSoC 2019 project on pulse-level control, https://gsoc2019-boxili.blogspot.com/

[5] Video of a recent talk on the GSoC 2019 project, https://fosdem.org/2020/schedule/event/quantum_qutip/

[6] Mitiq

Skills

- Background in quantum physics and quantum circuits.
- Git, python and familiarity with the Python scientific computing stack

Difficulty

- Medium

Mentors

- Nathan Shammah (nathan.shammah@gmail.com)
- Alex Pitchford (alex.pitchford@gmail.com)
- Eric Giguère (eric.giguere@usherbrooke.ca)
- Neill Lambert (nwlambert@gmail.com)
- Boxi Li (etamin1201@gmail.com) [QuTiP GSoC 2019 graduate]

References

8.3.4 GPU implementation of the Hierarchical Equations of Motion

Contents

The Hierarchical Equations of Motion (HEOM) method is a non-perturbative approach to simulate the evolution of the density matrix of dissipative quantum systems. The underlying equations are a system of coupled ODEs which can be run on a GPU. This will allow the study of larger systems as discussed in[1]. The goal of this project would be to extend QuTiP's HEOM method[2] and implement it on a GPU.

Since the method is related to simulating large, coupled ODEs, it can also be quite general and extended to other solvers.

[1] https://pubs.acs.org/doi/abs/10.1021/ct200126d?src=recsys&journalCode=jctcce
[2] https://arxiv.org/abs/2010.10806

Expected outcomes

- A version of HEOM which runs on a GPU.
- Performance comparison with the CPU version.
- Implement dynamic scaling.

Skills

- Git, python and familiarity with the Python scientific computing stack
- CUDA and OpenCL knowledge

Difficulty

- Hard

Mentors

- Neill Lambert (nwlambert@gmail.com)
- Alex Pitchford (alex.pitchford@gmail.com)
- Shahnawaz Ahmed (shahnawaz.ahmed95@gmail.com)
- Simon Cross (hodgestar@gmail.com)

References

8.3.5 Google Summer of Code

Many possible extensions and improvements to QuTiP have been documented as part of Google Summer of Code:

- GSoC 2021
- GSoC 2022

8.3.6 Completed Projects

These projects have been completed:

TensorFlow Data Backend

Contents

Note: This project was completed as part of GSoC 2021[3].

QuTiP's data layer provides the mathematical operations needed to work with quantum states and operators, i.e. `Qobj`, inside QuTiP. As part of Google Summer of Code 2020, the data layer was rewritten to allow new backends to be added more easily and for different backends to interoperate with each other. Backends using in-memory spares and dense matrices already exist, and we would like to add a backend that implements the necessary operations using TensorFlow[1].

Why a TensorFlow backend?

TensorFlow supports distributing matrix operations across multiple GPUs and multiple machines, and abstracts away some of the complexities of doing so efficiently. We hope that by using TensorFlow we might enable QuTiP to scale to bigger quantum systems (e.g. more qubits) and decrease the time taken to simulate them.

There is particular interest in trying the new backend with the BoFiN HEOM (Hierarchical Equations of Motion) solver[2].

Challenges

TensorFlow is a very different kind of computational framework to the existing dense and sparse matrix backends. It uses flow graphs to describe operations, and to work efficiently. Ideally large graphs of operations need to be executed together in order to efficiently compute results.

The QuTiP data layer might need to be adjusted to accommodate these differences, and it is possible that this will prove challenging or even that we will not find a reasonable way to achieve the desired performance.

Expected outcomes

- Add a `qutip.core.data.tensorflow` data type.
- Implement specialisations for some important operations (e.g. `add`, `mul`, `matmul`, `eigen`, etc).
- Write a small benchmark to show how `Qobj` operations scale on the new backend in comparison to the existing backends. Run the benchmark both with and without using a GPU.
- Implement enough for a solver to run on top of the new TensorFlow data backend and benchmark that (stretch goal).

Skills

- Git, Python and familiarity with the Python scientific computing stack
- Familiarity with TensorFlow (beneficial, but not required)
- Familiarity with Cython (beneficial, but not required)

[3] https://github.com/qutip/qutip-tensorflow/
[1] https://www.tensorflow.org/
[2] https://github.com/tehruhn/bofin

Difficulty

- Medium

Mentors

- Simon Cross (hodgestar@gmail.com)
- Jake Lishman (jake@binhbar.com)
- Alex Pitchford (alex.pitchford@gmail.com)

References

8.4 Working with the QuTiP Documentation

The user guide provides an overview of QuTiP's functionality. The guide is composed of individual reStructured-Text (`.rst`) files which each get rendered as a webpage. Each page typically tackles one area of functionality. To learn more about how to write `.rst` files, it is useful to follow the sphinx guide.

