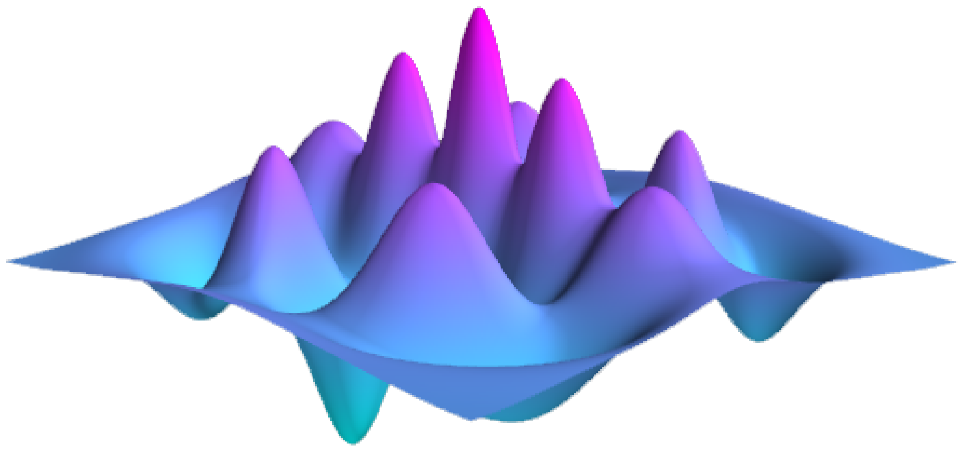
qutip_qtrl

Release 0.2.0.dev

QuTiP Community

May 08, 2024

INTRODUCTION AND INSTALLATION



CHAPTER

ONE

INTRODUCTION

1.1 The qutip-qtrl package

The qutip-qtrl package used to be a module `qutip.control` under QuTiP (Quantum Toolbox in Python). From QuTiP 5.0, the community has decided to decrease the size of the core QuTiP package by reducing the external dependencies, in order to simplify maintenance. Hence a few modules are separated from the core QuTiP and will become QuTiP family packages. They are still maintained by the QuTiP team but hosted under different repositories in the QuTiP organization.

The qutip-qtrl package, QuTiP quantum optimal control, aims at providing advanced tools for the optimal control of quantum devices. Compared to other libraries for quantum optimal control, qutip-qtrl puts additional emphasis on the physics layer and the interaction with the QuTiP package. The package offers support for both the CRAB and GRAPE methods.

1.2 Citing

If you use *qutip-qtrl* in your research, please cite the original QuTiP papers that are available [here](https://dml.riken.jp/?s=QuTiP).

CHAPTER

TWO

INSTALLATION

2.1 Quick start

To install the package `qutip-qtrl` from PyPI, use

```
pip install qutip-qtrl
```

2.2 Migrating from `qutip.control`

As the *Introduction* suggested, this package is based on a module in the QuTiP package `qutip.control`. If you were using the `qutip` package and now want to try out the new features included in this package, you can simply install this package and replace all the `qutip.control` in your import statement with `qutip_qtrl`. Everything should work smoothly as usual.

2.3 Prerequisites

This package is built upon QuTiP, of which the installation guide can be found at on QuTiP Installation.

In particular, following packages are necessary for running qutip-qtrl:

```
numpy scipy cython qutip
```

The following to packages are used for plotting and testing:

```
matplotlib pytest
```

In addition,

```
sphinx numpydoc sphinx_rtd_theme
```

are used to build and test the documentation.

2.4 Install qutip-qtrl from source code

To install the package, download to source code from GitHub website and run

```
pip install .
```

under the directory containing the `setup.cfg` file.

If you want to edit the code, use instead

```
pip install -e .
```

To test the installation from a download of the source code, run from the *qutip-qtrl* directory

` pytest tests `

CHAPTER

THREE

QUANTUM OPTIMAL CONTROL

3.1 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* [1]. 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 [1]. The solutions to the second question can be determined through optimal control algorithms, or control pulse optimisation.

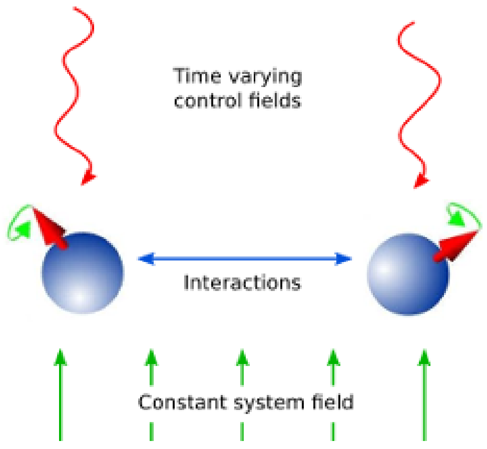


Fig. 1: Schematic showing the principle of quantum control.

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.

3.2 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*.

$$\tfrac{d}{dt}\psi = -iH(t)\psi$$

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

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

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 ψ_0 to ψ_1, this is state-to-state transfer, or drive the system from some arbitary state to a given state ψ_1, which is state preparation, or effect some unitary transformation U_{target}, called gate synthesis. The latter of these is most important in quantum computation.

3.3 The GRAPE algorithm

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

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

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

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

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

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

If the objective is state-to-state transfer then $X_0 = \psi_0$ and the target $X_{targ} = \psi_1$, 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.

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

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 [3]. 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 [4]. 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.

3.4 The CRAB Algorithm

It has been shown [5], 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 [6], [7] 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 [8] 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.

3.5 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 [9]. 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*

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.

