Krotov

Release 0.4.1

Michael Goerz *et. al.*

Oct 11, 2019

Contents:

1 Krotov Python Package

Python implementation of Krotov's method for quantum optimal control.

This implementation follows the original implementation in the QDYN Fortran library.

The *krotov* package is built on top of QuTiP.

Development happens on Github. You can read the full documentation at ReadTheDocs or download a PDF version.

If you use the *krotov* package in your research, please cite it.

1.1 Purpose

Optimal control is a cornerstone of quantum technology: relying not just on a passive understanding of quantum mechanics, but on the *active* utilization of the quantum properties of matter. Quantum optimal control asks how to manipulate the dynamics of a quantum system in some desired way. This is essential for the realization of quantum computers and related technologies such as quantum sensing.

Krotov's method and GRAPE are the two leading gradient-based optimization algorithms used in numerical quantum optimal control. Krotov's method distinguishes itself by guaranteeing monotonic convergence for near-continuous control fields. This makes is particularly useful for exploring the limits of controllability in a physical system. While GRAPE is found in various software packages, there has not been an open source implementation of Krotov's method to date. Our package provides that missing implementation.

The Krotov package targets both students wishing to enter the field of quantum control and researchers in the field. It was designed towards the following goals:

- Leverage the QuTiP library as a platform for numerically describing quantum systems.
- Provide a collection of examples inspired by recent publications in the Jupyter notebook format, allowing for interactive exploration of the method.
- Define a general interface for formulating *any* quantum control problem, which may extend to other optimization methods in the future.
- Serve as a reference implementation of Krotov's method, and as a foundation against which to test other implementations.

- Enable the more widespread use of Krotov's method, for example in the design of experiments.

1.2 Prerequisites

The Krotov package is available for Python versions >= 3.5. Its main dependency is QuTiP (apart from the core packages of the Python scientific ecosystem). Thus, you should consider QuTiP's installation instructions.

In any case, using some sort of virtual environment is strongly encouraged. Most packages in the Python scientific ecosystem are now available as wheels, making installation via pip easy. However, QuTiP currently does not provide wheels. Thus, on systems that do not have the necessary compilers installed (Windows, macOS), the conda package manager provides a good solution.

Assuming `conda` is installed (e.g. through Miniconda), the following commands set up a virtual (conda) environment into which the Krotov package can then be installed:

```
$ conda create -n qucontrolenv python=3.7
$ conda activate qucontrolenv
$ conda config --append channels conda-forge
$ conda install qutip
```

1.3 Installation

To install the latest released version of *krotov* into your current (conda) environment, run this command in your terminal:

```
$ python -m pip install krotov
```

This is the preferred method to install the *krotov* package, as it will always install the most recent stable release.

You may also do

```
$ python -m pip install krotov[dev,extras]
```

to install additional development dependencies, including packages required to run the example notebooks.

If you don't have pip installed, the Python installation guide, respectively the Python Packaging User Guide can guide you through the process.

To install the latest development version of *krotov* from Github:

```
$ python -m pip install git+https://github.com/qucontrol/krotov.git@master#egg=krotov
```

1.4 Usage

To use Krotov's method for quantum optimal control in a Python script or Jupyter notebook, start with:

```
import krotov
import qutip
```

Then,

1. define the necessary quantum operators and states using QuTiP.
2. create a list of objectives, as instances of *krotov.Objective*.
3. call *krotov.optimize_pulses* to perform an optimization of an arbitrary number of control fields over all the objectives.

See *Using Krotov with QuTiP* and *Examples* for details.

1.5 Citing the Krotov Package

Attention: Please cite the *krotov* package as

- M. H. Goerz *et al., Krotov: A Python implementation of Krotov's method for quantum optimal control*, arXiv:1902.11284 (2019)

You can also print this from `krotov.__citation__`:

```
>>> print(krotov.__citation__)
M. H. Goerz et al., Krotov: A Python implementation of Krotov's method for quantum␣
↪optimal control, arXiv:1902.11284 (2019)
```

The corresponding BibTeX entry is available in `krotov.__bibtex__`:

```
>>> print(krotov.__bibtex__)
@article{arxiv1902.11284,
    author = {Michael H. Goerz and Daniel Basilewitsch and Fernando Gago-Encinas and␣
↪Matthias G. Krauss and Karl P. Horn and Daniel M. Reich and Christiane P. Koch},
    title = {Krotov: A {Python} implementation of {Krotov's} method for quantum␣
↪optimal control},
    year = {2019},
    journal = {arXiv:1902.11284},
}
```

2 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

2.1 Code of Conduct

Everyone interacting in the `krotov` project's code base, issue tracker, and any communication channels is expected to follow the PyPA Code of Conduct.

2.2 Report Bugs

Report bugs at https://github.com/qucontrol/krotov/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

2.3 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/qucontrol/krotov/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

2.4 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in `docs/04_features.rst` and/or `HISTORY.rst`.
3. Check https://travis-ci.org/qucontrol/krotov/pull_requests and make sure that the tests pass for all supported Python versions.

2.5 Get Started!

Ready to contribute? Follow Aaron Meurer's Git Workflow Notes (with `qucontrol/krotov` instead of `sympy/sympy`)

In short, if you are not a member of the qucontrol organization,

1. Clone the repository from `git@github.com:qucontrol/krotov.git`
2. Fork the repo on GitHub to your personal account.
3. Add your fork as a remote.
4. Pull in the latest changes from the master branch.
5. Create a topic branch.
6. Make your changes and commit them (testing locally).
7. Push changes to the topic branch on *your* remote.
8. Make a pull request against the base master branch through the Github website of your fork.

The project uses tox for automated testing accross multiple versions of Python and for various development tasks such as linting and generating the documentation. See *Development Prerequisites* for details.

There is also a `Makefile` that wraps around tox, for convenience on Unix-based systems. In your checked-out clone, run

```
$ make help
```

to see the available make targets. If you cannot use `make`, but want to use `tox` directly (e.g., on Windows), run

```
$ tox -av
```

to see a list of tox environments and a description. For the initial configuration of tox environments, you may have to run

```
$ tox -e bootstrap
```

in order to set up the `tox.ini` configuration file.

If you are a member of the qucontrol organization, there is no need to fork `krotov` - you can directly pull and push to `git@github.com:qucontrol/krotov.git`.

2.6 Development Prerequisites

Contributing to the package's developments requires that you have Python 3.7 and tox installed. It is strongly recommended that you also have installations of all other supported Python versions. The recommended way to install multiple versions of Python at the same time is through Use pyenv (or pyenv-win on Windows).

Alternatively, you may install conda (via the Anaconda or Miniconda distributions, or also through pyenv). As `conda` can create environments with any version of Python (independent of which Python version `conda` was originally installed with), this alleviates the need for managing multiple versions. The advantage of using conda is that you may be able to avoid installing the compilers necessary for Python extension packages. The disadvantage is that environment creation is slower and the resulting environments are bigger, and that you may run into occasional binary incompatibilities between conda packages.

Warning: If you want to use *conda*, you must use the `tox-conda.ini` configuration file. That is, run all `make` comands as e.g. `make TOXINI=tox-conda.ini test` and `tox` commands as e.g. `tox -c tox-conda.ini -e py35-test,py36-test,py37-test`. Alternatively, make `tox-conda.ini` the default by copying it to `tox.ini`.

2.7 Branching Model

For developers with direct access to the repository, `krotov` uses a simple branching model where all developments happens directly on the `master` branch. Releases are tags on `master`. All commits on `master` *should* pass all tests and be well-documented. This is so that `git bisect` can be effective. For any non-trivial issue, it is recommended to create a topic branch, instead of working on `master`. There are no restrictions on commits on topic branches, they do not need to contain complete documentation, pass any tests, or even be able to run.

To create a topic-branch named `issue1`:

```
$ git branch issue1
$ git checkout issue1
```

You can then make commits, and push them to Github to trigger Continuous Integration testing:

```
$ git push -u origin issue1
```

Commit early and often! At the same time, try to keep your topic branch as clean and organized as possible. If you have not yet pushed your topic branch to the "origin" remote:

- Avoid having a series of meaningless granular commits like "start bugfix", "continue development", "add more work on bugfix", "fix typos", and so forth. Instead, use `git commit --amend` to add to your previous commit. This is the ideal way to "commit early and often". You do not have to wait until a commit is "perfect"; it is a good idea to make

hourly/daily "snapshots" of work in progress. Amending a commit also allows you to change the commit message of your last commit.

- You can combine multiple existing commits by "squashing" them. For example, use `git rebase -i HEAD~4` to combined the previous four commits into one. See the "Rewriting History" section of Pro Git book for details (if you feel this is too far outside of your git comfort zone, just skip it).
- If you work on a topic branch for a long time, and there is significant work on `master` in the meantime, periodically rebase your topic branch on the current master (`git rebase master`). Avoid merging `master` into your topic branch. See Merging vs. Rebasing.

If you have already pushed your topic branch to the remote origin, you have to be a bit more careful. If you are sure that you are the only one working on that topic branch, you can still follow the above guidelines, and force-push the issue branch (`git push --force`). This also applies if you are an external contributor preparing a pull request in your own clone of the project. If you are collaborating with others on the topic branch, coordinate with them whether they are OK with rewriting the history. If not, merge instead of rebasing. You must never rewrite history on the `master` branch (nor will you be able to, as the `master` branch is "protected" and can only be force-pushed to in coordination with the project maintainer). If something goes wrong with any advanced "history rewriting", there is always "git reflog" as a safety net – you will never lose work that was committed before.

When you are done with a topic branch (the issue has been fixed), finish up by merging the topic branch back into `master`:

```
$ git checkout master
$ git merge --no-ff issue1
```

The `--no-ff` option is critical, so that an explicit merge commit is created (especially if you rebased). Summarize the changes of the branch relative to `master` in the commit message.

Then, you can push master and delete the topic branch both locally and on Github:

```
$ git push origin master
$ git push --delete origin issue1
$ git branch -D issue1
```

2.8 Commit Message Guidelines

Write commit messages according to this template:

```
Short (50 chars or less) summary ("subject line")

More detailed explanatory text. Wrap it to 72 characters. The blank
line separating the summary from the body is critical (unless you omit
the body entirely).

Write your subject line in the imperative: "Fix bug" and not "Fixed
bug" or "Fixes bug." This convention matches up with commit messages
generated by commands like git merge and git revert. A properly formed
git commit subject line should always be able to complete the sentence
"If applied, this commit will <your subject line here>".

Further paragraphs come after blank lines.
```

(continues on next page)

(continued from previous page)

```
- Bullet points are okay, too.
- Typically a hyphen or asterisk is used for the bullet, followed by a
  single space. Use a hanging indent.

You should reference any issue that is being addressed in the commit, as
e.g. "#1" for issue #1. If the commit closes an issue, state this on the
last line of the message (see below). This will automatically close the
issue on Github as soon as the commit is pushed there.

Closes #1
```

See Closing issues using keywords for details on references to issues that Github will understand.

2.9 Testing

The Krotov package includes a full test-suite using pytest. We strive for a test coverage above 90%.

From a checkout of the `krotov` repository, you can use

```
$ make test
```

to run the entire test suite, or

```
$ tox -e py35-test,py36-test,py37-test
```

if `make` is not available.

The tests are organized in the `tests` subfolder. It includes python scripts whose name start with `test_`, which contain functions whose names also start with `test_`. Any such functions in any such files are picked up by pytest for testing. In addition, doctests from any docstring or any documentation file (`*.rst`) are picked up (by the pytest doctest plugin). Lastly, all *example notebooks* are validated as a test, through the nbval plugin.

2.10 Code Style

All code must be compatible with **PEP 8**. The line length limit is 79 characters, although exceptions are permissible if this improves readability significantly.

Beyond **PEP 8**, this project adopts the Black code style, with `--skip-string-normalization --line-length 79`. You can run `make black-check` or `tox -e run-blackcheck` to check adherence to the code style, and `make black` or `tox -e run-black` to apply it.

Imports within python modules must be sorted according to the isort configuration in `setup.cfg`. The command `make isort-check` or `tox -e run-isortcheck` checks whether all imports are sorted correctly, and `make isort` or `tox -e run-isort` modifies all Python modules in-place with the proper sorting.

The code style is enforced as part of the test suite, as well as through git pre-commit hooks that prevent committing code not does not meet the requirements. These hooks are managed through the pre-commit framework.

Warning: After cloning the `krotov` repository, you should run `make bootstrap`, `tox -e bootstrap`, or `python scripts/bootstrap.py` from within the project root folder. These set up `tox`, and the pre-commit hooks

You may use `make flake8-check` or `tox -e run-flake8` and `make pylint-check` or `tox -e run-pylint` for additional checks on the code with flake8 and pylint, but there is no strict requirement for a perfect score with either one of these linters. They only serve as a guideline for code that might be improved.

2.11 Write Documentation

The `krotov` package could always use more documentation, whether as part of the official docs, in docstrings, or even on the web in blog posts, articles, and such.

The package documentation is generated with Sphinx, the documentation (and docstrings) are formatted using the Restructured Text markup language (file extension `rst`). See also the Matplotlib Sphinx cheat sheet for some helpful tips.

Each function or class must have a docstring; this docstring must be written in the "Google Style" format (as implemented by Sphinx' napoleon extension). Docstrings and any other part of the documentation can include mathematical formulas in LaTeX syntax (using mathjax). In addition to Sphinx' normal syntax for inline math (`:math:`x``), you may also use easier-to-read dollar signs (`$x$`). The Krotov package defines some custom tex macros for quantum mechanics, which you are strongly encouraged to use. These include:

- `\bra`, e.g. `$\bra{\Psi}$` for $\langle\Psi|$ (or `\\Bra{}` for auto-resizing). Do not use `\langle`/`\rangle`/`\vert` manually!
- `\ket`, e.g. `$\ket{\Psi}$` for $|\Psi\rangle$ (or `\Ket{}` for auto-resizing).
- `\Braket`, e.g. `$\Braket{\Phi}{\Psi}$` for $\langle\Phi \mid \Psi\rangle$.
- `\Op` for quantum operators, e.g. `$\Op{H}$` for $\hat{H}$.
- `\Abs` for absolute values, e.g. `$\Abs{x}$` for $|x|$.
- `\AbsSq` for the absolute-square, e.g. `$\AbsSq{\Braket{\Phi}{\Psi}}$` for $|\langle\Phi \mid \Psi\rangle|^2$.
- `\avg` for the expectation values, e.g. `$\avg{\Op{H}}$` for $\langle\hat{H}\rangle$ (or `\Avg{}` for auto-resizing).
- `\Norm` for the norm, e.g. `$\Norm{\ket{\Psi}}$` for $\|\|\Psi\rangle\|$.
- `\identity` for the identity operator, $\mathbf{1}$.
- `\Liouville` for the Liouvillian symbol, $\mathcal{L}$.
- `\DynMap` for the symbolic dynamical map, $\mathcal{E}$.
- `\dd` for the differential, e.g. `$\int f(x) \dd x$` for $\int f(x)\,\mathrm{d}x$.
- Function names / mathematical operators `\tr`, `\diag`, `\abs`, `\pop`.
- Text labels `\aux`, `\opt`, `\tgt`, `\init`, `\lab`, `\rwa`.

Also see *Math in Example Notebooks*.

You may use the BibTeX plugin for citations.

At any point, from a checkout of the `krotov` repository, you may run

```
$ make docs
```

or

```
$ tox -e docs
```

to generate the documentation locally.

2.12 Contribute Examples

Examples should be contributed in the form of Jupyter notebooks.