The documentation build also utilizes a number of Sphinx Extensions including but not limited to doctest, autodoc, sphinx gallery and plot. Additional extensions can be configured in the conf.py file.

8.4.1 Directives

There are two Sphinx directives that can be used to write code examples in the user guide:

- Doctest
- Plot

For a more comprehensive account of the usage of each directive, please refer to their individual pages. Here we outline some general guidelines on how to these directives while making a user guide.

Doctest

The doctest directive enables tests on interactive code examples. The simplest way to do this is by specifying a prompt along with its respective output:

```
.. doctest::

    >>> a = 2
    >>> a
    2
```

This is rendered in the documentation as follows:

```
>>> a = 2
>>> a
2
```

While specifying code examples under the `.. doctest::` directive, either all statements must be specified by the >>> prompt or without it. For every prompt, any potential corresponding output must be specified immediately after it. This directive is ideally used when there are a number of examples that need to be checked in quick succession.

A different way to specify code examples (and test them) is using the associated `.. testcode::` directive which is effectively a code block:

```
.. testcode::

    a = 2
    print(a)
```

followed by its results. The result can be specified with the `.. testoutput::` block:

```
.. testoutput::

    2
```

The advantage of the `testcode` directive is that it is a lot simpler to specify and amenable to copying the code to clipboard. Usually, tests are more easily specified with this directive as the input and output are specified in different blocks. The rendering is neater too.

Note: The `doctest` and `testcode` directives should not be assumed to have the same namespace.

Output:

```
a = 2
print(a)
```

```
2
```

A few notes on using the doctest extension:

- By default, each `testcode` and `doctest` block is run in a fresh namespace. To share a common namespace, we can specify a common group across the blocks (within a single `.rst` file). For example,

  ```
  .. doctest:: [group_name]

    >>> a = 2
  ```

  can be followed by some explanation followed by another code block sharing the same namespace

  ```
  .. doctest:: [group_name]

    >>> print(a)
    2
  ```

- To only print the code blocks (or the output), use the option `+SKIP` to specify the block without the code being tested when running `make doctest`.
- To check the result of a `Qobj` output, it is useful to make sure that spacing irregularities between the expected and actual output are ignored. For that, we can use the option `+NORMALIZE_WHITESPACE`.

Plot

Since the doctest directive cannot render matplotlib figures, we use Matplotlib's Plot directive when rendering to LaTeX or HTML.

The plot directive can also be used in the doctest format. In this case, when running doctests (which is enabled by specifying all statements with the >>> prompts), tests also include those specified under the plot directive.

Example:

```
First we specify some data:

.. plot::
```

(continues on next page)

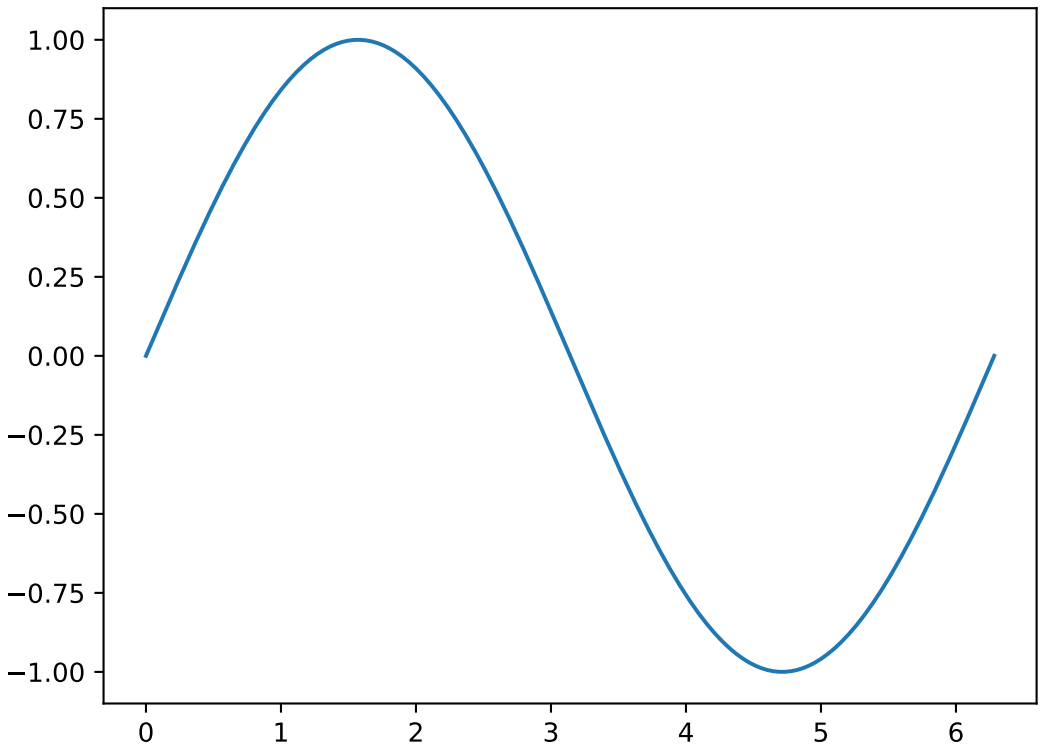
(continued from previous page)

```
  >>> import numpy as np
  >>> x = np.linspace(0, 2 * np.pi, 1000)
  >>> x[:10] # doctest: +NORMALIZE_WHITESPACE
  array([ 0.        ,  0.00628947,  0.01257895,  0.01886842,  0.0251579 ,
          0.03144737,  0.03773685,  0.04402632,  0.0503158 ,  0.05660527])


.. plot::
  :context:

  >>> import matplotlib.pyplot as plt
  >>> plt.plot(x, np.sin(x))
  [...]
```