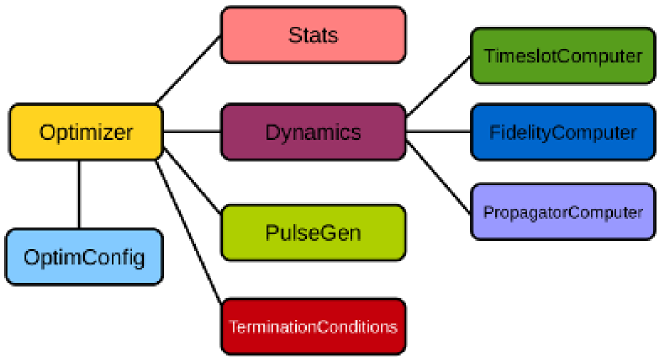


Fig. 2: Qtrl code object model.

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.

3.6 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.

CHAPTER

FOUR

CONTRIBUTING TO THE SOURCE CODE

4.1 Build up an development environment

Please follow the instruction on the QuTiP contribution guide to build a conda environment.

You don't need to build `qutip` in the editable mode unless you also want to contribute to *qutip*. Instead, you need to install `qutip-qtrl` by downloading the source code and running

```
pip install -e .
```

4.2 Docstrings for the code

Each class and function should be accompanied with a docstring explaining the functionality, including input parameters and returned values. The docstring should follow NumPy Style Python Docstrings.

4.3 Checking Code Style and Format

In order to check if your code in `some_file.py` follows PEP8 style guidelines, Black has to be installed.

```
pip install black
```

In the directory that contains `some_file.py`, use

```
black some_file.py --check
black some_file.py --diff --color
black some_file.py
```

Using `--check` will show if any of the file will be reformatted or not.

- Code 0 means nothing will be reformatted.
- Code 1 means one or more files could be reformatted. More than one files could be reformatted if `black some_directory --check` is used.

Using `--diff --color` will show a difference of the changes that will be made by `Black`. If you would prefer these changes to be made, use the last line of above code block.

4.4 Checking tests locally

You can run tests and generate code coverage report locally. First make sure required packages have been installed.

```
pip install pytest pytest-cov
```

`pytest` is used to test files containing tests. If you would like to test all the files contained in a directory then specify the path to this directory. In order to run tests in `test_something.py` then specify the exact path to this file for `pytest` or navigate to the file before running the tests.

```
pytest path_to_some_directory
pytest /path_to_test_something/test_something.py
~/path_to_test_something$ pytest test_something.py
```

A code coverage report in `html` format can be generated locally for `qutip-qtrl` using the code line given below. By default the coverage report is generated in a temporary directory `htmlcov`. The report can be output in other formats besides `html`.

```
pytest --cov-report html --cov=qutip_qtrl tests/
```

If you would prefer to check the code coverage of one specific file, specify the location of this file. Same as above the report can be accessed in `htmlcov`.

```
pytest --cov-report html --cov=qutip_qtrl tests/test_something.py
```

CHAPTER

FIVE

CONTRIBUTING TO THE DOCUMENTATION

The user guide provides an overview of the package's functionality. The guide is composed of individual reStructuredText (**.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, apidoc, sphinx gallery and plot. Additional extensions can be configured in the conf.py file.

5.1 Building the documentation

To build and test the documentation, the packages listed in `doc/requirements.txt` are required. You can install them using

```
pip install -r doc/requirements.txt
```

Under the `doc` directory, use

```
make html
```

to build the documentation in html format. The build is saved under the directory `doc/_build/html`

Use the command

```
make doctest
```

to run a test for the documentation.

5.2 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.

5.2.1 Doctest

The doctest directive enables tests on interactive code examples. The simplest way to do this is by specifying a prompt along with it's 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 it's 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.

```
a = 2
print(a)
```

Output:

```
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**.

5.2.2 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:

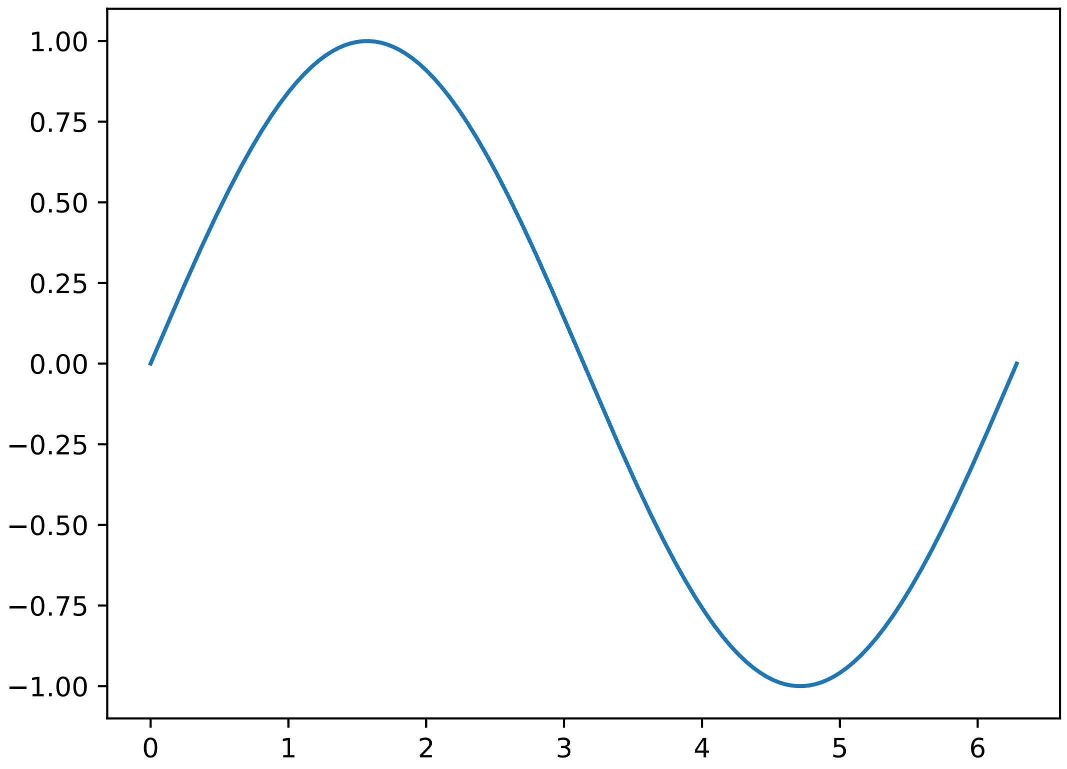
```
First we specify some data:

.. plot::

  >>> 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:

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.