Example notebooks are automatically rendered as part of the documentation (*Examples*), and they are also verified by the automated tests. For this to work properly, the following steps must be taken:

- Put all imports near the top of the notebook, with `# NBVAL_IGNORE_OUTPUT` as the first line. Use the watermark package to print out the versions of imported packages. For example:

```
# NBVAL_IGNORE_OUTPUT
%load_ext watermark
import qutip
import numpy as np
import scipy
import matplotlib
import matplotlib.pylab as plt
%watermark -v --iversions
```

- Put the notebook in the folder `docs/notebooks/`.
- Before committing, re-evaluate all example notebooks in a well-defined virtual environment by running

```
$ make notebooks
```

- Check that the examples can be verified across different Python version by running

```
$ make test
```

- You may also verify that the example is properly integrated in the documentation by running

```
$ make docs
```

2.12.1 Math in Example Notebooks

You may use the same tex macros described in the *Write Documentation* section. However, for the macros to work when viewing the notebook by itself, they must be redefined locally. To this end, add a markdown cell underneath the top cell that contains the imported packages (see above). The cell must contain the following:

```
$\newcommand{tr}[0]{\operatorname{tr}}
\newcommand{diag}[0]{\operatorname{diag}}
\newcommand{abs}[0]{\operatorname{abs}}
\newcommand{pop}[0]{\operatorname{pop}}
\newcommand{aux}[0]{\text{aux}}
\newcommand{opt}[0]{\text{opt}}
\newcommand{tgt}[0]{\text{tgt}}
\newcommand{init}[0]{\text{init}}
\newcommand{lab}[0]{\text{lab}}
\newcommand{rwa}[0]{\text{rwa}}
\newcommand{bra}[1]{\langle#1\vert}
\newcommand{ket}[1]{\vert#1\rangle}
\newcommand{Bra}[1]{\left\langle#1\right\vert}
\newcommand{Ket}[1]{\left\vert#1\right\rangle}
\newcommand{Braket}[2]{\left\langle #1\vphantom{#2} \mid #2\vphantom{#1}\right\rangle}
\newcommand{op}[1]{\hat{#1}}
\newcommand{Op}[1]{\hat{#1}}
\newcommand{dd}[0]{\,\text{d}}
\newcommand{Liouville}[0]{\mathcal{L}}
\newcommand{DynMap}[0]{\mathcal{E}}
\newcommand{identity}[0]{\mathbf{1}}
\newcommand{Norm}[1]{\lVert#1\rVert}
\newcommand{Abs}[1]{\left\vert#1\right\vert}
\newcommand{avg}[1]{\langle#1\rangle}
\newcommand{Avg}[1]{\left\langle#1\right\rangle}
\newcommand{AbsSq}[1]{\left\vert#1\right\vert^2}
\newcommand{Re}[0]{\operatorname{Re}}
\newcommand{Im}[0]{\operatorname{Im}}$
```

Upon executing the cell the definitions will be hidden, but the defined macros will be available in any cell in the rest of the notebook.

2.13 Versioning

Releases should follow Semantic Versioning, and version numbers published to PyPI must be compatible with **PEP 440**.

In short, versions number follow the pattern *major.minor.patch*, e.g. `0.1.0` for the first release, and `1.0.0` for the first *stable* release. If necessary, pre-release versions might be published as e.g:

```
1.0.0-dev1  # developer's preview 1 for release 1.0.0
1.0.0-rc1   # release candidate 1 for 1.0.0
```

Errors in the release metadata or documentation only may be fixed in a post-release, e.g.:

```
1.0.0.post1  # first post-release after 1.0.0
```

Post-releases should be used sparingly, but they are acceptable even though they are not supported by the Semantic Versioning specification.

The current version is available through the `__version__` attribute of the *krotov* package:

```
>>> import krotov
>>> krotov.__version__
```

Between releases, `__version__` on the master branch should either be the version number of the last release, with "+dev" appended (as a "local version identifier"), or the version number of the next planned release, with "-dev" appended ("pre-release identifier" with extra dash). The version string "1.0.0-dev1+dev" is a valid value after the "1.0.0-dev1" pre-release. The "+dev" suffix must never be included in a release to PyPI.

Note that twine applies normalization to the above recommended forms to make them strictly compatible with **PEP 440**, before uploading to PyPI. Users installing the package through pip may use the original version specification as well as the normalized one (or any other variation that normalizes to the same result).

When making a release via

```
$ make release
```

the above versioning conventions will be taken into account automatically.

Releases must be tagged in git, using the version string prefixed by "v", e.g. `v1.0.0-dev1` and `v1.0.0`. This makes them available at https://github.com/qucontrol/krotov/releases.

2.14 Developers' How-Tos

The following assumes your current working directory is a checkout of `krotov`, and that you have successfully run `make test` (which creates the tox environments that development relies on).

2.14.1 How to run a jupyter notebook server for working on the example notebooks

A notebook server that is isolated to the proper testing environment can be started via the Makefile:

```
$ make jupyter-notebook
```

This is equivalent to:

```
$ tox -e run-cmd -- jupyter notebook --config=/dev/null
```

You may run this with your own options, if you prefer. The `--config=/dev/null` guarantees that the notebook server is completely isolated. Otherwise, configuration files from your home directly (see Jupyter's Common Configuration system) may influence the server. Of course, if you know what you're doing, you may want this.

If you prefer, you may also use the newer jupyterlab:

```
$ make jupyter-lab
```

2.14.2 How to convert an example notebook to a script for easier debugging

Interactive debugging in notebooks is difficult. It becomes much easier if you convert the notebook to a script first. To convert a notebook to an (I)Python script and run it with auto-

matic debugging, execute e.g.:

```
$ tox -e run-cmd -- jupyter nbconvert --to=python --stdout docs/notebooks/01_example_
↪transmon_xgate.ipynb > debug.py
$ tox -e run-cmd -- ipython --pdb debug.py
```

You can then also set a manual breakpoint by inserting the following line anywhere in the code:

```
from IPython.terminal.debugger import set_trace; set_trace() # DEBUG
```

2.14.3 How to make `git diff` work for notebooks

Install nbdime and run `nbdime config-git --enable --global` to enable the git integration.

2.14.4 How to commit failing tests or example notebooks

The test-suite on the `master` branch should always pass without error. If you would like to commit any example notebooks or tests that currently fail, as a form of test-driven development, you have two options:

- Push onto a topic branch (which are allowed to have failing tests), see the *Branching Model*. The failing tests can then be fixed by adding commits to the same branch.
- Mark the test as failing. For normal tests, add a decorator:

  ```
  @pytest.mark.xfail
  ```

  See the pytest documentation on skip and xfail for details.

  For notebooks, the equivalent to the decorator is to add a comment to the first line of the failing cell, either:

  ```
  # NBVAL_RAISES_EXCEPTION
  ```

  (preferably), or:

  ```
  # NBVAL_SKIP
  ```

  (this may affect subsequent cells, as the marked cell is not executed at all). See the documentation of the nbval pluging on skipping and exceptions for details.

2.14.5 How to run a subset of tests

To run e.g. only the tests defined in `tests/test_krotov.py`, use any of the following:

```
$ make test TESTS=tests/test_krotov.py

$ tox -e py37-test -- tests/test_krotov.py

$ tox -e run-cmd -- pytest tests/test_krotov.py

$ .tox/py37/bin/pytest tests/test_krotov.py
```

See the pytest test selection docs for details.

2.14.6 How to run only as single test

Decorate the test with e.g. `@pytest.mark.xxx`, and then run, e.g:

```
$ tox -e run-cmd -- pytest -m xxx tests/
```

See the pytest documentation on markers for details.

2.14.7 How to run only the doctests

Run the following:

```
$ tox -e run-cmd -- pytest --doctest-modules src
```

2.14.8 How to go into an interactive debugger

Optionally, install the *pdbpp* package into the testing environment, for a better experience:

```
$ tox -e run-cmd -- pip install pdbpp
```

Then:

- before the line where you went to enter the debugger, insert a line:

  ```
  from IPython.terminal.debugger import set_trace; set_trace() # DEBUG
  ```

- Run `pytest` with the option `-s`, e.g.:

  ```
  $ tox -e run-cmd -- pytest -m xxx -s tests/
  ```

You may also see the pytest documentation on automatic debugging.

2.14.9 How to see the debug logger output in the example notebooks

The *optimize_pulses()* routine generates some logger messages for debugging purposes. To see these messages, set the level of "krotov" logger to INFO or DEBUG:

```
import logging
logger = logging.getLogger('krotov')
logger.setLevel(logging.DEBUG)
```

You can also configure the logger with custom formatters, e.g. to show the messages with time stamps:

```
ch = logging.StreamHandler()
ch.setLevel(logging.INFO)
formatter = logging.Formatter("%(asctime)s:%(message)s")
ch.setFormatter(formatter)
logger.addHandler(ch)
logging.getLogger().handlers = [] # disable root handlers
```

See the Configure Logging section of the Python documentation for more details.

2.14.10 How to use quantum mechanical tex macros

For docstrings or `*.rst` files, see *Write Documentation*. For notebooks, see *Math in Example Notebooks*.

3 Credits

3.1 Development Lead

- Michael Goerz <mail@michaelgoerz.net>

3.2 Development Team

- Daniel Basilewitsch <basilewitsch@physik.uni-kassel.de>
- Fernando Gago Encinas <fernando.gago@physik.uni-kassel.de>
- Matthias Krauss <matthias.krauss@physik.uni-kassel.de>
- Karl Horn <karlhorn@physik.uni-kassel.de>
- Daniel Reich <daniel.reich@physik.uni-kassel.de>
- Christiane Koch <christiane.koch@uni-kassel.de>

3.3 Acknowledgements

We thank @uiofgh, @nathanshammah, and @TejasAvinasShetty for reporting bugs and suggesting improvements.

4 Features

- Simultaneously optimize over an arbitrary list of objectives
- Optimize over multiple control fields at the same time
- Arbitrary equations of motion, through a *propagator* callback function
- Arbitrary optimization functionals, through *chi_constructor* callback function
- Allows injection of arbitrary code, through *modify_params_after_iter* function
- Customizable parallelization of the propagation of different objectives
- Customizable analysis and convergence check
- Support for dissipative dynamics (Liouville space)
- Convenience constructors for objectives describing gate optimization (in Hilbert space or Liouville space) and for “ensemble optimization” to obtain robust controls

Not yet implemented:

- non-linear controls
- state-dependent constraints

5 History

0.4.1 (2019-10-11)

- Update: Documentation now contains all information from https://arxiv.org/abs/1902.11284v4 (#54)
- Added: a PDF of the documentation is now available at https://github.com/qucontrol/krotov/tree/master/docs/pdf (#52, thanks to @TejasAvinashShetty)

0.4.0 (2019-10-08)

- Added: Support for Python 3.7
- Changed: The `'shape'` key in `pulse_options` was renamed to `'update_shape'`, to further avoid confusion between pulse shapes and update shapes.
- Changed: The `.adjoint` property of `Objective` is now a method
- Added: Ability to not use QuTiP `Qobj` objects, but arbitrary low-level objects instead.
- Improved: Printing an `Objective` now uses internal counters and a symbolic notation to identify objects shared between different objectives. (#43)
- Improved: `gate_objectives` now takes into account if target states are (reshuffled) basis states and does not create unnecessary new copies.
- Bugfix: Two `Objective` instances that contain numpy arrays as controls can now be compared with == (#44)
- Bugfix: Custom attributes (such as `weight`) are now preserved when copying an `Objective` (#44)
- Bugfix: Calling `copy.deepcopy` on an `Objective` now preserves control functions (#44)
- Improved: The `Objective.mesolve` and `Objective.propagate` methods can now receive arguments `H` and `c_ops` to override the respective attributes of the objectives. This make is easier to analyze perform a robustness analysis, where the result of an optimization should be propagated under a perturbed Hamiltonian.

- Improved: The `print_table` and `print_debug_information` info-hooks now flush their output buffers after each iteration. As a result, when writing to a file, that file can be watched with `tail -f`.
- Changed: Redefine `tau_vals` as their complex conjugate, fixing a bug in `chis_ss` and `chis_sm` (#46)
- Bugfix: Correctly calculate $\partial H/\partial \epsilon$ if ϵ occurs in H multiple times (#47, thanks to @uiofgh)
- Bugfix: Correctly calculate $\partial H/\partial \epsilon = 0$ if the specific ϵ currently being updated does not occur in H (#48)
- Added: Method `objectives_with_controls` for `Result` object.

0.3.0 (2019-03-01)

- Added: Preprint citation information (`krotov.__arxiv__`, `krotov.__citation__`, `krotov.__bibtex__`)
- Added: Ability to continue from a previous optimization (#26)
- Added: Parameter `out` to `print_table` info-hook
- Added: Parameter `finalize` to `Result.load`
- Added: Ability to dump optimization result every so many iterations (`dump_result` check-convergence routine)
- Added: *re-entrant* option for `DensityMatrixODEPropagator`
- Bugfix: Discretize controls to float values (#41)
- Bugfix: Fix overlap for non-Hermitian operators (#39)
- Bugfix: Interface for passing `tau_vals` to `chi_constructor` (#36)
- Added: function `above_value` for convergence check (#35)

0.2.0 (2019-02-14)

- Added: Implementation of all the standard functionals
- Added: The `info_hook` receives additional information, including $\int g_a(t)dt$ (#32)
- Added: Initialization of objectives for gate optimization in Liouville space
- Added: A new propagator `DensityMatrixODEPropagator` for faster density matrix propagation
- Added: Support for "stateful" propagators by subclassing from `krotov.propagators.Propagator`
- Changed: more flexibility for parallelization (#29)
- Added: Support for the second-order pulse update
- Changed: The options for the controls (λ_a, update-shape) are now passed through a simplified `dict` interface, instead of a custom `PulseOptions` class.

0.1.0 (2018-12-24)

- Initial release with complete implementation of first-order Krotov's method
- Support for state-to-state and gate optimization, for both closed and open systems

6 Introduction

Quantum information science has changed our perception of quantum physics from passive understanding to a source of technological advances [1]. By way of actively exploiting the two essential elements of quantum physics, coherence and entanglement, technologies such as quantum computing [2] or quantum sensing [3] hold the promise for solving computationally hard problems or reaching unprecedented sensitivity. These technologies rely on the ability to accurately perform quantum operations for increasingly complex quantum systems. Quantum optimal control allows to address this challenge by providing a set of tools to devise and implement shapes of external fields that accomplish a given task in the best way possible [4]. Originally developed in the context of molecular physics [5][6] and nuclear magnetic resonance [7][8], quantum optimal control theory has been adapted to the specific needs of quantum information science in recent years [4][9]. Calculation of optimized external field shapes for tasks such as state preparation or quantum gate implementation have thus become standard [4], even for large Hilbert space dimensions as encountered in e.g. Rydberg atoms [10][11]. Experimental implementation of the calculated field shapes, using arbitrary waveform generators, has been eased by the latter becoming available commercially. Successful demonstration of quantum operations in various experiments [4][12][13][14][15][16][17][18][19][20] attests to the level of maturity that quantum optimal control in quantum technologies has reached.

In order to calculate optimized external field shapes, two choices need to be made – about the optimization functional and about the optimization method. The functional consists of the desired figure of merit, such as a gate or state preparation error, as well as additional constraints, such as amplitude or bandwidth restrictions [4][9]. Optimal control methods in general can be classified into gradient-free and gradient-based algorithms that either evaluate the optimization functional alone or together with its gradient [4]. Gradient-based methods typically converge faster, unless the number of optimization parameters can be kept small. Most gradient-based methods rely on the iterative solution of a set of coupled equations that include forward propagation of initial states, backward propagation of adjoint states, and the control update [4]. A popular representative of concurrent update methods is GRadient Ascent Pulse Engineering (GRAPE) [21], whereas Krotov's method, which comes with the advantage of guaranteed monotonic convergence, requires sequential updates [5][22].