Note the use of the `NORMALIZE_WHITESPACE` option to ensure that the multiline output matches.

Render:

```
>>> import numpy as np
>>> x = np.linspace(0, 2 * np.pi, 1000)
>>> x[:10]
array([ 0.        ,  0.00628947,  0.01257895,  0.01886842,  0.0251579 ,
        0.03144737,  0.03773685,  0.04402632,  0.0503158 ,  0.05660527])
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, np.sin(x))
[...]
```



A few notes on using the plot directive:

- A useful argument to specify in plot blocks is that of `context` which ensures that the code is being run in the namespace of the previous plot block within the same file.
- By default, each rendered figure in one plot block (when using `:context:`) is carried over to the next block.
- When the `context` argument is specified with the `reset` option as `:context: reset`, the namespace is reset to a new one and all figures are erased.
- When the `context` argument is specified with the `close-figs` option as `:context: reset`, the namespace is reset to a new one and all figures are erased.

The Plot directive cannot be used in conjunction with Doctest because they do not share the same namespace when used in the same file. Since Plot can also be used in doctest mode, in the case where code examples require both testing and rendering figures, it is easier to use the Plot directive. To learn more about each directive, it is useful to refer to their individual pages.

8.5 Release and Distribution

8.5.1 Preamble

This document covers the process for managing updates to the current minor release and making new releases. Within this document, the git remote `upstream` refers to the main QuTiP organsiation repository, and `origin` refers to your personal fork.

In short, the steps you need to take are:

1. Prepare the release branch (see *git*).
2. Run the "Build wheels, optionally deploy to PyPI" GitHub action to build binary and source packages and upload them to PyPI (see *deploy*).
3. Retrieve the built documentation from GitHub (see *docbuild*).
4. Create a GitHub release and uploaded the built files to it (see *github*).
5. Update qutip.org with the new links and documentation (*web*).
6. Update the conda feedstock, deploying the package to `conda` (*cforge*).

8.5.2 Setting Up The Release Branch

In this step you will prepare a git branch on the main QuTiP repository that has the state of the code that is going to be released. This procedure is quite different if you are releasing a new minor or major version compared to if you are making a bugfix patch release. For a new minor or major version, do *update-changelog* and then jump to *release*. For a bug fix to an existing release, do *update-changelog* and then jump to *bugfix*.

Changes that are not backwards-compatible may only be made in a major release. New features that do not affect backwards-compatibility can be made in a minor release. Bug fix releases should be small, only fix bugs, and not introduce any new features.

There are a few steps that *should* have been kept up-to-date during day-to-day development, but might not be quite accurate. For every change that is going to be part of your release, make sure that:

- The user guide in the documentation is updated with any new features, or changes to existing features.
- Any new API classes or functions have entries in a suitable RST file in `doc/apidoc`.
- Any new or changed docstrings are up-to-date and render correctly in the API documentation.

Please make a normal PR to `master` correcting anything missing from these points and have it merged before you begin the release, if necessary.

Updating the Changelog

This needs to be done no matter what type of release is being made.

1. Create a new branch to use to make a pull request.
2. Update the changelog using `towncrier`:

   towncrier build –version=<version-number>

Where `<version-number>` is the expected version number of the release

1. Make a pull request on the main `qutip/qutip` repository with this changelog, and get other members of the admin team to approve it.
2. Merge this into `master`.

Now jump to *release* if you are making a major or minor release, or *bugfix* if you are only fixing bugs in a previous release.

Create a New Minor or Major Release

This involves making a new branch to hold the release and adding some commits to set the code into "release" mode. This release should be done by branching directly off the `master` branch at its current head.