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
1
2
3
4
5
6

- 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.

5.2.3 API documentation

If you are adding a new function or class in one of the existing modules, you only need to add it to the corresponding file in `doc/apidoc/`. If you are building a new module, first add the module to `doc/apidoc.rst`. Then add all the public classes and functions into `doc/apidoc/`. You may need to first use `make clean` to clean the build history before rebuilding the documentation.

If the new module contains many new functions/classes, you could turn on the `autosummary_generate = True` in `conf.py`. This will automatically generate all the files in `doc/apidoc/`, however, this will overwrite all the hand-optimized API documentation. Please revert other changes and keep only the file that you need.

CHAPTER

SIX

CHANGELOG

6.1 Version 0.1.1 (February 12, 2024)

This is a patch release of qutip-qtrl that provides updates to support the QuTiP 5.0.0a2 release.

6.1.1 Bug Fixes

- Fixed progress bar initialization and usage (#13, #15, Patrick Hopf).
- Replaced the logging utilities that were removed from QuTiP v5 with a vendored copy in this package (#9, #10, Èric Giguere, Boxi Li).
- Applied black to setup.py and doc/conf.py scripts (#8, Simon Cross).

6.2 Version 0.1.0 (March 12, 2023)

This is the first release of qutip-qtrl, the quantum control package in QuTiP.

The qutip-qtrl package used to be a module `qutip.control` under QuTiP (Quantum Toolbox in Python). From QuTiP 5.0, the community has decided to decrease the size of the core QuTiP package by reducing the external dependencies, in order to simplify maintenance. Hence a few modules are separated from the core QuTiP and will become QuTiP family packages. They are still maintained by the QuTiP team but hosted under different repositories in the QuTiP organization.

The qutip-qtrl package, QuTiP quantum optimal control, aims at providing advanced tools for the optimal control of quantum devices.

6.2.1 Features

- The *cy_grape_unitary* optimizer now uses the QuTiP 5 data layer directly and no longer requires Cython. The *cy_grape* module remains available for compatibility, but also no longer uses Cython.

6.2.2 Bug Fixes

- The *grape_unitary_adaptive* optimizer previously overwrote the target unitary with the identity. It no longer does this.

CHAPTER

SEVEN

QUTIP_QTRL PACKAGE

7.1 High-level interfaces

High-level interfaces to the optimal control features.

qutip_qtrl.pulseoptim	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.

7.1.1 qutip_qtrl.pulseoptim

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).

Functions

`create_pulse_optimizer`(drift, ctrls, ...[, ...])	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
`opt_pulse_crab`(drift, ctrls, initial, target)	Optimise a control pulse to minimise the fidelity error.
`opt_pulse_crab_unitary`(H_d, H_c, U_0, U_targ)	Optimise a control pulse to minimise the fidelity error, assuming that the dynamics of the system are generated by unitary operators.
`optimize_pulse`(drift, ctrls, initial, target)	Optimise a control pulse to minimise the fidelity error.
`optimize_pulse_unitary`(H_d, H_c, U_0, U_targ)	Optimise a control pulse to minimise the fidelity error, assuming that the dynamics of the system are generated by unitary operators.

`qutip_qtrl.pulseoptim.`**`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_qtrl.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 Dynam-

ics object, e.g. optimizer.dynamics.fid_computer The optimisation can be run through the optimizer.run_optimization

`qutip_qtrl.pulseoptim.`**`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_qtrl.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

`qutip_qtrl.pulseoptim.`**`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_qtrl.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.

`qutip_qtrl.pulseoptim.`**`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_qtrl.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.

`qutip_qtrl.pulseoptim.`**`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_qtrl.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.

7.2 Utilities

Optimal control utility functions.

qutip_qtrl.dump	Classes that enable the storing of historical objects created during the pulse optimisation.
qutip_qtrl.errors	Exception classes for the Quantum Control library
qutip_qtrl.loadparams	Loads parameters for config, termconds, dynamics and Optimiser objects from a parameter (ini) file with appropriate sections and options, these being Sections: optimconfig, termconds, dynamics, optimizer The options are assumed to be properties for these classes Note that new attributes will be created, even if they are not usually defined for that object
qutip_qtrl.io	
qutip_qtrl.stats	Statistics for the optimisation Note that some of the stats here are redundant copies from the optimiser used here for calculations

7.2.1 qutip_qtrl.dump

Classes that enable the storing of historical objects created during the pulse optimisation. These are intented for debugging. See the optimizer and dynamics objects for instrutcions on how to enable data dumping.

Classes

Dump()	A container for dump items.
DumpItem()	An item in a dump list
DumpSummaryItem()	A summary of the most recent iteration.
DynamicsDump(dynamics[, level])	A container for dumps of dynamics data.
EvoCompDumpItem(dump)	A copy of all objects generated to calculate one time evolution.
OptimDump(optim[, level])	A container for dumps of optimisation data generated during the pulse optimisation.