The choice of Python as an implementation language is due to Python's easy-to-learn syntax, expressiveness, and immense popularity in the scientific community. Moreover, the QuTiP library exists, providing a general purpose tool to numerically describe quantum systems and their dynamics. QuTiP already includes basic versions of other popular quantum control algorithms such as GRAPE and the gradient-free CRAB [23]. The Jupyter notebook framework is available to provide an ideal platform for the interactive exploration of the `krotov` package's

capabilities, and to facilitate reproducible research workflows.

The *krotov* package targets both students wishing to enter the field of quantum optimal control, and researchers in the field. By providing a comprehensive set of *Examples*, we enable users of our package to explore the formulation of typical control problems, and to understand how Krotov's method can solve them. These examples are inspired by recent publications [24][25][26][27][28][29], and thus show the use of the method in the purview of current research. In particular, the package is not restricted to closed quantum systems, but can fully address open system dynamics, and thus aide in the development of Noisy Intermediate-Scale Quantum (NISQ) technology [30]. Optimal control is also increasingly important in the design of experiments [4][12][13][14][15][16][17][18][19][20], and we hope that the availability of an easy-to-use implementation of Krotov's method will facilitate this further.

Large Hilbert space dimensions [31][32][10][11] and open quantum systems [26] in particular require considerable numerical effort to optimize. Compared to the Fortran and C/C++ languages traditionally used for scientific computing, and more recently Julia [33], pure Python code usually performs slower by two to three orders of magnitude [34][35]. Thus, for hard optimization problems that require several thousand iterations to converge, the Python implementation provided by the *krotov* package may not be sufficiently fast. In this case, it may be desirable to implement the entire optimization and time propagation in a single, more efficient (compiled) language. Our Python implementation of Krotov's method puts an emphasis on clarity, and the documentation provides detailed explanations of all necessary concepts, especially the correct *Time discretization* and the possibility to parallelize the optimization. Thus, the *krotov* package can serve as a reference implementation, leveraging Python's reputation as "executable pseudocode", and as a foundation against which to test other implementations.

7

Krotov's Method

7.1 The quantum control problem

Quantum optimal control methods formalize the problem of finding "control fields" that steer the time evolution of a quantum system in some desired way. For closed systems, described by a Hilbert space state $|\Psi(t)\rangle$, this time evolution is given by the Schrödinger equation,

$$\frac{\partial}{\partial t}|\Psi(t)\rangle = -\frac{\mathrm{i}}{\hbar}\hat{H}(t)|\Psi(t)\rangle \ ,$$

where the Hamiltonian $\hat{H}(t)$ depends on one or more control fields $\{\epsilon_l(t)\}$. We often assume the Hamiltonian to be linear in the controls,

$$\hat{H}(t) = \hat{H}_0 + \epsilon_1(t)\hat{H}_1 + \epsilon_2(t)\hat{H}_2 + \ldots$$

but non-linear couplings may also occur, for example when considering non-resonant multi-photon transitions. For open quantum systems described by a density matrix $\hat{\rho}(t)$, the Liouville-von-Neumann equation

$$\frac{\partial}{\partial t}\hat{\rho}(t) = \frac{1}{\hbar}\mathcal{L}(t)\hat{\rho}(t)$$

replaces the Schrödinger equation. The most direct example of a control problem is a state-to-state transition. The objective is for a known quantum state at time zero to evolve to a specific target state at final time T, controlling, e.g. a chemical reaction [36]. Another example is the realization of quantum gates, the building blocks of a quantum computer. In this case, the states forming a computational basis must transform according to a unitary transformation [2], see *How to optimize towards a quantum gate*. Thus, the control problem involves not just the time evolution of a single state, but a set of states $\{|\phi_k(t)\rangle\}$. Generalizing even further, each state $|\phi_k(t)\rangle$ in the control problem may evolve under a different Hamiltonian $\hat{H}_k(\{\epsilon_l(t)\})$, see *How to optimize for robust pulses*.

Physically, the control fields $\{\epsilon_l(t)\}$ might be the amplitudes of a laser pulse for the control of molecular systems or trapped atom/ion quantum computers, radio-frequency fields for nuclear magnetic resonance, or microwave fields for superconducting circuits. When there are multiple independent controls $\{\epsilon_l(t)\}$ involved in the dynamics, these may correspond e.g., to different color lasers used in the excitation of a Rydberg atom, or different polarization components of an electric field.

7.2 Optimization functional

The quantum control methods build on a rich field of classical control theory [37][38]. This includes Krotov's method [39][40][41][42], which was originally formulated to optimize the soft landing of a spacecraft from orbit to the surface of a planet, before being applied to quantum mechanical problems [5][43][44][45][22]. Fundamentally, they rely on the variational principle, that is, the minimization of a functional $J[\{|\phi_k^{(i)}(t)\rangle\}, \{\epsilon_l^{(i)}(t)\}]$ that includes any required constraints via Lagrange multipliers. The condition for minimizing J is then $\nabla_{\phi_k, \epsilon_l} J = 0$. In rare cases, the variational calculus can be solved in closed form, based on Pontryagin's maximum principle [38]. Numerical methods are required in any other case. These start from an initial guess control (or set of guess controls, if there are multiple controls), and calculate an update to these controls that will decrease the value of the functional. The updated controls then become the guess for the next iteration of the algorithm, until the value of the functional is sufficiently small, or convergence is reached.

Mathematically, Krotov's method, when applied to quantum systems [5][22], minimizes a functional of the most general form

$$J[\{|\phi_k^{(i)}(t)\rangle\}, \{\epsilon_l^{(i)}(t)\}] = J_T(\{|\phi_k^{(i)}(T)\rangle\}) + \sum_l \int_0^T g_a(\epsilon_l^{(i)}(t))\,\mathrm{d}t + \int_0^T g_b(\{\phi_k^{(i)}(t)\})\,\mathrm{d}t\,, \tag{7.1}$$

where the $\{|\phi_k^{(i)}(T)\rangle\}$ are the time-evolved initial states $\{|\phi_k\rangle\}$ under the controls $\{\epsilon_l^{(i)}(t)\}$ of the i'th iteration ($i > 0$). That is, the field $\epsilon_l^{(i)}(t)$ is the result of

$$\epsilon_l^{(i)}(t) = \epsilon_l^{(i-1)}(t) + \Delta\epsilon_l^{(i)}(t) \tag{7.2}$$

for the control update $\Delta\epsilon_l^{(i)}(t)$ and the guess control $\epsilon_l^{(i-1)}(t)$, starting from the initial guess control $\epsilon_l^{(0)}(t)$. The update is constructed such that the value of the functional decreases,

$$J[\{|\phi_k^{(i)}(t)\rangle\}, \{\epsilon_l^{(i)}(t)\}] \leq J[\{|\phi_k^{(i-1)}(t)\rangle\}, \{\epsilon_l^{(i-1)}(t)\}]\,.$$

In the simplest case of a single state-to-state transition, the index k vanishes. For the example of a two-qubit quantum gate, $\{|\phi_k\rangle\}$ would be the logical basis states $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$, all evolving under the same Hamiltonian $\hat{H}_k \equiv \hat{H}$. The sum over l vanishes if there is only a single control. For open system dynamics, the states $\{|\phi_k\rangle\}$ may be density matrices.

The functional consists of three parts:

- A final time functional J_T. This is the "main" part of the functional, and we can usually think of J as being an auxiliary functional in the optimization of J_T.
- A running cost on the control fields, g_a. The most commonly used expression (and the only one currently supported by the *krotov* package) is [46]

$$\begin{aligned} g_a(\epsilon_l^{(i)}(t)) &= \frac{\lambda_{a,l}}{S_l(t)} \left(\epsilon_l^{(i)}(t) - \epsilon_{l,\mathrm{ref}}^{(i)}(t)\right)^2 ; \quad \epsilon_{l,\mathrm{ref}}^{(i)}(t) = \epsilon_l^{(i-1)}(t) \\ &= \frac{\lambda_{a,l}}{S_l(t)} \left(\Delta\epsilon_l^{(i)}(t)\right)^2 , \end{aligned} \tag{7.3}$$

  with $\lambda_{a,l} > 0$ and $S_l(t) \in [0, 1]$. For this specific form, using the guess control field $\epsilon_l^{(i-1)}(t)$ as the "reference" field for the optimized $\epsilon_l^{(i)}(t)$, the update $\Delta\epsilon_l^{(i)}(t)$ in each iteration will be proportional to $\frac{S_l(t)}{\lambda_{a,l}}$, see *First order update*. Note that this also makes g_a proportional to $\frac{S_l(t)}{\lambda_{a,l}}$, so that g_a is still well-defined for $S_l(t) = 0$. The (inverse) Krotov "step width" $\lambda_{a,l}$ can be used to determine the overall magnitude of $\Delta\epsilon_l^{(i)}(t)$. Values that are too large

will change $\epsilon_l^{(i)}(t)$ by only a small amount in every iteration, causing slow convergence. Values that are too small will result in numerical instability, see *Time discretization* and *Choice of λ_a*. The "update shape" function $S_l(t)$ allows to ensure boundary conditions on $\epsilon_l^{(i)}(t)$: If both the guess field $\epsilon_l^{(i-1)}(t)$ and $S_l(t)$ switch on and off smoothly around $t = 0$ and $t = T$, then this feature will be preserved by the optimization. A typical example for an update shape is

$$S_l(t) = \begin{cases} B(t; t_0 = 0, t_1 = 2t_{\text{on}}) & \text{for} \quad 0 < t < t_{\text{on}} \\ 1 & \text{for} \quad t_{\text{on}} \leq t \leq T - t_{\text{off}} \\ B(t; t_0 = T - 2t_{\text{off}}, t_1 = T) & \text{for} \quad T - t_{\text{off}} < t < T\,, \end{cases} \tag{7.4}$$

cf. `krotov.shapes.flattop()`, with the Blackman shape

$$B(t; t_0, t_1) = \frac{1}{2}\left(1 - a - \cos\left(2\pi\frac{t - t_0}{t_1 - t_0}\right) + a\cos\left(4\pi\frac{t - t_0}{t_1 - t_0}\right)\right)\,, \quad a = 0.16\,, \tag{7.5}$$

which is similar to a Gaussian, but exactly zero at $t = t_0, t_1$. Moreover, any part of the control field can be kept unchanged in the optimization by choosing $S_l(t) = 0$ for the corresponding intervals of the time grid.

- An optional state-dependent running cost, g_b. This may be used to encode time-dependent control targets [47][48], or to penalize population in a subspace [49]. The presence of a state-dependent constraint in the functional entails an inhomogeneous term in the backward propagation in the calculation of the control updates in each iteration of Krotov's method, see Eq. (7.13), and is currently not supported by the `krotov` package. Penalizing population in a subspace can also be achieved through simpler methods that do not require a g_b, e.g., by using a non-Hermitian Hamiltonian to remove population from the forbidden subspace during the time evolution.

The most commonly used final-time functionals (cf. `krotov.functionals`) optimize for a set of initial states $\{|\phi_k\rangle\}$ to evolve to a set of target states $\{|\phi_k^{\text{tgt}}\rangle\}$. The functionals can then be expressed in terms of the complex overlaps of the target states with the final-time states under the given control. Thus,

$$\tau_k = \left\langle \phi_k^{\text{tgt}} \mid \phi_k(T) \right\rangle \tag{7.6}$$

in Hilbert space, or

$$\tau_k = \langle\!\langle \hat{\rho}^{\text{tgt}} | \hat{\rho}_k(T) \rangle\!\rangle \equiv \text{tr}\left[\hat{\rho}_k^{\text{tgt}\,\dagger}\hat{\rho}_k(T)\right]$$

in Liouville space.

The following functionals J_T can be formed from these complex overlaps, taking into account that any optimization functional J_T must be real. They differ by the way they treat the phases φ_k in the physical optimization goal $|\phi_k(T)\rangle \stackrel{!}{=} e^{i\varphi_k}|\phi_k^{\text{tgt}}\rangle$ [46]:

- Optimize for simultaneous state-to-state transitions, with completely arbitrary phases φ_k,

$$J_{T,\text{ss}} = 1 - \frac{1}{N}\sum_{k=1}^{N} |\tau_k|^2\,, \tag{7.7}$$

cf. `J_T_ss()`.

- Optimize for simultaneous state-to-state transitions, with an arbitrary *global* phase, i.e., $\varphi_k = \varphi_{\text{global}}$ for all k with arbitrary φ_{global},

$$J_{T,\text{sm}} = 1 - \frac{1}{N^2}\left|\sum_{k=1}^{N}\tau_k\right|^2 = 1 - \frac{1}{N^2}\sum_{k=1}^{N}\sum_{k'=1}^{N}\tau_{k'}^{*}\tau_k\,, \tag{7.8}$$

cf. `J_T_sm()`.

- Optimize for simultaneous state-to-state transitions, with a global phase of zero, i.e., $\varphi_k = 0$ for all k,

$$J_{T,\mathrm{re}} = 1 - \frac{1}{N} \mathrm{Re}\left[\sum_{k=1}^{N} \tau_k\right], \tag{7.9}$$

cf. `J_T_re()`.

Note: In the remainder of this chapter, we review some of the mathematical details of how Krotov's method calculates the update in Eq. (7.2). These details are not necessary to *use* the `krotov` package as a "black box" optimization tool, so you may skip ahead to *Using Krotov with QuTiP* and come back at a later time.

7.3 First order update

Krotov's method is based on a rigorous examination of the conditions for calculating the updated fields $\{\epsilon_l^{(i)}(t)\}$ such that $J(\{|\phi_k^{(i)}(t)\rangle\}, \{\epsilon_l^{(i)}(t)\}) \leq J(\{|\phi_k^{(i-1)}(t)\rangle\}, \{\epsilon_l^{(i-1)}(t)\})$ is true *by construction* [41][42][46][45][22]. For a general functional of the form in Eq. (7.1), with a convex final-time functional J_T, the condition for monotonic convergence is

$$\left.\frac{\partial g_a}{\partial \epsilon_l(t)}\right|_{(i)} = 2\,\mathrm{Im}\left[\sum_{k=1}^{N} \left\langle \chi_k^{(i-1)}(t) \middle| \left(\left.\frac{\partial \hat{H}}{\partial \epsilon_l(t)}\right|_{(i)} \right) \middle| \phi_k^{(i)}(t) \right\rangle\right], \tag{7.10}$$

see Ref. [46]. If there are multiple controls, the condition holds for every control field $\epsilon_l(t)$ independently.

For g_a as in Eq. (7.3), this results in an *update* equation [5][46][45],

$$\Delta\epsilon_l^{(i)}(t) = \frac{S_l(t)}{\lambda_{a,l}}\,\mathrm{Im}\left[\sum_{k=1}^{N} \left\langle \chi_k^{(i-1)}(t) \middle| \left(\left.\frac{\partial \hat{H}}{\partial \epsilon_l(t)}\right|_{(i)} \right) \middle| \phi_k^{(i)}(t) \right\rangle\right], \tag{7.11}$$

with the equation of motion for the forward propagation of $|\phi_k^{(i)}\rangle$ under the optimized controls $\{\epsilon_l^{(i)}(t)\}$ of the iteration (i),

$$\frac{\partial}{\partial t}\left|\phi_k^{(i)}(t)\right\rangle = -\frac{\mathrm{i}}{\hbar}\hat{H}^{(i)}\left|\phi_k^{(i)}(t)\right\rangle. \tag{7.12}$$

The co-states $|\chi_k^{(i-1)}(t)\rangle$ are propagated backwards in time under the guess controls of iteration (i), i.e., the optimized controls from the previous iteration $(i-1)$, as

$$\frac{\partial}{\partial t}\left|\chi_k^{(i-1)}(t)\right\rangle = -\frac{\mathrm{i}}{\hbar}\hat{H}^{\dagger\,(i-1)}\left|\chi_k^{(i-1)}(t)\right\rangle + \left.\frac{\partial g_b}{\partial \langle\phi_k|}\right|_{(i-1)}, \tag{7.13}$$

with the boundary condition

$$\left|\chi_k^{(i-1)}(T)\right\rangle = -\left.\frac{\partial J_T}{\partial \langle\phi_k(T)|}\right|_{(i-1)}, \tag{7.14}$$

where the right-hand-side is evaluated for the set of states $\{|\phi_k^{(i-1)}(T)\rangle\}$ resulting from the forward-propagation of the initial states under the guess controls of iteration (i) – that is, the optimized controls of the previous iteration $(i-1)$.