1. On your machine, make sure your copy of `master` is up-to-date (`git checkout master; git pull upstream master`). This should at least involve fetching the changelog PR that you just made. Now create a new branch off a commit in `master` that has the state of the code you want to release. The command is `git checkout -b qutip-<major>.<minor>.X`, for example `qutip-4.7.X`. This branch name will be public, and must follow this format.
2. Push the new branch (with no commits in it relative to `master`) to the main `qutip/qutip` repository (`git push upstream qutip-4.7.X`). Creating a branch is one of the only situations in which it is ok to push to `qutip/qutip` without making a pull request.
3. Create a second new branch, which will be pushed to your fork and used to make a pull request against the `qutip-<major>.<minor>.X` branch on `qutip/qutip` you just created. You can call this branch whatever you like because it is not going to the main repository, for example `git checkout -b prepare-qutip-4.7.0`.
4. - Change the `VERSION` file to contain the new version number exactly, removing the `.dev` suffix. For example, if you are releasing the first release of the minor 4.7 track, set `VERSION` to contain the string `4.7.0`. (*Special circumstances*: if you are making an alpha, beta or release candidate release, append a `.a<n>`, `.b<n>` or `.rc<n>` to the version string, where `<n>` is an integer starting from 0 that counts how many of that pre-release track there have been.)
   - Edit `setup.cfg` by changing the "Development Status" line in the `classifiers` section to

     ```
     Development Status :: 5 - Production/Stable
     ```

   Commit both changes (`git add VERSION setup.cfg; git commit -m "Set release mode for 4.7.0"`), and then push them to your fork (`git push -u origin prepare-qutip-4.7.0`)
5. Using GitHub, make a pull request to the release branch (e.g. `qutip-4.7.X`) using this branch that you just created. You will need to change the "base branch" in the pull request, because GitHub will always try to make the PR against `master` at first. When the tests have passed, merge this in.
6. Finally, back on `master`, make a new pull request that changes the `VERSION` file to be `<next-expected-version>.dev`, for example `4.8.0.dev`. The "Development Status" in `setup.cfg` on `master` should not have changed, and should be

   ```
   Development Status :: 2 - Pre-Alpha
   ```

   because `master` is never directly released.

You should now have a branch that you can see on the GitHub website that is called `qutip-4.7.X` (or whatever minor version), and the state of the code in it should be exactly what you want to release as the new minor release. If you notice you have made a mistake, you can make additional pull requests to the release branch to fix it. `master` should look pretty similar, except the `VERSION` will be higher and have a `.dev` suffix, and the "Development Status" in `setup.cfg` will be different.

You are now ready to actually perform the release. Go to *deploy*.

Create a Bug Fix Release

In this you will modify an already-released branch by "cherry-picking" one or more pull requests that have been merged to `master` (including your new changelog), and bump the "patch" part of the version number.

1. On your machine, make sure your copy of `master` is up-to-date (`git checkout master; git pull upstream master`). In particular, make sure the changelog you wrote in the first step is visible.
2. Find the branch of the release that you will be modifying. This should already exist on the `qutip/qutip` repository, and be called `qutip-<major>.<minor>.X` (e.g. `qutip-4.6.X`). If you cannot see it, run `git fetch upstream` to update all the branch references from the main repository. Checkout a new private branch, starting from the head of the release branch (`git checkout -b prepare-qutip-4.6.1 upstream/qutip-4.6.X`). You can call this branch whatever you like (in the example it is `prepare-qutip-4.6.1`), because it will only be used to make a pull request.
3. Cherry-pick all the commits that will be added to this release in order, including your PR that wrote the new changelog entries (this will be the last one you cherry-pick). You will want to use `git log` to find the relevant commits, going from **oldest to newest** (their "age" is when they were merged into `master`, not when the PR was first opened). The command is slightly different depending on which merge strategy was used for a particular PR:
   - "merge": you only need to find one commit though the log will have included several; there will be an entry in `git log` with a title such as "Merge pull request #1000 from <...>". Note the first 7 characters of its hash. Cherry-pick this by `git cherry-pick --mainline 1 <hash>`.
   - "squash and merge": there will only be a single commit for the entire PR. Its name will be "<Name of the pull request> (#1000)". Note the first 7 characters of its hash. Cherry-pick this by `git cherry-pick <hash>`.
   - "rebase and merge": this is the most difficult, because there will be many commits that you will have to find manually, and cherry-pick all of them. Go to the GitHub page for this PR, and go to the "Commits" tab. Using your local `git log` (you may find `git log --oneline` useful), find the hash for every single commit that is listed on the GitHub page, in order from **oldest to newest** (top-to-bottom in the GitHub view, which is bottom-to-top in `git log`). You will need to use the commit message to do this; the hashes that GitHub reports will probably not be the same as how they appear locally. Find the first 7 characters of each of the hashes. Cherry-pick these all in one go by `git cherry-pick <hash1> <hash2> ... <hash10>`, where `<hash1>` is the oldest.