`class qutip_qtrl.dump.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 qutip_qtrl.dump.DumpItem`
An item in a dump list

`class qutip_qtrl.dump.DumpSummaryItem`
A summary of the most recent iteration. Abstract class only.

Attributes

idx
[int] Index in the summary list in which this is stored

`class qutip_qtrl.dump.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 qutip_qtrl.dump.**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 qutip_qtrl.dump.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.

7.2.2 qutip_qtrl.errors

Exception classes for the Quantum Control library

Exceptions

`Error`	Base class for all qutip control exceptions
`FunctionalError`(msg)	A function behaved in an unexpected way .
`GoalAchievedTerminate`(fid_err)	Exception raised to terminate execution when the goal has been reached during the optimisation algorithm
`GradMinReachedTerminate`(gradient)	Exception raised to terminate execution when the minimum gradient normal has been reached during the optimisation algorithm
`MaxFidFuncCallTerminate`()	Exception raised to terminate execution when the number of calls to the fidelity error function has exceeded the maximum
`MaxWallTimeTerminate`()	Exception raised to terminate execution when the optimisation time has exceeded the maximum set in the config
`OptimizationTerminate`	Superclass for all early terminations from the optimisation algorithm
`UsageError`(msg)	A function has been used incorrectly.

exception `qutip_qtrl.errors.`**`Error`**

Base class for all qutip control exceptions

exception `qutip_qtrl.errors.`**`FunctionalError`**(*msg*)

A function behaved in an unexpected way .. attribute:: funcname

function name where error occurred

`msg`

Explanation

exception `qutip_qtrl.errors.`**`GoalAchievedTerminate`**(*fid_err*)

Exception raised to terminate execution when the goal has been reached during the optimisation algorithm

exception `qutip_qtrl.errors.`**`GradMinReachedTerminate`**(*gradient*)

Exception raised to terminate execution when the minimum gradient normal has been reached during the optimisation algorithm

exception `qutip_qtrl.errors.`**`MaxFidFuncCallTerminate`**

Exception raised to terminate execution when the number of calls to the fidelity error function has exceeded the maximum

exception `qutip_qtrl.errors.`**`MaxWallTimeTerminate`**

Exception raised to terminate execution when the optimisation time has exceeded the maximum set in the config

exception `qutip_qtrl.errors.`**`OptimizationTerminate`**

Superclass for all early terminations from the optimisation algorithm

exception `qutip_qtrl.errors.`**`UsageError`**(*msg*)

A function has been used incorrectly. Most likely when a base class was used when a sub class should have been. .. attribute:: funcname function name where error occurred

`msg`

Explanation

7.2.3 qutip_qtrl.loadparams

Loads parameters for config, termconds, dynamics and Optimiser objects from a parameter (ini) file with appropriate sections and options, these being Sections: optimconfig, termconds, dynamics, optimizer The options are assumed to be properties for these classes Note that new attributes will be created, even if they are not usually defined for that object

Functions

`load_parameters`(file_name[, config, ...])	Import parameters for the optimisation objects Will throw a ValueError if file_name does not exist
`set_param`(parser, section, option, obj, ...)	Set the object attribute value based on the option value from the config file.

`qutip_qtrl.loadparams.`**`load_parameters`**(*file_name*, *config=None*, *term_conds=None*, *dynamics=None*, *optim=None*, *pulsegen=None*, *obj=None*, *section=None*)

Import parameters for the optimisation objects Will throw a ValueError if file_name does not exist

`qutip_qtrl.loadparams.`**`set_param`**(*parser*, *section*, *option*, *obj*, *attrib_name*)

Set the object attribute value based on the option value from the config file. If the attribute exists already, then its datatype is used to call the appropriate parser.get method Otherwise the parameter is assumed to be a string

7.2.4 qutip_qtrl.io

Functions

`create_dir`(dir_name[, desc])	Checks if the given directory exists, if not it is created

`qutip_qtrl.io.`**`create_dir`**(*dir_name*, *desc='output'*)

Checks if the given directory exists, if not it is created

Returns

dir_ok
[boolean] True if directory exists (previously or created) False if failed to create the directory

dir_name
[string] Path to the directory, which may be been made absolute

msg
[string] Error msg if directory creation failed

7.2.5 qutip_qtrl.stats

Statistics for the optimisation Note that some of the stats here are redundant copies from the optimiser used here for calculations

Classes

Stats()	Base class for all optimisation statistics Used for configurations where all timeslots are updated each iteration e.g.
StatsDynTsUpdate()	Optimisation stats class for configurations where all timeslots are not necessarily updated at each iteration.

class `qutip_qtrl.stats.`**`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 qutip_qtrl.stats.StatsDynTsUpdate`

Optimisation stats class for configurations where all timeslots are not necessarily updated at each iteration. In this case it may be interesting to know how many Hamiltions etc are computed each ctrl amplitude update

Attributes

num_dyn_gen_computes
[integer] Total number of dynamics generator (Hamiltonian) computations, that is combining drift and control dynamics to calculate the combined dynamics generator for the timeslot

mean_num_dyn_gen_computes_per_update
[float] # Mean average number of dynamics generator computations per update

mean_wall_time_dyn_gen_compute
[float] # Mean average time to compute a timeslot dynamics generator

num_prop_computes
[integer] Total number of propagator (and propagator gradient for exact gradient types) computations

mean_num_prop_computes_per_update
[float] Mean average number of propagator computations per update

mean_wall_time_prop_compute
[float] Mean average time to compute a propagator (and its gradient)

num_fwd_prop_step_computes
[integer] Total number of steps (matrix product) computing forward propagation

mean_num_fwd_prop_step_computes_per_update
[float] Mean average number of steps computing forward propagation

mean_wall_time_fwd_prop_compute
[float] Mean average time to compute forward propagation

num_onwd_prop_step_computes
[integer] Total number of steps (matrix product) computing onward propagation

mean_num_onwd_prop_step_computes_per_update
[float] Mean average number of steps computing onward propagation

mean_wall_time_onwd_prop_compute
Mean average time to compute onward propagation

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

7.3 Low-level interfaces

Internal interfaces to the optimal control features.