For example, for the functional $J_{T,\text{ss}}$ in Eq. (7.7) for a single state-to-state transition ($N = 1$),

$$\begin{aligned}|\chi^{(i-1)}(T)\rangle &= \frac{\partial}{\partial \langle\phi(T)|} \underbrace{\langle\phi(T) \mid \phi^{\text{tgt}}\rangle \langle\phi^{\text{tgt}} \mid \phi(T)\rangle}_{=\left|\left\langle\phi^{\text{tgt}}|\phi(T)\right\rangle\right|^2}\Bigg|_{(i-1)} \\ &= \left(\left\langle\phi^{\text{tgt}} \mid \phi^{(i-1)}(T)\right\rangle\right) |\phi^{\text{tgt}}\rangle \,,\end{aligned}$$

cf. *krotov.functionals.chis_ss()* and the *krotov.functionals* module in general.

7.4 Second order update

The update Eq. (7.11) assumes that the equation of motion is linear ($\hat{H}$ does not depend on the states $|\phi_k(t)\rangle$), the functional J_T is convex, and no state-dependent constraints are used ($g_b \equiv 0$). When any of these conditions are not fulfilled, it is still possible to derive an optimization algorithm with monotonic convergence via a "second order" term in Eqs. (7.10), (7.11) [42][22],

The full update equation then reads

$$\begin{aligned}\Delta\epsilon_l^{(i)}(t) = \frac{S_l(t)}{\lambda_{a,l}} \text{Im} \Bigg[&\sum_{k=1}^{N} \left\langle \chi_k^{(i-1)}(t) \middle| \left(\frac{\partial \hat{H}}{\partial \epsilon_l(t)}\middle|_{(i)} \right) \middle| \phi_k^{(i)}(t) \right\rangle \\ &+ \frac{1}{2}\sigma(t) \left\langle \Delta\phi_k^{(i)}(t) \middle| \left(\frac{\partial \hat{H}}{\partial \epsilon_l(t)}\middle|_{(i)} \right) \middle| \phi_k^{(i)}(t) \right\rangle \Bigg] \,,\end{aligned} \tag{7.15}$$

with

$$|\Delta\phi_k^{(i)}(t)\rangle \equiv |\phi_k^{(i)}(t)\rangle - |\phi_k^{(i-1)}(t)\rangle \,,$$

see Ref. [22] for the full construction of the second-order condition. In Eq. (7.15), $\sigma(t)$ is a scalar function that must be properly chosen to ensure monotonic convergence.

In Refs. [27][28], a non-convex final-time functional for the optimization towards an arbitrary perfect entangler is considered. For this specific example, a suitable choice is

$$\sigma(t) \equiv -\max\left(\varepsilon_A, 2A + \varepsilon_A\right) \,,$$

where ε_A is a small non-negative number. The optimal value for A in each iteration can be approximated numerically as [22]

$$A = \frac{\sum_{k=1}^{N} 2\,\text{Re}\left[\langle\chi_k(T)|\Delta\phi_k(T)\rangle\right] + \Delta J_T}{\sum_{k=1}^{N} |\Delta\phi_k(T)|^2} \,,$$

cf. *krotov.second_order.numerical_estimate_A()*, with

with

$$\Delta J_T \equiv J_T(\{\phi_k^{(i)}(T)\}) - J_T(\{\phi_k^{(i-1)}(T)\}) \,.$$

See the *Optimization towards a Perfect Entangler* for an example.

Note: Even when the second order update equation is mathematically required to guarantee monotonic convergence, very often an optimization with the first-order update equation (7.11) will give converging results. Since the second order update requires more numerical resources (calculation and storage of the states $|\Delta\phi_k(t)\rangle$), you should always try the optimization with the first-order update equation first.

7.5 Time discretization

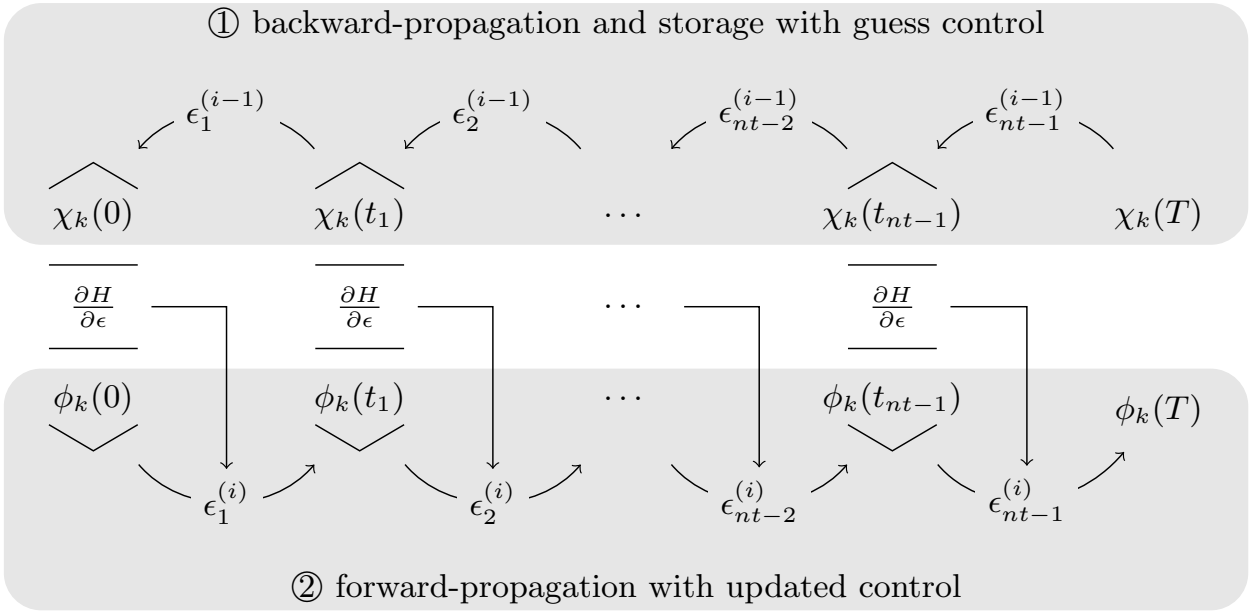


Fig. 7.1: Sequential update scheme in Krotov's method on a time grid.

The derivation of Krotov's method assumes time-continuous control fields. Only in this case, monotonic convergence is mathematically guaranteed. However, for practical numerical applications, we have to consider controls on a discrete time grid with nt points running from $t = 0$ to $t = T$, with a time step $\mathrm{d}t$. The states are defined on the points of the time grid, while the controls are assumed to be constant on the intervals of the time grid. See the notebook Time Discretization in Quantum Optimal Control for details.

The discretization yields the numerical scheme shown in Fig. 7.1 for a single control field (no index l), and assuming the first-order update is sufficient to guarantee monotonic convergence for the chosen functional. For simplicity, we also assume that the Hamiltonian is linear in the control, so that $\partial \hat{H}/\partial \epsilon(t)$ is not time-dependent. The scheme proceeds as follows:

1. Construct the states $\{|\chi_k^{(i-1)}(T)\rangle\}$ according to Eq. (7.14). For most functionals, specifically any that are more than linear in the overlaps τ_k defined in Eq. (7.6), the states $\{|\chi_k^{(i-1)}(T)\rangle\}$ depend on the states $\{|\phi_k^{(i-1)}(T)\rangle\}$ forward-propagated under the optimized pulse from the previous iteration, that is, the guess pulse in the current iteration.
2. Perform a backward propagation using Eq. (7.13) as the equation of motion over the entire time grid. The resulting state at each point in the time grid must be stored in memory.
3. Starting from the known initial states $\{|\phi_k\rangle\} = \{|\phi_k(t = t_0 = 0)\rangle\}$, calculate the pulse update for the first time step according to

$$\Delta\epsilon_1^{(i)} \equiv \Delta\epsilon^{(i)}(\tilde{t}_0) = \frac{S(\tilde{t}_0)}{\lambda_a} \mathrm{Im}\left[\sum_{k=1}^{N} \left\langle \chi_k^{(i-1)}(t_0) \middle| \frac{\partial \hat{H}}{\partial \epsilon} \middle| \phi_k(t_0) \right\rangle\right] . \tag{7.16}$$

   The value $\Delta\epsilon_1^{(i)}$ is taken on the midpoint of the first time interval, $\tilde{t}_0 \equiv t_0 + \mathrm{d}t/2$, based on the assumption of a piecewise-constant control field and an equidistant time grid with spacing $\mathrm{d}t$.

4. Use the updated field $\epsilon_1^{(i)}$ for the first interval to propagate $|\phi_k(t=t_0)\rangle$ for a single time step to $|\phi_k^{(i)}(t=t_0+\mathrm{d}t)\rangle$, with Eq. (7.12) as the equation of motion. The updates then proceed sequentially, using the discretized update equation

$$\Delta\epsilon_{n+1}^{(i)} \equiv \Delta\epsilon^{(i)}(\tilde{t}_n) = \frac{S(\tilde{t}_n)}{\lambda_a}\,\mathrm{Im}\left[\sum_{k=1}^{N}\left\langle \chi_k^{(i-1)}(t_n)\middle|\frac{\partial\hat{H}}{\partial\epsilon}\middle|\phi_k^{(i)}(t_n)\right\rangle\right] \tag{7.17}$$

   with $\tilde{t}_n \equiv t_n + \mathrm{d}t/2$ for each time interval n, until the final forward-propagated state $|\phi_k^{(i)}(T)\rangle$ is reached.

5. The updated control field becomes the guess control for the next iteration of the algorithm, starting again at step 1. The optimization continues until the value of the functional J_T falls below some predefined threshold, or convergence is reached, i.e., ΔJ_T approaches zero so that no further significant improvement of J_T is to be expected.

Eq. (7.11) re-emerges as the continuous limit of the time-discretized update equation (7.17), i.e., $\mathrm{d}t \to 0$ so that $\tilde{t}_n \to t_n$. Note that Eq. (7.17) resolves the seeming contradiction in the time-continuous Eq. (7.11) that the calculation of $\epsilon^{(i)}(t)$ requires knowledge of the states $|\phi_k^{(i)}(t)\rangle$ which would have to be obtained from a propagation under $\epsilon^{(i)}(t)$. By having the time argument $\tilde{t}_n$ on the left-hand-side of Eq. (7.17), and $t_n < \tilde{t}_n$ on the right-hand-side (with $S(\tilde{t}_n)$ known at all times), the update for each interval only depends on "past" information.

For multiple objectives, the scheme can run in parallel, and each objective contributes a term to the update. Summation of these terms yields the sum in Eq. (7.11). See *krotov.parallelization* for details. For a second-order update, the forward propagated states from step 4, both for the current iteration and the previous iteration, must be stored in memory over the entire time grid.

7.6 Pseudocode

A complete pseudocode for Krotov's method as described in the previous section *Time discretization* is available in PDF format: https://krotov.readthedocs.io/en/latest/krotov_pseudocode.pdf.

7.7 Choice of λ_a

The monotonic convergence of Krotov's method is only guaranteed in the continuous limit; a coarse time step must be compensated by larger values of the inverse step size $\lambda_{a,l}$, slowing down convergence. Values that are too small will cause sharp spikes in the optimized control and numerical instabilities. A lower limit for $\lambda_{a,l}$ can be determined from the requirement that the change $\Delta\epsilon_l^{(i)}(t)$ should be at most of the same order of magnitude as the guess pulse $\epsilon_l^{(i-1)}(t)$ for that iteration. The Cauchy-Schwarz inequality applied to the update equation (7.11) yields

$$\|\Delta\epsilon_l(t)\|_\infty \leq \frac{\|S(t)\|}{\lambda_{a,l}} \sum_k \|\,|\chi_k(t)\rangle\|_\infty \,\|\,|\phi_k(t)\rangle\|_\infty \left\|\frac{\partial\hat{H}}{\partial\epsilon_l(t)}\right\|_\infty \overset{!}{\leq} \left\|\epsilon_l^{(i)}(t)\right\|_\infty ,$$

where $\|\partial\hat{H}/\partial\epsilon_l(t)\|_\infty$ denotes the supremum norm of the operator $\partial\hat{H}/\partial\epsilon_l$ obtained at time t. Since $S(t) \in [0,1]$ and $|\phi_k\rangle$ are normalized, the condition for $\lambda_{a,l}$ becomes

$$\lambda_{a,l} \geq \frac{1}{\left\|\epsilon_l^{(i)}(t)\right\|_\infty}\left[\sum_k \|\,|\chi_k(t)\rangle\|_\infty\right]\left\|\frac{\partial\hat{H}}{\partial\epsilon_l(t)}\right\|_\infty .$$

From a practical point of view, the best strategy is to start the optimization with a comparatively large value of $\lambda_{a,l}$, and after a few iterations lower $\lambda_{a,l}$ as far as possible without introducing numerical instabilities. In principle, the value of $\lambda_{a,l}$ may be adjusted dynamically with respect to the rate of convergence, via the *modify_params_after_iter* argument to `optimize_pulses()`. Generally, the ideal choice of $\lambda_{a,l}$ requires some trial and error, but once a suitable value has been found, it does not have to be adjusted further. In particular, it is not necessary to perform a line search over $\lambda_{a,l}$.

7.8 Complex controls and the RWA

When using the rotating wave approximation (RWA), it is important to remember that the target states are usually defined in the lab frame, not in the rotating frame. This is relevant for the construction of $|\chi_k(T)\rangle$. When doing a simple optimization, such as a state-to-state or a gate optimization, the easiest approach is to transform the target states to the rotating frame before calculating $|\chi_k(T)\rangle$. This is both straightforward and numerically efficient.

Another solution would be to transform the result of the forward propagation $|\phi_k(T)\rangle$ from the rotating frame to the lab frame, then constructing $|\chi_k(T)\rangle$, and finally to transform $|\chi_k(T)\rangle$ back to the rotating frame, before starting the backward propagation.

When the RWA is used, the control fields are complex-valued. In this case the Krotov update equation is valid for both the real and the imaginary part independently. The most straightforward implementation of the method is for real controls only, requiring that any complex control Hamiltonian is rewritten as two independent control Hamiltonians, one for the real part and one for the imaginary part of the control field. For example,

$$\epsilon^*(t)\hat{a} + \epsilon(t)\hat{a}^\dagger = \epsilon_{\mathrm{re}}(t)(\hat{a} + \hat{a}^\dagger) + \epsilon_{\mathrm{im}}(t)(i\hat{a}^\dagger - i\hat{a})$$

with two independent control fields $\epsilon_{\mathrm{re}}(t) = \mathrm{Re}[\epsilon(t)]$ and $\epsilon_{\mathrm{im}}(t) = \mathrm{Im}[\epsilon(t)]$.

See the *Optimization of a State-to-State Transfer in a Lambda System in the RWA* for an example.