   If any of the cherry-picks have merge conflicts, first verify that you are cherry-picking in order from oldest to newest. If you still have merge conflicts, you will either need to manually fix them (if it is a *very* simple fix), or else you will need to find which additional PR this patch depends on, and restart the bug fix process including this additional patch. This generally should not happen if you are sticking to very small bug fixes; if the fixes had far-reaching changes, a new minor release may be more appropriate.
4. Change the `VERSION` file by bumping the last number up by one (double-digit numbers are fine, so `4.6.10` comes after `4.6.9`), and commit the change.
5. Push this branch to your fork, and make a pull request against the release branch. On GitHub in the PR screen, you will need to change the "Base" branch to `qutip-4.6.X` (or whatever version), because GitHub will default to making it against `master`. It should be quite clear if you have forgotten to do this, because there will probably be many merge conflicts. Once the tests have passed and you have another admin's approval, merge the PR.

You should now see that the `qutip-4.6.X` (or whatever) branch on GitHub has been updated, and now includes all the changes you have just made. If you have made a mistake, feel free to make additonal PRs to rectify the situation.

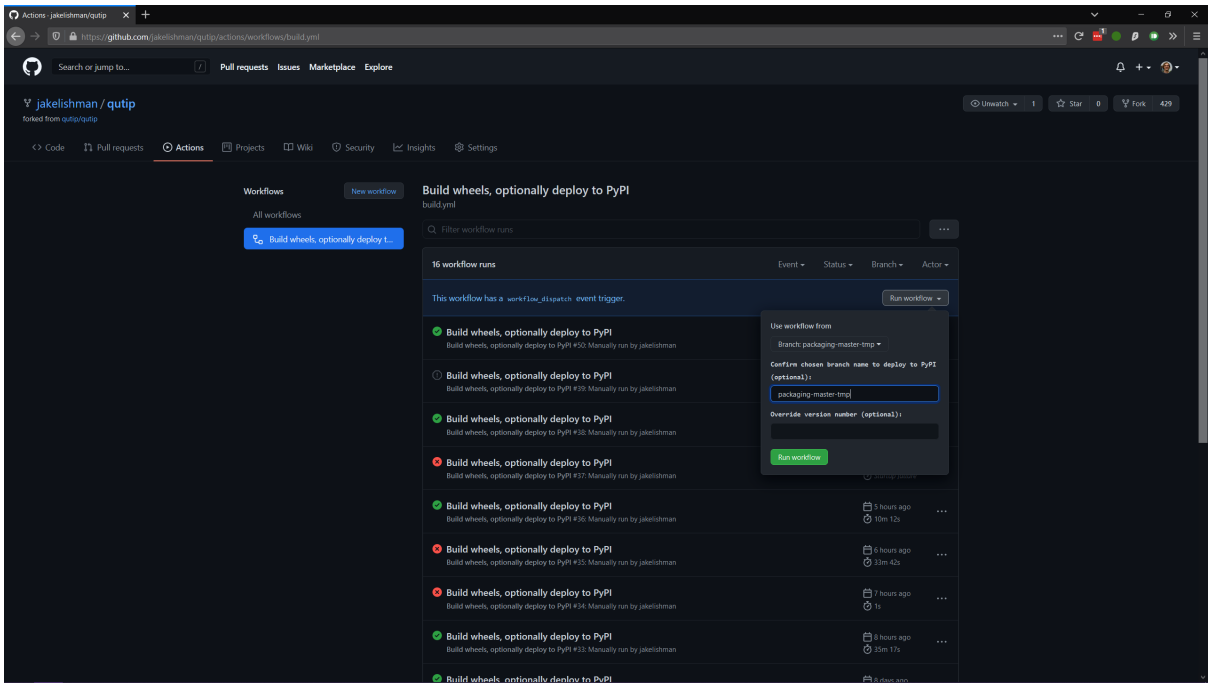
You are now ready to actually perform the release. Go to *deploy*.

8.5.3 Build Release Distribution and Deploy

This step builds the source (sdist) and binary (wheel) distributions, and uploads them to PyPI (pip). You will also be able to download the built files yourself in order to upload them to the QuTiP website.

Build and Deploy

This is handled entirely by a GitHub Action. Go to the "Actions" tab at the top of the QuTiP code repository. Click on the "Build wheels, optionally deploy to PyPI" action in the left-hand sidebar. Click the "Run workflow" dropdown in the header notification; it should look like the image below.