`qutip_qtrl.dynamics`	Classes that define the dynamics of the (quantum) system and target evolution to be optimised.
`qutip_qtrl.fidcomp`	Fidelity Computer
`qutip_qtrl.optimizer`	Classes here are expected to implement a run_optimization function that will use some method for optimising the control pulse, as defined by the control amplitudes.
`qutip_qtrl.optimconfig`	Configuration parameters for control pulse optimisation
`qutip_qtrl.optimresult`	Class containing the results of the pulse optimisation
`qutip_qtrl.propcomp`	Propagator Computer Classes used to calculate the propagators, and also the propagator gradient when exact gradient methods are used
`qutip_qtrl.pulsegen`	Pulse generator - Generate pulses for the timeslots Each class defines a gen_pulse function that produces a float array of size num_tslots.
`qutip_qtrl.symplectic`	Utility functions for symplectic matrices
`qutip_qtrl.termcond`	Classes containing termination conditions for the control pulse optimisation i.e. attributes that will be checked during the optimisation, that will determine if the algorithm has completed its task / exceeded limits.
`qutip_qtrl.tslotcomp`	Timeslot Computer These classes determine which dynamics generators, propagators and evolutions are recalculated when there is a control amplitude update.

7.3.1 qutip_qtrl.dynamics

Classes that define the dynamics of the (quantum) system and target evolution to be optimised. The contols are also defined here, i.e. the dynamics generators (Hamiltonians, Limbladians etc). The dynamics for the time slices are calculated here, along with the evolution as determined by the control amplitudes.

See the subclass descriptions and choose the appropriate class for the application. The choice depends on the type of matrix used to define the dynamics.

These class implement functions for getting the dynamics generators for the combined (drift + ctrls) dynamics with the approriate operator applied

Note the methods in these classes were inspired by: DYNAMO - Dynamic Framework for Quantum Optimal Control See Machnes et.al., arXiv.1011.4874

Classes

`Dynamics`(optimconfig[, params])	This is a base class only.
`DynamicsGenMat`(optimconfig[, params])	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.
`DynamicsSymplectic`(optimconfig[, params])	Symplectic systems This is the subclass to use for systems where the dynamics is described by symplectic matrices, e.g.
`DynamicsUnitary`(optimconfig[, params])	This is the subclass to use for systems with dynamics described by unitary matrices.

class `qutip_qtrl.dynamics.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_qtrl.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 qutip_qtrl.dynamics.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 qutip_qtrl.dynamics.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 qutip_qtrl.dynamics.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

7.3.2 qutip_qtrl.fidcomp

Fidelity Computer

These classes calculate the fidelity error - function to be minimised and fidelity error gradient, which is used to direct the optimisation

They may calculate the fidelity as an intermediary step, as in some case e.g. unitary dynamics, this is more efficient

The idea is that different methods for computing the fidelity can be tried and compared using simple configuration switches.

Note the methods in these classes were inspired by: DYNAMO - Dynamic Framework for Quantum Optimal Control See Machnes et.al., arXiv.1011.4874 The unitary dynamics fidelity is taken directly frm DYNAMO The other fidelity measures are extensions, and the sources are given in the class descriptions.

Classes

`FidCompTraceDiff`(dynamics[, params])	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.
`FidCompTraceDiffApprox`(dynamics[, params])	As FidCompTraceDiff, except uses the finite difference method to compute approximate gradients
`FidCompUnitary`(dynamics[, params])	Computes fidelity error and gradient assuming unitary dynamics, e.g.
`FidelityComputer`(dynamics[, params])	Base class for all Fidelity Computers.

class `qutip_qtrl.fidcomp.`**`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 qutip_qtrl.fidcomp.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 qutip_qtrl.fidcomp.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_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 qutip_qtrl.fidcomp.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_qtrl.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

7.3.3 qutip_qtrl.optimizer

Classes here are expected to implement a run_optimization function that will use some method for optimising the control pulse, as defined by the control amplitudes. The system that the pulse acts upon are defined by the Dynamics object that must be passed in the instantiation.

The methods are typically N dimensional function optimisers that find the minima of a fidelity error function. Note the number of variables for the fidelity function is the number of control timeslots, i.e. n_ctrls x Ntimeslots The methods will call functions on the Dynamics.fid_computer object, one or many times per interation, to get the fidelity error and gradient wrt to the amplitudes. The optimisation will stop when one of the termination conditions are met, for example: the fidelity aim has be reached, a local minima has been found, the maximum time allowed has been exceeded

These function optimisation methods are so far from SciPy.optimize The two methods implemented are:

BFGS - Broyden–Fletcher–Goldfarb–Shanno algorithm

> This a quasi second order Newton method. It uses successive calls to the gradient function to make an estimation of the curvature (Hessian) and hence direct its search for the function minima The SciPy implementation is pure Python and hance is execution speed is not high use subclass: OptimizerBFGS

L-BFGS-B - Bounded, limited memory BFGS

> This a version of the BFGS method where the Hessian approximation is only based on a set of the most recent gradient calls. It generally performs better where the are a large number of variables The SciPy implementation of L-BFGS-B is wrapper around a well established and actively maintained implementation in Fortran Its is therefore very fast. # See SciPy documentation for credit and details on the # scipy.optimize.fmin_l_bfgs_b function use subclass: OptimizerLBFGSB

The baseclass Optimizer implements the function wrappers to the fidelity error, gradient, and iteration callback functions. These are called from the within the SciPy optimisation functions. The subclasses implement the algorithm specific pulse optimisation function.