7.9 Optimization in Liouville space

The coupled equations (7.11)–(7.13) can be generalized to open system dynamics by replacing Hilbert space states with density matrices, $\hat{H}$ with $\mathrm{i}\mathcal{L}$, and brakets (inner products) with Hilbert-Schmidt products, $\langle\cdot|\cdot\rangle \rightarrow \langle\langle\cdot|\cdot\rangle\rangle$. In full generality, $\hat{H}$ in Eq. (7.11) is the operator H on the right-hand side of whatever the equation of motion for the forward propagation of the states is, written in the form $\mathrm{i}\hbar\dot{\phi} = H\phi$, cf. Eq. (7.12). See `krotov.mu` for details.

Note also that the backward propagation Eq. (7.13) uses the adjoint H, which is relevant both for a dissipative Liouvillian [50][51][26] and a non-Hermitian Hamiltonian [24][52].

See the *Optimization of Dissipative Qubit Reset* for an example.

8

Using Krotov with QuTiP

The *krotov* package is designed around QuTiP, a very powerful "Quantum Toolbox" in Python. This means that all operators and states are expressed as `qutip.Qobj` quantum objects. The *optimize_pulses()* interface for Krotov's optimization method is closely linked to the interface of QuTiP's central `mesolve()` routine for simulating the system dynamics of a closed or open quantum system. In particular, when setting up an optimization, the (time-dependent) system Hamiltonian should be represented by a nested list. This is, a Hamiltonian of the form $\hat{H} = \hat{H}_0 + \epsilon(t)\hat{H}_1$ is represented as `H = [H0, [H1, eps]]` where `H0` and `H1` are `Qobj` operators, and `eps` is a function with signature `eps(t, args)`, or an array of control values with the length of the time grid (*tlist* parameter in `mesolve()`). The operator can depend on multiple controls, resulting in expressions of the form `H = [H0, [H1, eps1], [H2, eps2], ...]`.

The central routine provided by the *krotov* package is *optimize_pulses()*. It takes as input a list of objectives, each of which is an instance of *Objective*. Each objective has an *initial_state*, which is a `qutip.Qobj` representing a Hilbert space state or density matrix, a *target* (usually the target state that the *initial_state* should evolve into when the objective is fulfilled), and a Hamiltonian *H* in the nested-list format described above. For dissipative dynamics, *H* should be a Liouvillian, which can be obtained from the Hamiltonian and a set of Lindblad operators via *krotov.objectives.liouvillian()*. The Liouvillian again is in nested list format to express time-dependencies. Alternatively, each objective could also directly include a list *c_ops* of collapse (Lindblad) operators , where each collapse operator is a `Qobj` operator. However, this only makes sense if the time propagation routine takes the collapse operators into account explicitly, such as in the Monte-Carlo `mcsolve()`. Otherwise, the use of *c_ops* is strongly discouraged.

In order to simulate the dynamics of the guess control, you can use *Objective.mesolve()*, which delegates to `qutip.mesolve.mesolve()`. There is also a related method *Objective.propagate()* that uses a different sampling of the control values, see *krotov.propagators*.

The optimization routine will automatically extract all controls that it can find in the objectives, and iteratively calculate updates to all controls in order to meet all *objectives* simultaneously. The result of the optimization will be in the returned *Result* object, with a list of the optimized controls in *optimized_controls*. The *optimized_objectives* property contains a copy of the objectives with the *optimized_controls* plugged into the Hamiltonian or Liouvillian and/or collapse operators. The dynamics under the optimized controls can then again be simulated through *Objective.mesolve()*.

While the guess controls that are in the *objectives* on input may be functions, or an array of control values on the time grid, the output *optimized_controls* will always be an array of control values.

9 Examples

9.1 Optimization of a State-to-State Transfer in a Two-Level-System

[1]:
```
# NBVAL_IGNORE_OUTPUT
%load_ext watermark
import qutip
import numpy as np
import scipy
import matplotlib
import matplotlib.pylab as plt
import krotov
%watermark -v --iversions
```

```
qutip            4.4.1
numpy            1.17.2
matplotlib       3.1.1
krotov           0.4.1
matplotlib.pylab 1.17.2
scipy            1.3.1
CPython 3.7.3
IPython 7.8.0
```

This first example illustrates the basic use of the `krotov` package by solving a simple canonical optimization problem: the transfer of population in a two level system.

9.1.1 Two-level-Hamiltonian

We consider the Hamiltonian $\hat{H}_0 = -\frac{\omega}{2}\hat{\sigma}_z$, representing a simple qubit with energy level splitting ω in the basis $\{|0\rangle, |1\rangle\}$. The control field $\epsilon(t)$ is assumed to couple via the Hamiltonian $\hat{H}_1(t) = \epsilon(t)\hat{\sigma}_x$ to the qubit, i.e., the control field effectively drives transitions between both qubit states.

[2]:
```
def hamiltonian(omega=1.0, ampl0=0.2):
    """Two-level-system Hamiltonian
```

(continues on next page)

(continued from previous page)

```
    Args:
        omega (float): energy separation of the qubit levels
        ampl0 (float): constant amplitude of the driving field
    """
    H0 = -0.5 * omega * qutip.operators.sigmaz()
    H1 = qutip.operators.sigmax()

    def guess_control(t, args):
        return ampl0 * krotov.shapes.flattop(
            t, t_start=0, t_stop=5, t_rise=0.3, func="blackman"
        )

    return [H0, [H1, guess_control]]
```

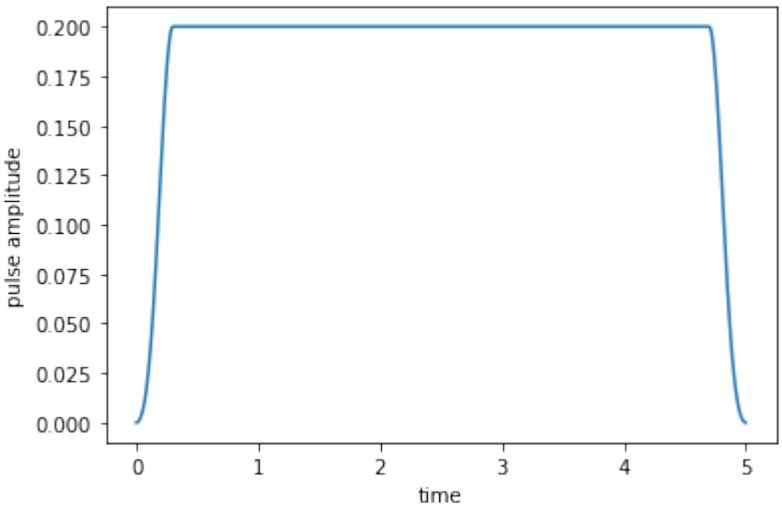
[3]:
```
H = hamiltonian()
```

The control field here switches on from zero at $t = 0$ to it's maximum amplitude 0.2 within the time period 0.3 (the switch-on shape is half a Blackman pulse). It switches off again in the time period 0.3 before the final time $T = 5$). We use a time grid with 500 time steps between 0 and T:

[4]:
```
tlist = np.linspace(0, 5, 500)
```

[5]:
```
def plot_pulse(pulse, tlist):
    fig, ax = plt.subplots()
    if callable(pulse):
        pulse = np.array([pulse(t, args=None) for t in tlist])
    ax.plot(tlist, pulse)
    ax.set_xlabel('time')
    ax.set_ylabel('pulse amplitude')
    plt.show(fig)
```

[6]:
```
plot_pulse(H[1][1], tlist)
```



9.1.2 Optimization target

The `krotov` package requires the goal of the optimization to be described by a list of `Objective` instances. In this example, there is only a single objective: the state-to-state transfer from initial state $|\Psi_{\text{init}}\rangle = |0\rangle$ to the target state $|\Psi_{\text{tgt}}\rangle = |1\rangle$, under the dynamics of the Hamiltonian $\hat{H}(t)$:

[7]:
```
objectives = [
    krotov.Objective(
        initial_state=qutip.ket("0"), target=qutip.ket("1"), H=H
    )
]

objectives
```

[7]:
```
[Objective[|Ψ₀(2)⟩ to |Ψ₁(2)⟩ via [H₀[2,2], [H₁[2,2], u₁(t)]]]]
```

In addition, we would like to maintain the property of the control field to be zero at $t = 0$ and $t = T$, with a smooth switch-on and switch-off. We can define an "update shape" $S(t) \in [0, 1]$ for this purpose: Krotov's method will update the field at each point in time proportionally to $S(t)$; wherever $S(t)$ is zero, the optimization will not change the value of the control from the original guess.

[8]:
```
def S(t):
    """Shape function for the field update"""
    return krotov.shapes.flattop(
        t, t_start=0, t_stop=5, t_rise=0.3, t_fall=0.3, func='blackman'
    )
```

Beyond the shape, Krotov's method uses a parameter λ_a for each control field that determines the overall magnitude of the respective field in each iteration (the smaller λ_a, the larger the update; specifically, the update is proportional to $\frac{S(t)}{\lambda_a}$). Both the update-shape $S(t)$ and the λ_a parameter must be passed to the optimization routine as "pulse options":

[9]:
```
pulse_options = {
    H[1][1]: dict(lambda_a=5, update_shape=S)
}
```

9.1.3 Simulate dynamics under the guess field

Before running the optimization procedure, we first simulate the dynamics under the guess field $\epsilon_0(t)$. The following solves equation of motion for the defined objective, which contains the initial state $|\Psi_{\text{init}}\rangle$ and the Hamiltonian $\hat{H}(t)$ defining its evolution. This delegates to QuTiP's usual `mesolve` function.

We use the projectors $\hat{P}_0 = |0\rangle\langle 0|$ and $\hat{P}_1 = |1\rangle\langle 1|$ for calculating the population:

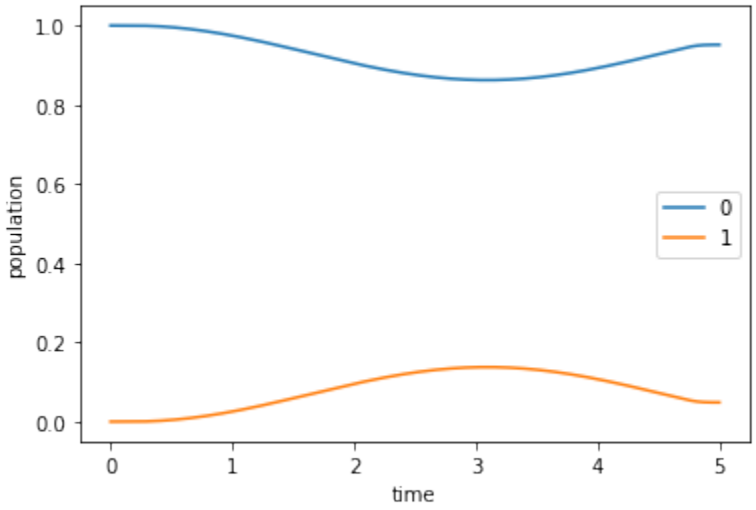
[10]:
```
proj0 = qutip.ket2dm(qutip.ket("0"))
proj1 = qutip.ket2dm(qutip.ket("1"))
```

[11]:
```
guess_dynamics = objectives[0].mesolve(tlist, e_ops=[proj0, proj1])
```

The plot of the population dynamics shows that the guess field does not transfer the initial state $|\Psi_{\text{init}}\rangle = |0\rangle$ to the desired target state $|\Psi_{\text{tgt}}\rangle = |1\rangle$ (so the optimization will have something to do).

[12]:
```
def plot_population(result):
    fig, ax = plt.subplots()
    ax.plot(result.times, result.expect[0], label='0')
    ax.plot(result.times, result.expect[1], label='1')
    ax.legend()
    ax.set_xlabel('time')
    ax.set_ylabel('population')
    plt.show(fig)
```

[13]:
```
plot_population(guess_dynamics)
```



9.1.4 Optimize

In the following we optimize the guess field $\epsilon_0(t)$ such that the intended state-to-state transfer $|\Psi_{\text{init}}\rangle \rightarrow |\Psi_{\text{tgt}}\rangle$ is solved, via the `krotov` package's central `optimize_pulses` routine. It requires, besides the previously defined `objectives`, information about the optimization functional J_T (implicitly, via `chi_constructor`, which calculates the states $|\chi\rangle = \frac{J_T}{\langle\Psi|}$).

Here, we choose $J_T = J_{T,\text{ss}} = 1 - F_{\text{ss}}$ with $F_{\text{ss}} = \left|\langle\Psi_{\text{tgt}} \mid \Psi(T)\rangle\right|^2$, with $|\Psi(T)\rangle$ the forward propagated state of $|\Psi_{\text{init}}\rangle$. Even though J_T is not explicitly required for the optimization, it is nonetheless useful to be able to calculate and print it as a way to provide some feedback about the optimization progress. Here, we pass as an `info_hook` the function `krotov.info_hooks.print_table`, using `krotov.functionals.J_T_ss` (which implements the above functional; the `krotov` library contains implementations of all the "standard" functionals used in quantum control). This `info_hook` prints a tabular overview after each iteration, containing the value of J_T, the magnitude of the integrated pulse update, and information on how much J_T (and the full Krotov functional J) changes between iterations. It also stores the value of J_T internally in the `Result.info_vals` attribute.

The value of J_T can also be used to check the convergence. In this example, we limit the number of total iterations to 10, but more generally, we could use the `check_convergence` parameter to stop the optimization when J_T falls below some threshold. Here, we only pass a function that checks that the value of J_T is monotonically decreasing. The `krotov.convergence.check_monotonic_error` relies on `krotov.info_hooks.print_table` internally having stored the value of J_T to the `Result.info_vals` in each iteration.