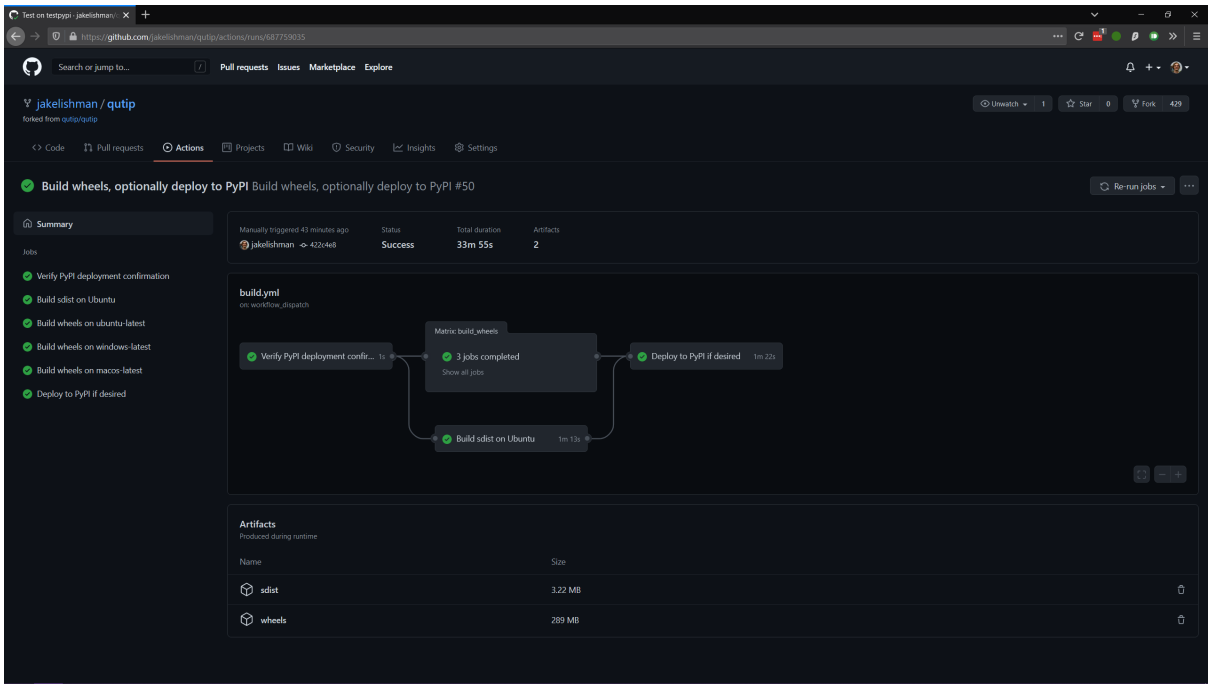


- Use the drop-down menu to choose the branch or tag you want to release from. This should be called `qutip-4.5.X` or similar, depending on what you made earlier. This must *never* be `master`.
- To make the release to PyPI, type the branch name (e.g. `qutip-4.5.X`) into the "Confirm chosen branch name [...]" field. You *may* leave this field blank to skip the deployment and only build the package.
- (Special circumstances) If for some reason you need to override the version number (for example if the previous deployment to PyPI only partially succeeded), you can type a valid Python version identifier into the "Override version number" field. You probably do not need to do this. The mechanism is designed to make alpha-testing major upgrades with nightly releases easier. For even a bugfix release, you should commit the change to the `VERSION` file.
- Click the lower "Run workflow" to perform the build and deployment.

At this point, the deployment will take care of itself. It should take between 30 minutes and an hour, after which the new version will be available for install by `pip install qutip`. You should see the new version appear on QuTiP's PyPI page.

Download Built Files

When the build is complete, click into its summary screen. This is the main screen used to both monitor the build and see its output, and should look like the below image on a success.



The built binary wheels and the source distribution are the "build artifacts" at the bottom. You need to download both the wheels and the source distribution. Save them on your computer, and unzip both files; you should have many wheel `qutip-*.whl` files, and two sdist files: `qutip-*.tar.gz` and `qutip-*.zip`. These are the same files that have just been uploaded to PyPI.

Monitoring Progress (optional)

While the build is in progress, you can monitor its progress by clicking on its entry in the list below the "Run workflow" button. You should see several subjobs, like the completed screen, except they might not yet be completed.

The "Verify PyPI deployment confirmation" should get ticked, no matter what. If it fails, you have forgotten to choose the correct branch in the drop-down menu or you made a typo when confirming the correct branch, and you will need to restart this step. You can check that the deployment instruction has been understood by clicking the "Verify PyPI deployment confirmation" job, and opening the "Compare confirmation to current reference" subjob. You will see a message saying "Built wheels will be deployed" if you typed in the confirmation, or "Only building wheels" if you did not. If you see "Only building wheels" but you meant to deploy the release to PyPI, you can cancel the workflow and re-run it after typing the confirmation.

8.5.4 Getting the Built Documentation

The documentation will have been built automatically for you by a GitHub Action when you merged the final pull request into the release branch before building the wheels. You do not need to re-release the documentation on either GitHub or the website if this is a patch release, unless there were changes within it.