Classes

OptimIterSummary()	A summary of the most recent iteration of the pulse optimisation
Optimizer(config, dyn[, params])	Base class for all control pulse optimisers.
OptimizerBFGS(config, dyn[, params])	Implements the run_optimization method using the BFGS algorithm
OptimizerCrab(config, dyn[, params])	Optimises the pulse using the CRAB algorithm [Caneva].
OptimizerCrabFmin(config, dyn[, params])	Optimises the pulse using the CRAB algorithm [Doria], [Caneva].
OptimizerLBFGSB(config, dyn[, params])	Implements the run_optimization method using the L-BFGS-B algorithm

`class qutip_qtrl.optimizer.`**`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 qutip_qtrl.optimizer.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_qtrl.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_qtrl.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 qutip_qtrl.optimizer.`**`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 qutip_qtrl.optimizer.`**`OptimizerCrab`**`(`*config*, *dyn*, *params=None*`)`

Optimises the pulse using the CRAB algorithm [Caneva]. 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.

References

[Caneva]

`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 qutip_qtrl.optimizer.`**`OptimizerCrabFmin`**`(`*config*, *dyn*, *params=None*`)`

Optimises the pulse using the CRAB algorithm [Doria], [Caneva]. 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

[Doria], [Caneva]

`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 qutip_qtrl.optimizer.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

7.3.4 qutip_qtrl.optimconfig

Configuration parameters for control pulse optimisation

Classes

OptimConfig()	Configuration parameters for control pulse optimisation

class `qutip_qtrl.optimconfig.`**`OptimConfig`**

Configuration parameters for control pulse optimisation

Attributes

log_level
[integer] level of messaging output from the logger. Options are attributes of qutip_qtrl.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

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)

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)

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)

`check_create_output_dir`(*output_dir*, *desc='output'*)

Checks if the given directory exists, if not it is created.

Returns

dir_ok
[boolean] True if directory exists (previously or created) False if failed to create the directory

output_dir
[string] Path to the directory, which may be been made absolute

msg
[string] Error msg if directory creation failed

7.3.5 qutip_qtrl.optimresult

Class containing the results of the pulse optimisation

Classes

OptimResult()	Attributes give the result of the pulse optimisation attempt

`class qutip_qtrl.optimresult.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

7.3.6 qutip_qtrl.propcomp

Propagator Computer Classes used to calculate the propagators, and also the propagator gradient when exact gradient methods are used

Note the methods in the _Diag class was inspired by: DYNAMO - Dynamic Framework for Quantum Optimal Control See Machnes et.al., arXiv.1011.4874

Classes

PropCompApproxGrad(dynamics[, params])	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.
PropCompAugMat(dynamics[, params])	Augmented Matrix (deprecated - see _Frechet)
PropCompDiag(dynamics[, params])	Coumputes the propagator exponentiation using diagonalisation of of the dynamics generator
PropCompFrechet(dynamics[, params])	Frechet method for calculating the propagator: exponentiating the combined dynamics generator and the propagator gradient.
PropagatorComputer(dynamics[, params])	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.

`class qutip_qtrl.propcomp.`**`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 qutip_qtrl.propcomp.`**`PropCompAugMat`**(*dynamics*, *params=None*)

Augmented Matrix (deprecated - see _Frechet)

It should work for all systems, e.g. open, symplectic There will be other PropagatorComputer subclasses that are more efficient The _Frechet class should provide exactly the same functionality more efficiently.

Note the propagator gradient calculation using the augmented matrix 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

reset()

reset any configuration data

class qutip_qtrl.propcomp.**PropCompDiag**(*dynamics*, *params=None*)

Coumputes the propagator exponentiation using diagonalisation of of the dynamics generator

reset()

reset any configuration data

class qutip_qtrl.propcomp.**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 qutip_qtrl.propcomp.**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

7.3.7 qutip_qtrl.pulsegen

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

Functions

`create_pulse_gen`([pulse_type, dyn, pulse_params])	Create and return a pulse generator object matching the given type.

Classes

`PulseGen`([dyn, params])	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.
`PulseGenCrab`([dyn, num_coeffs, params])	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
`PulseGenCrabFourier`([dyn, num_coeffs, ...])	Generates a pulse using the Fourier basis functions, i.e. sin and cos.
`PulseGenGaussian`([dyn, params])	Generates pulses with a Gaussian profile
`PulseGenGaussianEdge`([dyn, params])	Generate pulses with inverted Gaussian ramping in and out It's intended use for a ramping modulation, which is often required in experimental setups.
`PulseGenLinear`([dyn, params])	Generates linear pulses
`PulseGenPeriodic`([dyn, params])	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
`PulseGenRandom`([dyn, params])	Generates random pulses as simply random values for each timeslot
`PulseGenRndFourier`([dyn, params])	Generates pulses by summing sine waves as a Fourier series with random coefficients
`PulseGenRndWalk1`([dyn, params])	Generates pulses by using a random walk algorithm
`PulseGenRndWalk2`([dyn, params])	Generates pulses by using a random walk algorithm Note this is best used with bounds as the walks tend to wander far
`PulseGenRndWaves`([dyn, params])	Generates pulses by summing sine waves with random frequencies amplitudes and phase offset
`PulseGenSaw`([dyn, params])	Generates saw tooth wave pulses
`PulseGenSine`([dyn, params])	Generates sine wave pulses
`PulseGenSquare`([dyn, params])	Generates square wave pulses
`PulseGenTriangle`([dyn, params])	Generates triangular wave pulses
`PulseGenZero`([dyn, params])	Generates a flat pulse

class `qutip_qtrl.pulsegen.`**`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_qtrl.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 qutip_qtrl.pulsegen.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 :rtype: list (or 1d array) of floats

`init_coeffs`(*num_coeffs=None*, *init_coeffs=None*)

Generate or set 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_coeffs
[float array[num_coeffs * num_basis_funcs]] Typically this will be the initial basis coefficients. If set to *None* (the default), the initial coefficients will be automatically generated.