[14]:
```
opt_result = krotov.optimize_pulses(
    objectives,
    pulse_options=pulse_options,
    tlist=tlist,
    propagator=krotov.propagators.expm,
    chi_constructor=krotov.functionals.chis_ss,
    info_hook=krotov.info_hooks.print_table(J_T=krotov.functionals.J_T_ss),
    check_convergence=krotov.convergence.Or(
        krotov.convergence.value_below('1e-3', name='J_T'),
        krotov.convergence.check_monotonic_error,
    ),
    store_all_pulses=True,
)
```

```
  iter.        J_T    ∫gₐ(t)dt          J       ΔJ_T         ΔJ  secs
     0   9.51e-01    0.00e+00   9.51e-01        n/a        n/a     0
     1   9.24e-01    2.32e-03   9.27e-01  -2.70e-02  -2.47e-02     1
     2   8.83e-01    3.53e-03   8.87e-01  -4.11e-02  -3.75e-02     1
     3   8.23e-01    5.22e-03   8.28e-01  -6.06e-02  -5.54e-02     1
     4   7.38e-01    7.39e-03   7.45e-01  -8.52e-02  -7.78e-02     1
     5   6.26e-01    9.75e-03   6.36e-01  -1.11e-01  -1.01e-01     1
     6   4.96e-01    1.16e-02   5.07e-01  -1.31e-01  -1.19e-01     1
     7   3.62e-01    1.21e-02   3.74e-01  -1.34e-01  -1.22e-01     1
     8   2.44e-01    1.09e-02   2.55e-01  -1.18e-01  -1.07e-01     1
     9   1.53e-01    8.43e-03   1.62e-01  -9.03e-02  -8.19e-02     1
    10   9.20e-02    5.80e-03   9.78e-02  -6.14e-02  -5.56e-02     1
    11   5.35e-02    3.66e-03   5.72e-02  -3.85e-02  -3.48e-02     1
    12   3.06e-02    2.19e-03   3.28e-02  -2.29e-02  -2.07e-02     2
    13   1.73e-02    1.27e-03   1.86e-02  -1.33e-02  -1.20e-02     2
    14   9.79e-03    7.24e-04   1.05e-02  -7.55e-03  -6.82e-03     1
    15   5.52e-03    4.10e-04   5.93e-03  -4.27e-03  -3.86e-03     1
    16   3.11e-03    2.31e-04   3.35e-03  -2.41e-03  -2.18e-03     1
    17   1.76e-03    1.30e-04   1.89e-03  -1.36e-03  -1.23e-03     1
    18   9.92e-04    7.36e-05   1.07e-03  -7.65e-04  -6.91e-04     1
```

[15]:
```
opt_result
```

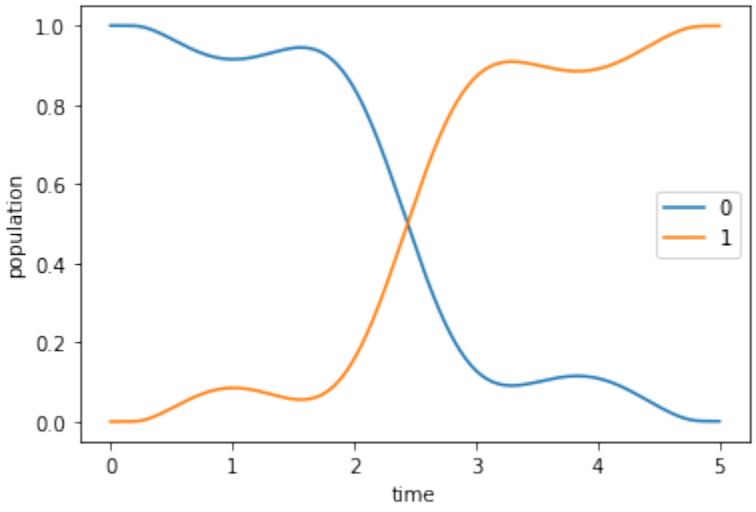
[15]:
```
Krotov Optimization Result
--------------------------
- Started at 2019-10-11 12:18:39
- Number of objectives: 1
- Number of iterations: 18
- Reason for termination: Reached convergence: J_T < 1e-3
- Ended at 2019-10-11 12:19:08
```

9.1.5 Simulate the dynamics under the optimized field

Having obtained the optimized control field, we can simulate the dynamics to verify that the optimized field indeed drives the initial state $|\Psi_{\text{init}}\rangle = |0\rangle$ to the desired target state $|\Psi_{\text{tgt}}\rangle = |1\rangle$.

[16]:
```
opt_dynamics = opt_result.optimized_objectives[0].mesolve(
    tlist, e_ops=[proj0, proj1])
```

[17]:
```
plot_population(opt_dynamics)
```



To gain some intuition on how the controls and the dynamics change throughout the optimization procedure, we can generate a plot of the control fields and the dynamics after each iteration of the optimization algorithm. This is possible because we set `store_all_pulses=True` in the call to `optimize_pulses`, which allows to recover the optimized controls from each iteration from `Result.all_pulses`. The flag is not set to True by default, as for long-running optimizations with thousands or tens of thousands iterations, the storage of all control fields may require significant memory.

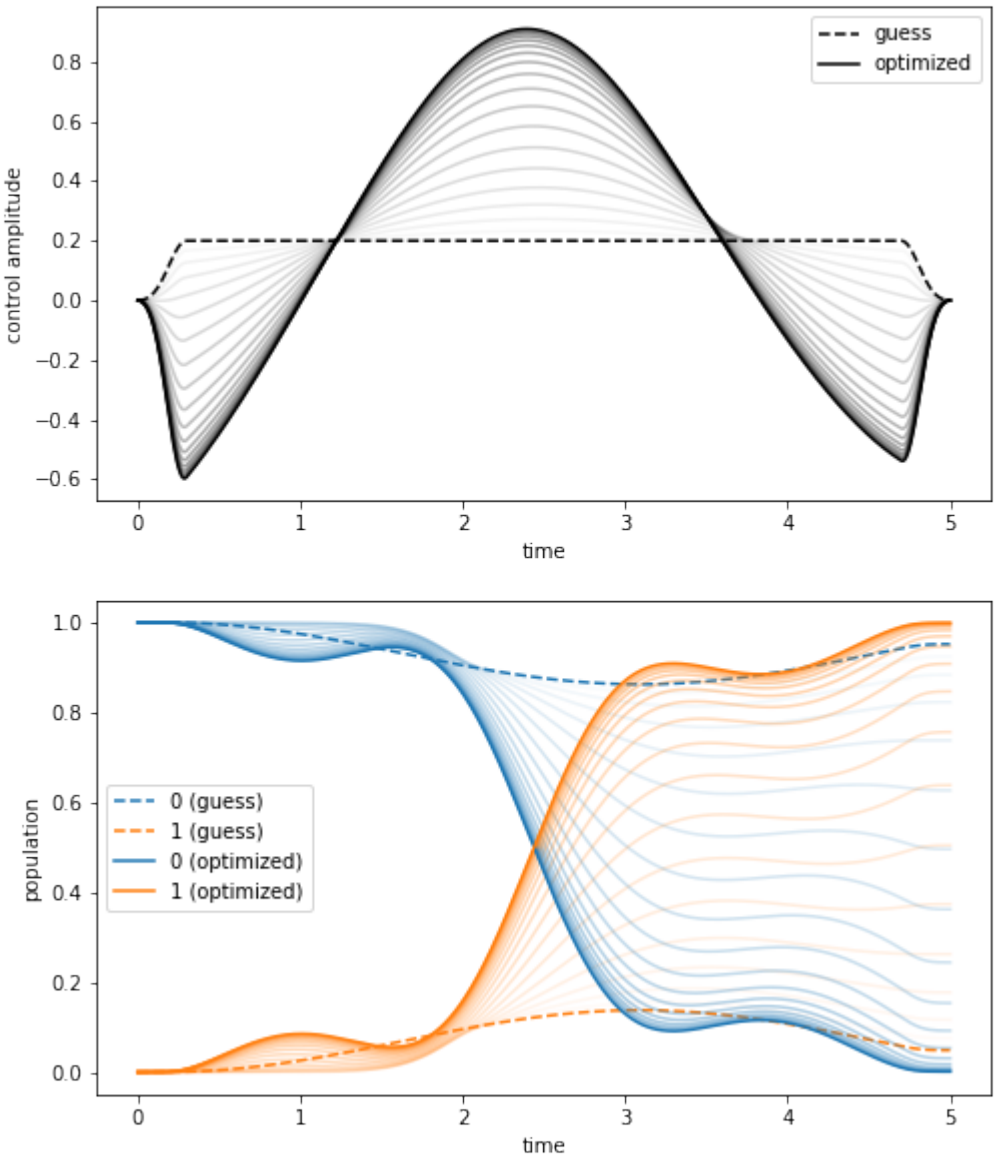
[18]:
```
def plot_iterations(opt_result):
    """Plot the control fields in population dynamics over all iterations.

    This depends on ``store_all_pulses=True`` in the call to
    `optimize_pulses`.
    """
    fig, [ax_ctr, ax_dyn] = plt.subplots(nrows=2, figsize=(8, 10))
    n_iters = len(opt_result.iters)
    for (iteration, pulses) in zip(opt_result.iters, opt_result.all_pulses):
        controls = [
            krotov.structural_conversions.pulse_onto_tlist(pulse)
            for pulse in pulses
        ]
        objectives = opt_result.objectives_with_controls(controls)
        dynamics = objectives[0].mesolve(
            opt_result.tlist, e_ops=[proj0, proj1]
        )
        if iteration == 0:
            ls = '--'  # dashed
            alpha = 1  # full opacity
            ctr_label = 'guess'
            pop_labels = ['0 (guess)', '1 (guess)']
        elif iteration == opt_result.iters[-1]:
            ls = '-'  # solid
            alpha = 1  # full opacity
            ctr_label = 'optimized'
            pop_labels = ['0 (optimized)', '1 (optimized)']
        else:
            ls = '-'  # solid
```

(continues on next page)

(continued from previous page)

```
            alpha = 0.5 * float(iteration) / float(n_iters)  # max 50%
            ctr_label = None
            pop_labels = [None, None]
        ax_ctr.plot(
            dynamics.times,
            controls[0],
            label=ctr_label,
            color='black',
            ls=ls,
            alpha=alpha,
        )
        ax_dyn.plot(
            dynamics.times,
            dynamics.expect[0],
            label=pop_labels[0],
            color='#1f77b4',  # default blue
            ls=ls,
            alpha=alpha,
        )
        ax_dyn.plot(
            dynamics.times,
            dynamics.expect[1],
            label=pop_labels[1],
            color='#ff7f0e',  # default orange
            ls=ls,
            alpha=alpha,
        )
    ax_dyn.legend()
    ax_dyn.set_xlabel('time')
    ax_dyn.set_ylabel('population')
    ax_ctr.legend()
    ax_ctr.set_xlabel('time')
    ax_ctr.set_ylabel('control amplitude')
    plt.show(fig)
```

[19]:
```
plot_iterations(opt_result)
```



The initial guess (dashed) and final optimized (solid) control amplitude and resulting dynamics are shown with full opacity, whereas the curves corresponding intermediate iterations are shown with decreasing transparency.

9.2 Optimization of a State-to-State Transfer in a Lambda System in the RWA

[1]:
```
# NBVAL_IGNORE_OUTPUT
%load_ext watermark
import os
import numpy as np
import scipy
```

(continues on next page)

(continued from previous page)

```
import matplotlib
import matplotlib.pylab as plt
import krotov
import qutip
from qutip import Qobj
%watermark -v --iversions
```

```
qutip            4.4.1
krotov           0.4.1
matplotlib.pylab 1.17.2
scipy            1.3.1
matplotlib       3.1.1
numpy            1.17.2
CPython 3.7.3
IPython 7.8.0
```

This example is illustrates the use of complex-valued control fields. This is accomplished by rewriting the Hamiltonian as the sum of two independent controls (real and imaginary parts). We consider a 3-level system in a Λ configuration, and seek control pulses that implement a (phase-sensitive) state-to-state transition $|1\rangle \rightarrow |3\rangle$.

9.2.1 The rotating wave Hamiltonian

The system consists of three levels $|1\rangle$, $|2\rangle$ and $|3\rangle$ with energy levels E_1, E_2 and E_3 which interact with a pair of laser pulses $\epsilon_P(t)$ ("pump laser") and $\epsilon_S(t)$ ("Stokes laser"), respectively, see Chapter 15.4.2 in "Introduction to Quantum Mechanics: A Time-Dependent Perspective" by David Tannor for details.

In the lab frame, the Hamiltonian reads

$$\hat{H}_{\text{lab}} = \begin{pmatrix} E_1 & -\mu_{12}\epsilon_P(t) & 0 \\ -\mu_{12}\epsilon_P(t) & E_2 & -\mu_{23}\epsilon_S(t) \\ 0 & -\mu_{23}\epsilon_S(t) & E_2 \end{pmatrix}.$$

with the dipole values μ_{12}, μ_{23} describing the coupling to the (real-valued) control fields $\epsilon_P(t)$, $\epsilon_S(t)$. The "rotating frame" is defined as

$$|\Psi_{\text{rot}}\rangle = \hat{U}_0^\dagger \, |\Psi_{\text{lab}}\rangle$$

with the transformation

$$\hat{U}_0 = |1\rangle\langle 1| e^{-i(E_2-\omega_P)t} + |2\rangle\langle 2| e^{-iE_2 t} + |3\rangle\langle 3| e^{-i(E_2-\omega_S)t},$$

where ω_P and ω_S are the two central frequencies defining the rotating frame.

The condition of having to fulfill the Schrödinger equation in the rotating frame implies a rotating frame Hamiltonian defined as

$$\hat{H}_{\text{rot}} = \hat{U}_0^\dagger \hat{H}_{\text{lab}} \hat{U}_0 - i\hat{U}_0^\dagger \dot{\hat{U}}_0 .$$

Note that most textbooks use $\hat{U}$ instead of $\hat{U}^\dagger$, and thus the adjoint of the above equation to define the rotating frame transformation, but we follow the example of Tannor's book here.

The rotating frame Hamiltonian reads

$$\hat{H}_{\text{rot}} = \begin{pmatrix} E_1 + \omega_P - E_2 & -\mu_{12}\epsilon_P(t)e^{-i\omega_P t} & 0 \\ -\mu_{12}\epsilon_P(t)e^{+i\omega_P t} & 0 & -\mu_{23}\epsilon_S(t)e^{-i\omega_S t} \\ 0 & -\mu_{23}\epsilon_S(t)e^{+i\omega_S t} & E3 + \omega_S - E_2 \end{pmatrix}.$$

We can now write the fields as

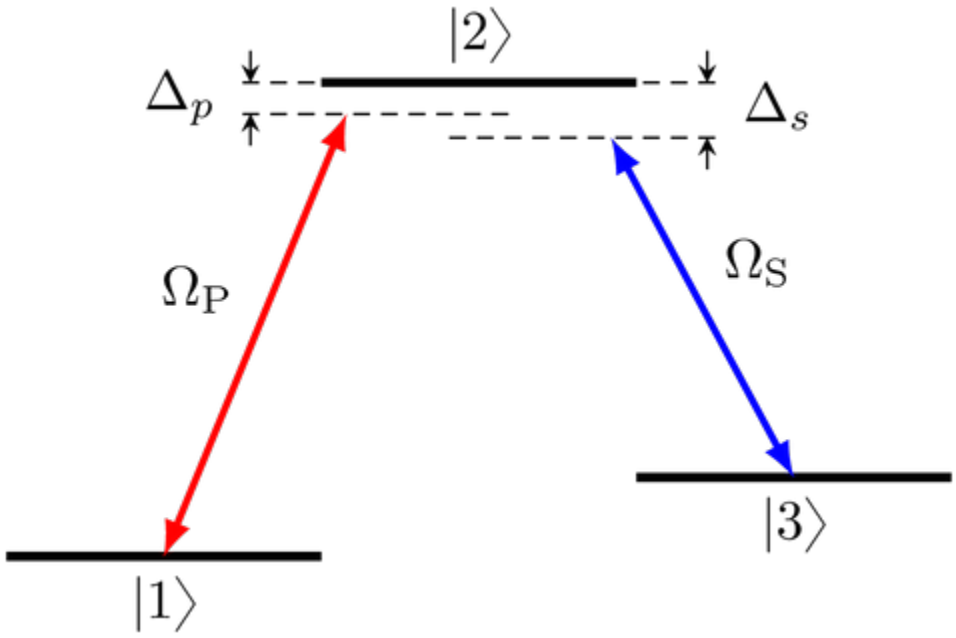
$$\begin{aligned}\mu_{12}\epsilon_P(t) &= \Omega_P^{(1)}(t)\cos(\omega_P t) - \Omega_P^{(2)}(t)\sin(\omega_P t)\\ &= \Omega_P^{(1)}(t)\left(e^{i\omega_P t} + e^{-i\omega_P t}\right) + i\Omega_P^{(2)}(t)\left(e^{i\omega_P t} - e^{-i\omega_P t}\right)\,,\end{aligned}$$

and similarly for $\epsilon_S(t)$, where we have split each field into two arbitrary (real-valued) auxiliary fields $\Omega_P^{(1)}(t), \Omega_P^{(2)}(t)$, and $\Omega_S^{(1)}(t), \Omega_S^{(2)}(t)$. This rewriting is suggestive of controls being spectrally centered around ω_P and ω_S, respectively, in which case any oscillations in $\Omega_{P,S}^{(1,2)}(t)$ are on a much slower time scale than $\omega_{P,S}$. Mathematically, however, *any* control fields can written in the above form. Thus, we have not placed any restriction on the controls at this time.