Go to the "Actions" tab at the top of the `qutip/qutip` repository, and click the "Build HTML documentation" heading in the left column. You should see a list of times this action has run; click the most recent one whose name is exactly "Build HTML documentation", with the release branch name next to it (e.g. `qutip-4.6.X`). Download the `qutip_html_docs` artifact to your local machine and unzip it somewhere safe. These are all the HTML files for the built documentation; you should be able to open `index.html` in your own web browser and check that everything is working.

8.5.5 Making a Release on GitHub

This is all done through the "Releases" section of the `qutip/qutip` repository on GitHub.

- Click the "Draft a new release" button.
- Choose the correct branch for your release (e.g. `qutip-4.5.X`) in the drop-down.
- For the tag name, use `v<your-version>`, where the version matches the contents of the `VERSION` file. In other words, if you are releasing a micro version 4.5.3, use `v4.5.3` as the tag, or if you are releasing major version 5.0.0, use `v5.0.0`.
- The title is "QuTiP <your-version>", e.g. "QuTiP 4.6.0".
- For the description, write a short (~two-line for a patch release) summary of the reason for this release, and note down any particular user-facing changes that need special attention. Underneath, put the changelog you wrote when you did the documentation release. Note that there may be some syntax differences between the `.rst` file of the changelog and the Markdown of this description field (for example, GitHub's markdown typically maintains hard-wrap linebreaks, which is probably not what you wanted).
- Drag-and-drop all the `qutip-*.whl`, `qutip-*.tar.gz` and `qutip-*.zip` files you got after the build step into the assets box. You may need to unzip the files `wheels.zip` and `sdist.zip` to find them if you haven't already; **don't** upload those two zip files.

Click on the "Publish release" button to finalise.

8.5.6 Website

This assumes that qutip.github.io has already been forked and familiarity with the website updating workflow. The documentation need not be updated for every patch release.

Copying New Files

You only need to copy in new documentation to the website repository. Do not copy the `.whl`, `.tar.gz` or `.zip` files into the git repository, because we can access the public links from the GitHub release stage, and this keeps the website `.git` folder a reasonable size.

For all releases move (no new docs) or copy (for new docs) the `qutip-doc-<MAJOR>.<MINOR>.pdf` into the folder `downloads/<MAJOR>.<MINOR>.<MICRO>`.

The legacy html documentation should be in a subfolder like

```
docs/<MAJOR>.<MINOR>
```

For a major or minor release the previous version documentation should be moved into this folder.

The latest version HTML documentation should be the folder

```
docs/latest
```

For any release which new documentation is included - copy the contents `qutip/doc/_build/html` into this folder. **Note that the underscores at start of the subfolder names will need to be removed, otherwise Jekyll will ignore the folders**. There is a script in the `docs` folder for this. https://github.com/qutip/qutip.github.io/blob/master/docs/remove_leading_underscores.py

HTML File Updates

- Edit `download.html`
  - The 'Latest release' version and date should be updated.
  - The tar.gz and zip links need to have their micro release numbers updated in their filenames, labels and trackEvent javascript. These links should point to the "Source code" links that appeared when you made in the GitHub Releases section. They should look something like `https://github.com/qutip/qutip/archive/refs/tags/v4.6.0.tar.gz`.
  - For a minor or major release links to the last micro release of the previous version will need to be moved (copied) to the 'Previous releases' section.
- Edit `_includes/sidebar.html`
  - The 'Latest release' version should be updated. The gztar and zip file links will need the micro release number updating in the traceEvent and file name.
  - The link to the documentation folder and PDF file (if created) should be updated.
- Edit `documentation.html`
  - The previous release tags should be moved (copied) to the 'Previous releases' section.

8.5.7 Conda Forge

If not done previously then fork the qutip-feedstock.

Checkout a new branch on your fork, e.g.

```
$ git checkout -b version-4.0.2
```

Find the sha256 checksum for the tarball that the GitHub web interface generated when you produced the release called "Source code". This is *not* the sdist that you downloaded earlier, it's a new file that GitHub labels "Source code". When you download it, though, it will have a name that *looks* like it's the sdist

```
$ openssl sha256 qutip-4.0.2.tar.gz
```

Edit the `recipe/meta.yaml` file. Change the version at the top of the file, and update the sha256 checksum. Check that the recipe package version requirements at least match those in `setup.cfg`, and that any changes to the build process are reflected in `meta.yml`. Also ensure that the build number is reset

```
build:
    number: 0
```

Push changes to your fork, e.g.

```
$ git push --set-upstream origin version-4.0.2
```

Make a Pull Request. This will trigger tests of the package build process.

If (when) the tests pass, the PR can be merged, which will trigger the upload of the packages to the conda-forge channel. To test the packages, add the conda-forge channel with lowest priority

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

This should mean that the prerequistes come from the default channel, but the qutip packages are found in conda-forge.