`init_pulse`(*num_coeffs=None*, *init_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 qutip_qtrl.pulsegen.PulseGenCrabFourier`(*dyn=None*, *num_coeffs=None*, *params=None*, *fix_freqs=True*)

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

fix_freqs
[bool] If True (default) then the frequencies of the basis functions are fixed and the number of basis functions is set to 2 (sin and cos). If False then the frequencies are also optimised, adding an additional parameter for each pair of basis functions.

`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*, *init_coeffs=None*`)`
Set the initial freq and coefficient values

`reset()`
reset attributes to default values

`class qutip_qtrl.pulsegen.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 qutip_qtrl.pulsegen.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 qutip_qtrl.pulsegen.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 qutip_qtrl.pulsegen.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 qutip_qtrl.pulsegen.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 qutip_qtrl.pulsegen.PulseGenRndFourier`(*dyn=None*, *params=None*)

Generates pulses by summing sine waves as a Fourier series with random coefficients

Attributes

scaling
[float] The pulses should fit approximately within -/+scaling (before the offset is applied) as it is used to set a maximum for each component wave Use bounds to be sure (copied from Dynamics.initial_ctrl_scaling if given)

min_wavelen
[float] Minimum wavelength of any component wave Set by default to 1/10th of the pulse time

`gen_pulse`(*min_wavelen=None*)

Generate a random pulse based on a Fourier series with a minimum wavelength `reset()`

reset attributes to default values

`class qutip_qtrl.pulsegen.PulseGenRndWalk1(`*dyn=None*, *params=None*`)`

Generates pulses by using a random walk algorithm

Attributes

scaling
[float] Used as the range for the starting amplitude Note must used bounds if values must be restricted. Also scales the max_d_amp value (copied from Dynamics.initial_ctrl_scaling if given)

max_d_amp
[float] Maximum amount amplitude will change between timeslots Note this is also factored by the scaling attribute

`gen_pulse(`*max_d_amp=None*`)`

Generate a pulse by changing the amplitude a random amount between -max_d_amp and +max_d_amp at each timeslot. The walk will start at a random amplitude between -/+scaling.

`reset()`

reset attributes to default values

`class qutip_qtrl.pulsegen.PulseGenRndWalk2(`*dyn=None*, *params=None*`)`

Generates pulses by using a random walk algorithm Note this is best used with bounds as the walks tend to wander far

Attributes

scaling
[float] Used as the range for the starting amplitude Note must used bounds if values must be restricted. Also scales the max_d2_amp value (copied from Dynamics.initial_ctrl_scaling if given)

max_d2_amp
[float] Maximum amount amplitude gradient will change between timeslots Note this is also factored by the scaling attribute

`gen_pulse(`*init_grad_range=None*, *max_d2_amp=None*`)`

Generate a pulse by changing the amplitude gradient a random amount between -max_d2_amp and +max_d2_amp at each timeslot. The walk will start at a random amplitude between -/+scaling. The gradient will start at 0

`reset()`

reset attributes to default values

`class qutip_qtrl.pulsegen.PulseGenRndWaves(`*dyn=None*, *params=None*`)`

Generates pulses by summing sine waves with random frequencies amplitudes and phase offset

Attributes

scaling
[float] The pulses should fit approximately within -/+scaling (before the offset is applied) as it is used to set a maximum for each component wave Use bounds to be sure (copied from Dynamics.initial_ctrl_scaling if given)

num_comp_waves
[integer] Number of component waves. That is the number of waves that are summed to make the pulse signal Set to 20 by default.

min_wavelen
[float] Minimum wavelength of any component wave Set by default to 1/10th of the pulse time

max_wavelen
[float] Maximum wavelength of any component wave Set by default to twice the pulse time

`gen_pulse`(*num_comp_waves=None*, *min_wavelen=None*, *max_wavelen=None*)
Generate a random pulse by summing sine waves with random freq, amplitude and phase offset

`reset`()
reset attributes to default values

`class qutip_qtrl.pulsegen.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 qutip_qtrl.pulsegen.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 qutip_qtrl.pulsegen.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 qutip_qtrl.pulsegen.PulseGenTriangle`(*dyn=None*, *params=None*)
Generates triangular wave pulses

`gen_pulse`(*num_waves=None*, *wavelen=None*, *freq=None*, *start_phase=None*)
Generate a triangular 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 qutip_qtrl.pulsegen.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

`qutip_qtrl.pulsegen.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.

7.3.8 qutip_qtrl.symplectic

Utility functions for symplectic matrices

Functions

calc_omega(n)	Calculate the 2n x 2n Omega matrix Used as dynamics generator phase to calculate symplectic propagators

`qutip_qtrl.symplectic.`**`calc_omega`**`(n)`

Calculate the 2n x 2n Omega matrix Used as dynamics generator phase to calculate symplectic propagators

Parameters

n
[scalar(int)] number of modes in oscillator system

Returns

array(float)
Symplectic phase Omega

7.3.9 qutip_qtrl.termcond

Classes containing termination conditions for the control pulse optimisation i.e. attributes that will be checked during the optimisation, that will determine if the algorithm has completed its task / exceeded limits

Classes

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

`class` `qutip_qtrl.termcond.`**`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

7.3.10 qutip_qtrl.tslotcomp

Timeslot Computer These classes determine which dynamics generators, propagators and evolutions are recalculated when there is a control amplitude update. The timeslot computer processes the lists held by the dynamics object

The default (UpdateAll) updates all of these each amp update, on the assumption that all amplitudes are changed each iteration. This is typical when using optimisation methods like BFGS in the GRAPE algorithm

The alternative (DynUpdate) assumes that only a subset of amplitudes are updated each iteration and attempts to minimise the number of expensive calculations accordingly. This would be the appropriate class for Krotov type methods. Note that the Stats_DynTsUpdate class must be used for stats in conjunction with this class. NOTE: AJGP 2011-10-2014: This _DynUpdate class currently has some bug, no pressing need to fix it presently

If all amplitudes change at each update, then the behavior of the classes is equivalent. _UpdateAll is easier to understand and potentially slightly faster in this situation.