Plugging this into $\hat{H}_{\text{rot}}$ and invoking the rotating wave approximation that neglects all fast oscillating terms $\propto e^{\pm i2\omega_{P,S}t}$, we find

$$\hat{H}_{\text{RWA}} = \begin{pmatrix} \Delta_P & -\frac{1}{2}\Omega_P(t) & 0\\ -\frac{1}{2}\Omega_P^*(t) & 0 & -\frac{1}{2}\Omega_S(t)\\ 0 & -\frac{1}{2}\Omega_S^*(t) & \Delta_S \end{pmatrix}\,,$$

with the detunings $\Delta_P \equiv E_1 + \omega_P - E_2$, $\Delta_S \equiv E3 + \omega_S - E_2$ and the complex-valued control fields $\Omega_P(t) \equiv \Omega_P^{(1)}(t) + i\Omega_P^{(2)}(t)$ and $\Omega_S(t) \equiv \Omega_S^{(1)}(t) + i\Omega_S^{(2)}(t)$, illustrated in the following diagram:



Most textbooks (including Tannor's) only allow control fields of the form $\epsilon_{P,S}(t) \propto \Omega_{P,S}(t)\cos(\omega_{P,S}t)$ with the pulse envelopes $\Omega_{P,S}(t) \in \mathbb{R}^+$. This will result in the same $\hat{H}_{\text{RWA}}$ as above, but with the positive real-valued envelopes instead of the complex-valued $\Omega_{P,S}(t)$. However, this restriction is unnecessary: complex-valued control fields in the RWA are more general and entirely physical, with the relation to the real-valued field in the lab frame as defined above. The spectra of the optimized pulses are free to deviate from the frequencies of the rotating frame, limited only by the numerical resolution of the time grid and the RWA.

The `krotov` package requires that all control pulses are real-valued. Therefore, the real and imaginary parts of Ω_P and Ω_S are treated as independent Hamiltonians, and we write

$$\hat{H}_{\text{RWA}} = \hat{H}_0 + \Omega_P^{(1)}(t)\hat{H}_{P,\text{re}} + \Omega_P^{(2)}(t)\hat{H}_{P,\text{im}} + \Omega_S^{(1)}(t)\hat{H}_{S,\text{re}} + \Omega_S^{(2)}(t)\hat{H}_{S,\text{im}}$$

for the purpose of the optimization, with

$$\hat{H}_0 = \Delta_P |1\rangle\langle 1| + \Delta_S |3\rangle\langle 3| \,,$$
$$\hat{H}_{P,\mathrm{re}} = -\frac{1}{2}\left(|1\rangle\langle 2| + |2\rangle\langle 1|\right) \,,$$
$$\hat{H}_{P,\mathrm{im}} = -\frac{i}{2}\left(|1\rangle\langle 2| - |2\rangle\langle 1|\right) \,,$$
$$\hat{H}_{S,\mathrm{re}} = -\frac{1}{2}\left(|2\rangle\langle 3| + |3\rangle\langle 2|\right) \,,$$
$$\hat{H}_{S,\mathrm{im}} = -\frac{i}{2}\left(|2\rangle\langle 3| - |3\rangle\langle 2|\right) \,.$$

9.2.2 Guess controls

We choose the initial guess for the four control fields based on the intuition of the "stimulated Raman adiabatic passage" (STIRAP) scheme. STIRAP allows to transfer the population in $|1\rangle$ $|3\rangle$ without having to pass through $|2\rangle$; it requires the Stokes-pulse to precede but overlap the pump-pulse.

Here, we leave it up to Krotov's method to find appropriate pulses for a STIRAP-like transfer (without requiring that the $|2\rangle$ level remains unpopulated). We start from a low intensity real-valued $\Omega_S(t)$ pulse with a Blackman shape, followed by an overlapping real-valued $\Omega_P(t)$ of the same shape. The entire scheme is in the time interval [0, 5].

[2]:
```
def Omega_P1(t, args):
    """Guess for the real part of the pump pulse"""
    Ω0 = 5.0
    return Ω0 * krotov.shapes.blackman(t, t_start=2.0, t_stop=5.0)


def Omega_P2(t, args):
    """Guess for the imaginary part of the pump pulse"""
    return 0.0


def Omega_S1(t, args):
    """Guess for the real part of the Stokes pulse"""
    Ω0 = 5.0
    return Ω0 * krotov.shapes.blackman(t, t_start=0.0, t_stop=3.0)


def Omega_S2(t, args):
    """Guess for the imaginary part of the Stokes pulse"""
    return 0.0
```

We can now instantiate the Hamiltonian including these guess controls:

[3]:
```
def hamiltonian(E1=0.0, E2=10.0, E3=5.0, omega_P=9.5, omega_S=4.5):
    """Lambda-system Hamiltonian in the RWA"""

    # detunings
    ΔP = E1 + omega_P - E2
    ΔS = E3 + omega_S - E2

    H0 = Qobj([[ΔP, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, ΔS]])
```

(continues on next page)

(continued from previous page)

```
    HP_re = -0.5 * Qobj([[0.0, 1.0, 0.0], [1.0, 0.0, 0.0], [0.0, 0.0, 0.0]])
    HP_im = -0.5 * Qobj([[0.0, 1.0j, 0.0], [-1.0j, 0.0, 0.0], [0.0, 0.0, 0.0]])

    HS_re = -0.5 * Qobj([[0.0, 0.0, 0.0], [0.0, 0.0, 1.0], [0.0, 1.0, 0.0]])
    HS_im = -0.5 * Qobj([[0.0, 0.0, 0.0], [0.0, 0.0, 1.0j], [0.0, -1.0j, 0.0]])

    return [
        H0,
        [HP_re, Omega_P1],
        [HP_im, Omega_P2],
        [HS_re, Omega_S1],
        [HS_im, Omega_S2],
    ]
```

[4]:
```
H = hamiltonian()
```

9.2.3 Target state in the rotating frame

The basis states of the Λ-system are defined as

[5]:
```
ket1 = qutip.Qobj(np.array([1.0, 0.0, 0.0]))
ket2 = qutip.Qobj(np.array([0.0, 1.0, 0.0]))
ket3 = qutip.Qobj(np.array([0.0, 0.0, 1.0]))
```

We would like to implement a phase-sensitive transition $|1\rangle \rightarrow |3\rangle$ *in the lab frame*. Since we are defining the dynamics in the RWA, this means we have to adjust the target state to be in the rotating frame as well (the initial state at $t = 0$ is not affected by the RWA).

As defined earlier, the states in the rotating frame are obtained from the states in the lab frame by the transformation $|\Psi_{\mathrm{rot}}\rangle = \hat{U}_0^\dagger |\Psi_{\mathrm{lab}}\rangle$. In our case, this means that we get $|3\rangle$ with and additional phase factor:

[6]:
```
def rwa_target_state(ket3, E2=10.0, omega_S=4.5, T=5):
    return np.exp(1j * (E2 - omega_S) * T) * ket3
```

[7]:
```
psi_target = rwa_target_state(ket3)
```

We can now instantiate the control objective:

[8]:
```
objective = krotov.Objective(initial_state=ket1, target=psi_target, H=H)
objective
```

[8]:
```
Objective[|Ψ₀(3)⟩ to |Ψ₁(3)⟩ via [H₀[3,3], [H₁[3,3], u₁(t)], [H₂[3,3], u₂(t)], [H₃[3,3],
↪ u₃(t)], [H₄[3,3], u₄(t)]]]
```

9.2.4 Simulate dynamics under the guess field

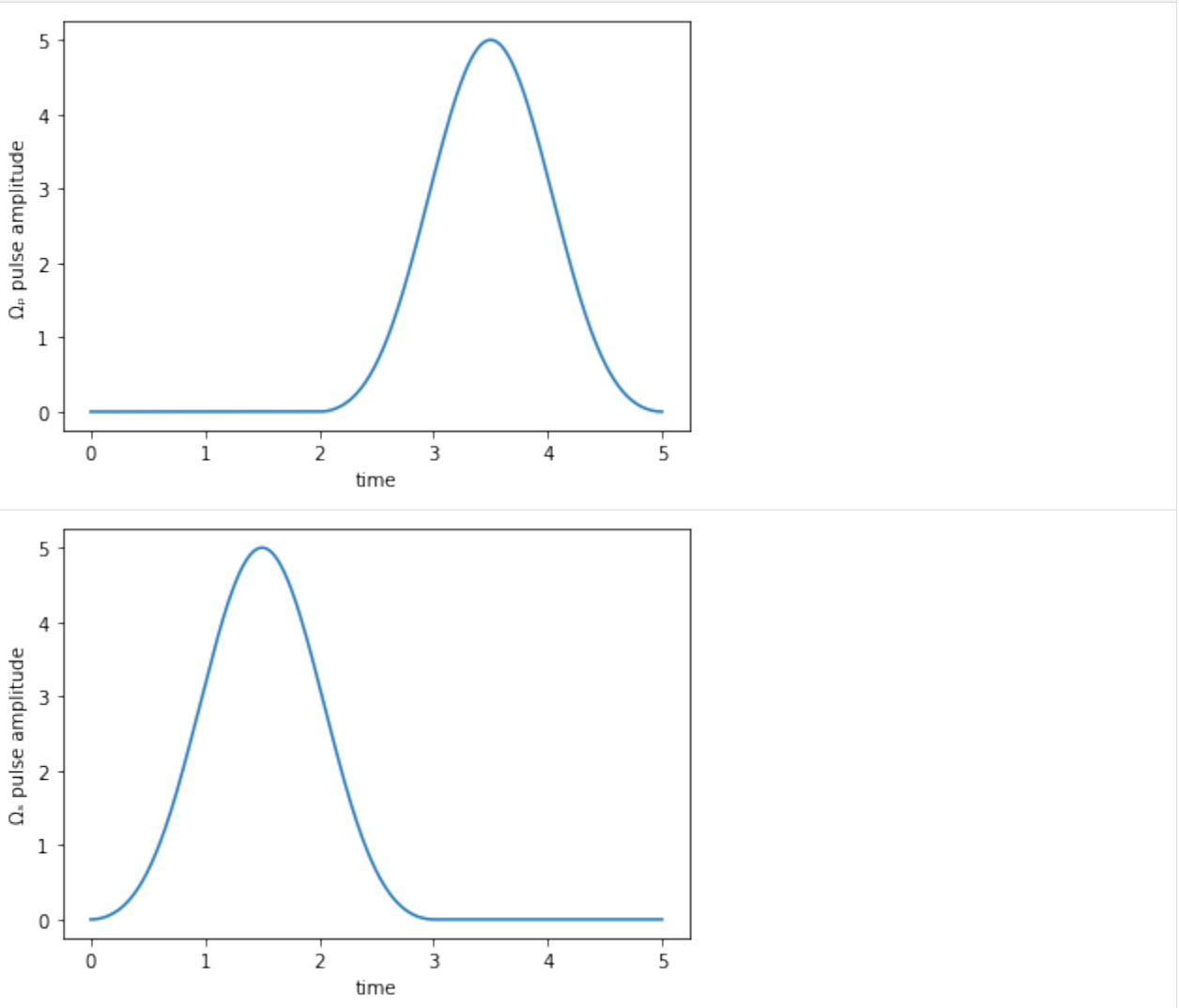
We use a time grid with 500 steps between $t = 0$ and $T = 5$:

[9]:
```
tlist = np.linspace(0, 5, 500)
```

Before propagating, we visually verify the guess pulses we defined earlier:

[10]:
```
def plot_pulse(pulse, tlist, label):
    fig, ax = plt.subplots()
    if callable(pulse):
        pulse = np.array([pulse(t, args=None) for t in tlist])
    ax.plot(tlist, pulse)
    ax.set_xlabel('time')
    ax.set_ylabel('%s pulse amplitude' % label)
    plt.show(fig)
```

[11]:
```
plot_pulse(H[1][1], tlist, 'Ωₚ')
plot_pulse(H[3][1], tlist, 'Ωₛ')
```



The imaginary parts are zero:

[12]:
```
assert np.all([H[2][1](t, None) == 0 for t in tlist])
assert np.all([H[4][1](t, None) == 0 for t in tlist])
```

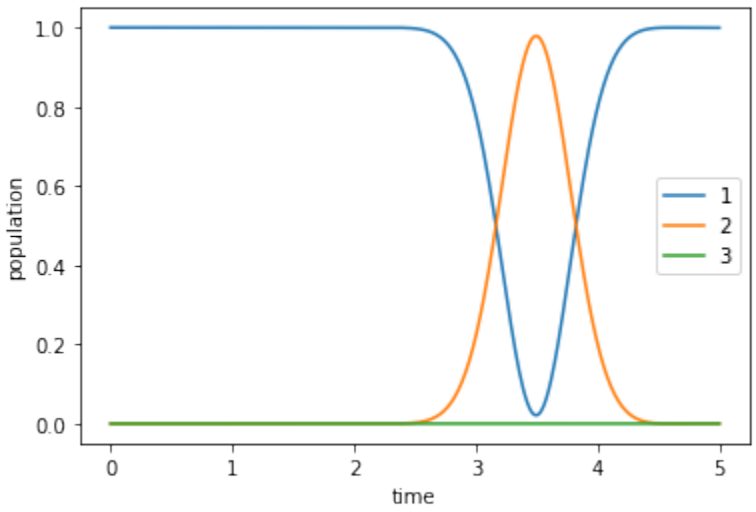
We introduce projectors $\hat{P}_i = |i\rangle\langle i|$ for each of the three energy levels, allowing use to plot the population dynamics:

[13]:
```
proj1 = qutip.ket2dm(ket1)
proj2 = qutip.ket2dm(ket2)
proj3 = qutip.ket2dm(ket3)
```

[14]:
```
guess_dynamics = objective.mesolve(tlist, e_ops=[proj1,proj2,proj3])
```

[15]:
```
def plot_population(result):
    fig, ax = plt.subplots()
    ax.plot(result.times, result.expect[0], label='1')
    ax.plot(result.times, result.expect[1], label='2')
    ax.plot(result.times, result.expect[2], label='3')
    ax.legend()
    ax.set_xlabel('time')
    ax.set_ylabel('population')
    plt.show(fig)
```

[16]:
```
plot_population(guess_dynamics)
```



We find that our guess pulses are too disjoint to implement the STIRAP scheme. Thus, the Stokes pulse has no effect, whilst the pump pulse merely transfers population out of $|1\rangle$ into $|2\rangle$ and back again.

9.2.5 Optimize

In order to invoke `optimize_pulses`, we must define the required parameters for each control, a pulse shape (used to ensure that the controls remain 0 at $t = 0$ and $t = T$), and the parameter λ_a that determines the overall magnitude of the pulse updates in each iteration.

[17]:
```
def S(t):
    """Scales the Krotov methods update of the pulse value at the time t"""
    return krotov.shapes.flattop(
        t, t_start=0.0, t_stop=5.0, t_rise=0.3, func='sinsq'
    )
```

[18]:
```
pulse_options = {
    H[1][1]: dict(lambda_a=0.5, update_shape=S),
    H[2][1]: dict(lambda_a=0.5, update_shape=S),
    H[3][1]: dict(lambda_a=0.5, update_shape=S),
    H[4][1]: dict(lambda_a=0.5, update_shape=S)
}
```

We now run the optimization, using the phase-sensitive functional $J_{T,\mathrm{re}} = 1 - \mathrm{Re}\left\langle \Psi(t) \mid \Psi_{\mathrm{tgt}} \right\rangle$, printing the integrated pulse update for each control in each iteration. The optimization stops when J_T falls below 10^{-3}, changes by less than 10^{-5}, or after at most 15 iterations. We also check for monotonic convergence.