Chapter 9

Bibliography

Chapter 10

Copyright and Licensing

The text of this documentation is licensed under the Creative Commons Attribution 3.0 Unported License. Unless specifically indicated otherwise, all code samples, the source code of QuTiP, and its reproductions in this documentation, are licensed under the terms of the 3-clause BSD license, reproduced below.

10.1 License Terms for Documentation Text

The canonical form of this license is available at https://creativecommons.org/licenses/by/3.0/, which should be considered the binding version of this license. It is reproduced here for convenience.

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

    a. "Adaptation" means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

    b. "Collection" means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for the purposes of this License.

    c. "Distribute" means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.

    d. "Licensor" means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

e. "Original Author" means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

f. "Work" means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

g. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

h. "Publicly Perform" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

i. "Reproduce" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

    a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

    b. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

    c. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

    d. to Distribute and Publicly Perform Adaptations.

    For the avoidance of doubt:

    i. Non-waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

ii. Waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,

iii. Voluntary License Schemes. The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

   a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(b), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(b), as requested.

   b. If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv) , consistent with Section 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4 (b) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

   c. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation,

modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5. Representations, Warranties and Disclaimer

   UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

   a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

   b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

   a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

   b. Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

   c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

   d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

   e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

   f. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in

which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.

10.2 License Terms for Source Code of QuTiP and Code Samples

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

Chapter 11

Indices and tables

- genindex
- modindex
- search

Bibliography

[LACN19] Neill Lambert, Shahnawaz Ahmed, Mauro Cirio, and Franco Nori. Virtual excitations in the ultra-strongly-coupled spin-boson model: physical results from unphysical modes. *arXiv preprint arXiv:1903.05892*, 2019.

[Tan20] Yoshitaka Tanimura. Numerically "exact" approach to open quantum dynamics: the hierarchical equations of motion (heom). *The Journal of Chemical Physics*, 153(2):020901, 2020. URL: https://doi.org/10.1063/5.0011599, doi:10.1063/5.0011599.

[TK89] Yoshitaka Tanimura and Ryogo Kubo. Time evolution of a quantum system in contact with a nearly gaussian-markoffian noise bath. *J. Phys. Soc. Jpn.*, 58(1):101–114, 1989. doi:10.1143/jpsj.58.101.

[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).

[1] Shore, B. W., "The Theory of Coherent Atomic Excitation", Wiley, 1990.

[1] https://en.wikipedia.org/wiki/Concurrence_(quantum_computing)

[1] Peter E. Kloeden and Exkhard Platen, *Numerical Solution of Stochastic Differential Equations*.

[2] H.-P. Breuer and F. Petruccione, *The Theory of Open Quantum Systems*.

[3] Pierre Rouchon and Jason F. Ralpha, *Efficient Quantum Filtering for Quantum Feedback Control*, arXiv:1410.5345 [quant-ph], Phys. Rev. A 91, 012118, (2015).

[4] Howard M. Wiseman, Gerard J. Milburn, *Quantum measurement and control*.

[1] J. Rodriguez-Laguna, P. Migdal, M. Ibanez Berganza, M. Lewenstein and G. Sierra, *Qubism: self-similar visualization of many-body wavefunctions*, New J. Phys. 14 053028, arXiv:1112.3560 (2012), open access.

[BCSZ08] W. 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] F. Mezzadri, *How to generate random matrices from the classical compact groups*, Notices of the AMS **54** 592-604 (2007). arXiv:math-ph/0609050.

[Moh08] M. 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] M. Grifoni, P. Hänggi, *Driven quantum tunneling*, Physics Reports **304**, 299 (1998). doi:10.1016/S0370-1573(98)00022-2.

[Gar03] Gardineer and Zoller, *Quantum Noise* (Springer, 2004).

[Bre02] H.-P. Breuer and F. Petruccione, *The Theory of Open Quantum Systems* (Oxford, 2002).

[Coh92] C. 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] D. d'Alessandro, *Introduction to Quantum Control and Dynamics*, (Chapman & Hall/CRC, 2008).

[Byrd95] R. H. 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] F. F. 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] S. Lloyd and S. Montangero, *Information theoretical analysis of quantum optimal control*, Phys. Rev. Lett. **113**, 010502 (2014). doi:10.1103/PhysRevLett.113.010502

[Doria11] P. 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] T. Caneva, T. Calarco, & S. Montangero, *Chopped random-basis quantum optimization*, Phys. Rev. A **84**, 022326 (2011). doi:10.1103/PhysRevA.84.022326

[Rach15] N. 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] S. 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

[Wis09] Wiseman, H. M. & Milburn, G. J. *Quantum Measurement and Control*, (Cambridge University Press, 2009).

[NKanej] 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). doi:10.1016/j.jmr.2004.11.004

Python Module Index

q

Index

A

B

C

D

E

F

G

N

O

P

Q

R

S

T

U

V

W

X

Z