Note the methods in the _DynUpdate class were inspired by: DYNAMO - Dynamic Framework for Quantum Optimal Control See Machnes et.al., arXiv.1011.4874

Classes

EvoCompSummary()	A summary of the most recent time evolution computation Used in stats calculations and for data dumping
TSlotCompDynUpdate(dynamics[, params])	Timeslot Computer - Dynamic Update
TSlotCompUpdateAll(dynamics[, params])	Timeslot Computer - Update All Updates all dynamics generators, propagators and evolutions when ctrl amplitudes are updated
TimeslotComputer(dynamics[, params])	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

`class qutip_qtrl.tslotcomp.`**`EvoCompSummary`**

A summary of the most recent time evolution computation Used in stats calculations and for data dumping

Attributes

evo_dump_idx
[int] Index of the linked `dump.EvoCompDumpItem` None if no linked item

iter_num
[int] Iteration number of the pulse optimisation None if evolution compute outside of a pulse optimisation

fid_func_call_num
[int] Fidelity function call number of the pulse optimisation None if evolution compute outside of a pulse optimisation

grad_func_call_num
[int] Gradient function call number of the pulse optimisation None if evolution compute outside of a pulse optimisation

num_amps_changed
[int] Number of control timeslot amplitudes changed since previous evolution calculation

num_timeslots_changed
[int] Number of timeslots in which any amplitudes changed since previous evolution calculation

wall_time_dyn_gen_compute
[float] Time spent computing dynamics generators (in seconds of elapsed time)

wall_time_prop_compute
[float] Time spent computing propagators (including and propagator gradients) (in seconds of elapsed time)

wall_time_fwd_prop_compute
[float] Time spent computing the forward evolution of the system see `dynamics.fwd_evo` (in seconds of elapsed time)

wall_time_onwd_prop_compute
[float] Time spent computing the 'backward' evolution of the system see `dynamics.onwd_evo` and `dynamics.onto_evo` (in seconds of elapsed time)

`class qutip_qtrl.tslotcomp.`**`TSlotCompDynUpdate`**`(`*dynamics*, *params=None*`)`

Timeslot Computer - Dynamic Update

Warning: CURRENTLY HAS ISSUES (AJGP 2014-10-02) and is therefore not being maintained i.e. changes made to _UpdateAll are not being implemented here

Updates only the dynamics generators, propagators and evolutions as required when a subset of the ctrl amplitudes are updated. Will update all if all amps have changed.

`compare_amps``(`*new_amps*`)`

Determine which timeslots will have changed Hamiltonians i.e. any where control amplitudes have changed for that slot and mark (using masks) them and corresponding exponentiations and time evo operators for update Returns: True if amplitudes are the same, False if they have changed

`flag_all_calc_now``()`

Flags all Hamiltonians, propagators and propagations to be calculated now

`get_timeslot_for_fidelity_calc``()`

Returns the timeslot index that will be used calculate current fidelity value. Attempts to find a timeslot where the least number of propagator calculations will be required. Flags the associated evolution operators for calculation now `init_comp()`

Initialise the flags

`recompute_evolution()`

Recalculates the evo_init2t (forward) and evo_t2targ (onward) time evolution operators DynGen (Hamiltonians etc) and prop (propagator) are calculated as necessary

`class qutip_qtrl.tslotcomp.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 qutip_qtrl.tslotcomp.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_qtrl.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

CHAPTER

EIGHT

BIBLIOGRAPHY

CHAPTER NINE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Caneva] T. Caneva, T. Calarco, and S. Montangero. Chopped random-basis quantum optimization, Phys. Rev. A, 84:022326, 2011 (doi:10.1103/PhysRevA.84.022326)

[Doria] P. Doria, T. Calarco & S. Montangero. Phys. Rev. Lett. 106, 190501 (2011).

[Caneva] T. Caneva, T. Calarco, & S. Montangero. Phys. Rev. A 84, 022326 (2011).

[1] D. d'Alessandro. *Introduction to Quantum Control and Dynamics*. Chapman & Hall/CRC, 2008.

[2] 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.

[3] R. H. Byrd, J. Nocedal P. Lu, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.*, 16:1190, 1995. doi:10.1137/0916069.

[4] 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.

[5] S. Lloyd and S. Montangero. Information theoretical analysis of quantum optimal control. *Phys. Rev. Lett.*, 13:010502, 2014. doi:10.1103/PhysRevLett.113.010502.

[6] P. Doria, T. Calarco, and S. Montangero. Optimal control technique for many-body quantum dynamics. *Phys. Rev. Lett.*, 106:190501, 2011. doi:10.1103/PhysRevLett.106.190501.

[7] T. Caneva, T. Calarco, and S. Montangero. Chopped random-basis quantum optimization. *Phys. Rev. A*, 84:022326, 2011. doi:10.1103/PhysRevA.84.022326.

[8] 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.

[9] 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. URL: https://arxiv.org/abs/1011.4874.

PYTHON MODULE INDEX

INDEX