[19]:
```
opt_result = krotov.optimize_pulses(
    [objective],
    pulse_options,
    tlist,
    propagator=krotov.propagators.expm,
    chi_constructor=krotov.functionals.chis_re,
    info_hook=krotov.info_hooks.print_table(
        J_T=krotov.functionals.J_T_re,
        show_g_a_int_per_pulse=True,
        unicode=False,
    ),
    check_convergence=krotov.convergence.Or(
        krotov.convergence.value_below(1e-3, name='J_T'),
        krotov.convergence.delta_below(1e-5),
        krotov.convergence.check_monotonic_error,
    ),
    iter_stop=15,
)
```

```
  iter.        J_T   g_a_int_1   g_a_int_2   g_a_int_3   g_a_int_4     g_a_int
↪   J  Delta_J_T    Delta J  secs
     0   1.01e+00    0.00e+00    0.00e+00    0.00e+00    0.00e+00    0.00e+00   1.
↪01e+00        n/a         n/a     1
     1   6.72e-01    1.72e-01    5.72e-04    1.63e-01    7.44e-04    3.37e-01   1.
↪01e+00  -3.37e-01  -2.83e-05     2
     2   4.02e-01    1.44e-01    8.41e-04    1.24e-01    8.40e-04    2.70e-01   6.
↪72e-01  -2.70e-01  -3.32e-05     2
     3   2.22e-01    9.81e-02    9.26e-04    7.98e-02    7.75e-04    1.80e-01   4.
↪02e-01  -1.80e-01  -3.55e-05     2
     4   1.17e-01    5.78e-02    7.70e-04    4.58e-02    6.02e-04    1.05e-01   2.
↪22e-01  -1.05e-01  -3.11e-05     2
     5   6.00e-02    3.13e-02    5.35e-04    2.46e-02    4.20e-04    5.68e-02   1.
↪17e-01  -5.69e-02  -2.30e-05     2
     6   3.05e-02    1.62e-02    3.40e-04    1.27e-02    2.78e-04    2.95e-02   6.
↪00e-02  -2.95e-02  -1.51e-05     2
     7   1.54e-02    8.16e-03    2.11e-04    6.47e-03    1.82e-04    1.50e-02   3.
↪05e-02  -1.50e-02  -9.25e-06     2
     8   7.85e-03    4.08e-03    1.33e-04    3.25e-03    1.20e-04    7.59e-03   1.
↪54e-02  -7.59e-03  -5.45e-06     2
     9   4.03e-03    2.03e-03    8.59e-05    1.63e-03    8.01e-05    3.83e-03   7.
↪85e-03  -3.83e-03  -3.15e-06     2
    10   2.09e-03    1.01e-03    5.76e-05    8.13e-04    5.45e-05    1.94e-03   4.
↪02e-03  -1.94e-03  -1.81e-06     2
    11   1.10e-03    5.03e-04    3.97e-05    4.06e-04    3.76e-05    9.87e-04   2.
↪09e-03  -9.88e-04  -1.04e-06     2
```

(continues on next page)

(continued from previous page)

```
     12   5.91e-04    2.51e-04    2.79e-05    2.03e-04    2.62e-05    5.09e-04   1.
↪10e-03  -5.09e-04  -6.04e-07     2
```

[20]:
```
opt_result
```

[20]:
```
Krotov Optimization Result
--------------------------
- Started at 2019-10-11 12:19:20
- Number of objectives: 1
- Number of iterations: 12
- Reason for termination: Reached convergence: J_T < 0.001
- Ended at 2019-10-11 12:19:51
```

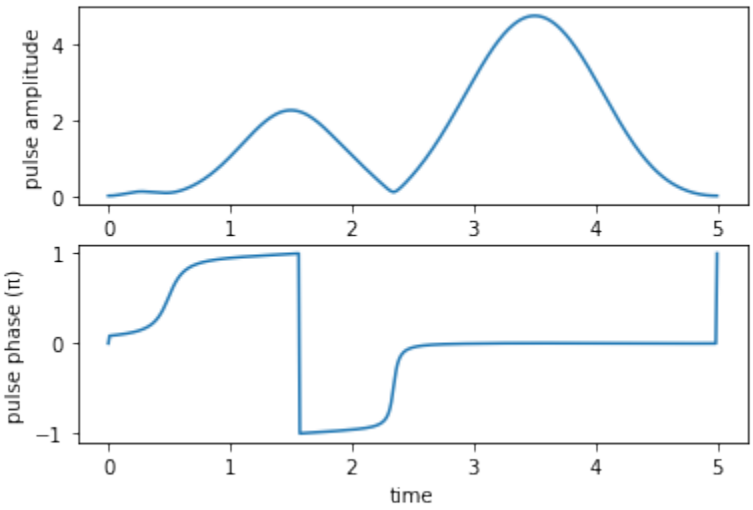
We dump the result of the optimization to disk for later use in the *Ensemble Optimization for Robust Pulses*.

[21]:
```
if not os.path.isfile('lambda_rwa_opt_result.dump'):
    opt_result.dump('lambda_rwa_opt_result.dump')
```

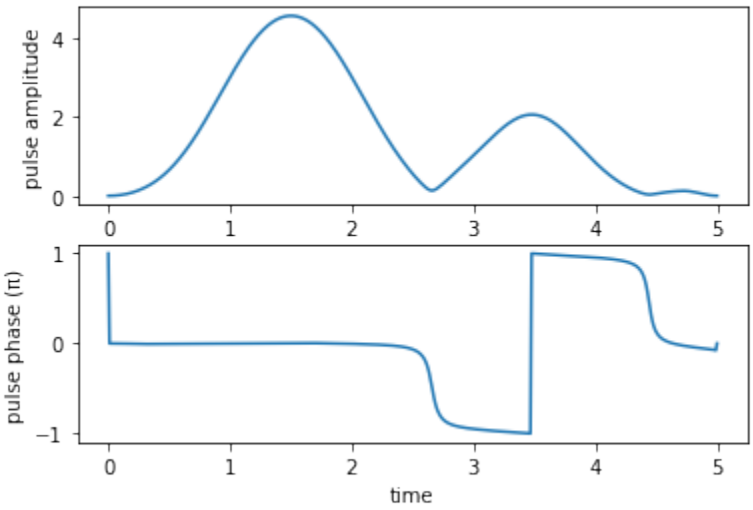
The optimized complex pulses look as follows:

[22]:
```
def plot_pulse_amplitude_and_phase(pulse_real, pulse_imaginary,tlist):
    ax1 = plt.subplot(211)
    ax2 = plt.subplot(212)
    amplitudes = [np.sqrt(x*x + y*y) for x,y in zip(pulse_real,pulse_imaginary)]
    phases = [np.arctan2(y,x)/np.pi for x,y in zip(pulse_real,pulse_imaginary)]
    ax1.plot(tlist,amplitudes)
    ax1.set_xlabel('time')
    ax1.set_ylabel('pulse amplitude')
    ax2.plot(tlist,phases)
    ax2.set_xlabel('time')
    ax2.set_ylabel('pulse phase (π)')
    plt.show()

print("pump pulse amplitude and phase:")
plot_pulse_amplitude_and_phase(
    opt_result.optimized_controls[0], opt_result.optimized_controls[1], tlist)
print("Stokes pulse amplitude and phase:")
plot_pulse_amplitude_and_phase(
    opt_result.optimized_controls[2], opt_result.optimized_controls[3], tlist)
```

```
pump pulse amplitude and phase:
```



Stokes pulse amplitude and phase:



We can convert the complex controls in the rotating frame back into the real-valued pulses in the lab frame:

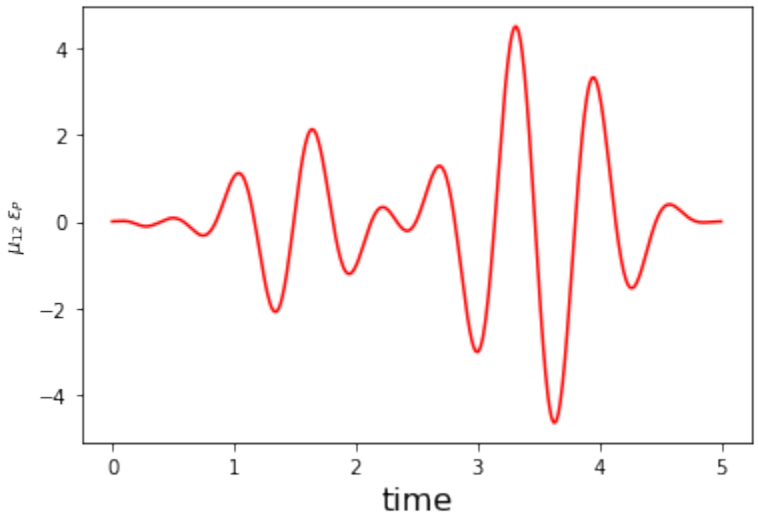
[23]:
```
def plot_physical_field(pulse_re, pulse_im, tlist, case=None):

    if case == 'pump':
        w = 9.5
    elif case == 'stokes':
        w = 4.5
    else:
        print('Error: selected case is not a valid option')
        return

    ax = plt.subplot(111)
    ax.plot(tlist,pulse_re*np.cos(w*tlist)-pulse_im*np.sin(w*tlist), 'r')
    ax.set_xlabel('time', fontsize = 16)
    if case == 'pump':
        ax.set_ylabel(r'$\mu_{12}\,\epsilon_{P}$')
```

(continues on next page)

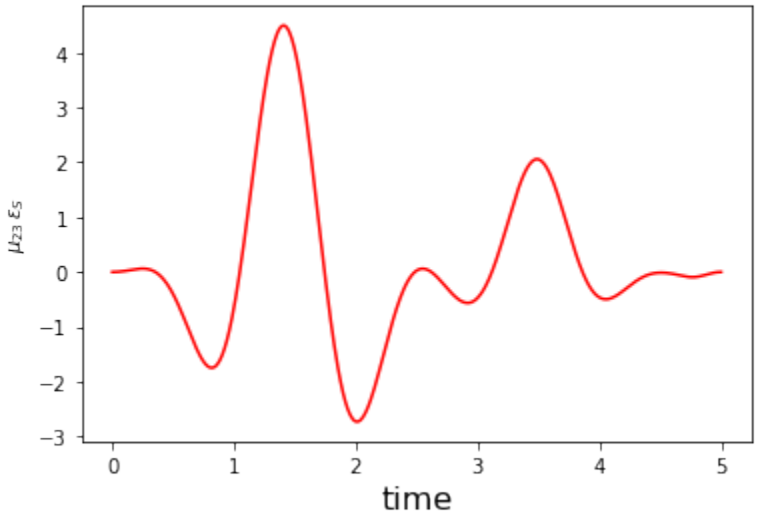
(continued from previous page)

```
    elif case == 'stokes':
        ax.set_ylabel(r'$ \mu_{23}\,\epsilon_{S}$')
    plt.show()

print('Physical electric pump pulse in the lab frame:')
plot_physical_field(
    opt_result.optimized_controls[0], opt_result.optimized_controls[1], tlist, case =
↪'pump')


print('Physical electric Stokes pulse in the lab frame:')
plot_physical_field(
    opt_result.optimized_controls[2], opt_result.optimized_controls[3], tlist, case =
↪'stokes')
```

```
Physical electric pump pulse in the lab frame:
```




```
Physical electric Stokes pulse in the lab frame:
```

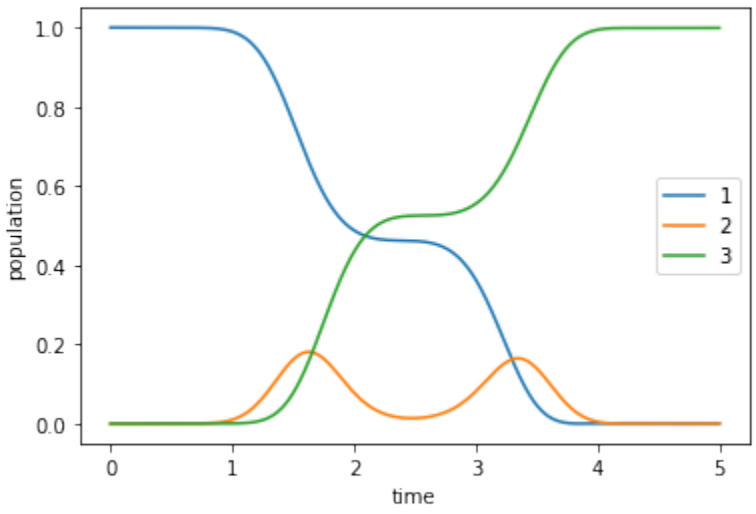


Lastly, we check the population dynamics to verify that we indeed implement the desired

state-to-state transfer:

[24]:
```
opt_dynamics = opt_result.optimized_objectives[0].mesolve(
    tlist, e_ops=[proj1, proj2, proj3])
```

[25]:
```
plot_population(opt_dynamics)
```



9.3 Optimization of a Dissipative State-to-State Transfer in a Lambda System

[1]:
```
# NBVAL_IGNORE_OUTPUT
%load_ext watermark
import os
import qutip
import numpy as np
import scipy
import matplotlib
import matplotlib.pylab as plt
import krotov
import qutip
from qutip import Qobj
import pickle
%watermark -v --iversions
```

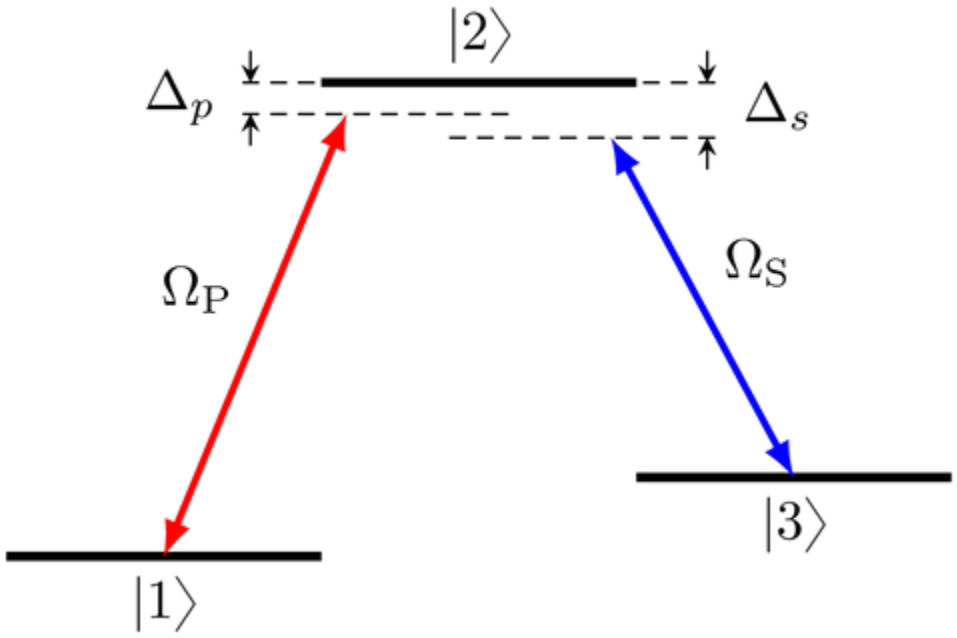
```
krotov           0.4.1
matplotlib.pylab 1.17.2
qutip            4.4.1
numpy            1.17.2
scipy            1.3.1
matplotlib       3.1.1
CPython 3.7.3
IPython 7.8.0
```

This example illustrates the use of Krotov's method with a non-Hermitian Hamiltonian. It considers the same system as the *previous example*, a transition $|1\rangle \rightarrow |3\rangle$ in a three-level system in a Λ-configuration. However, here we add a non-Hermitian decay term to model loss

from the intermediary level $|2\rangle$.

9.3.1 The effective Hamiltonian

We consider the system as in the following diagram:



with the Hamiltonian

$$\hat{H}_{\text{lab}} = \begin{pmatrix} E_1 & -\mu_{12}\epsilon_P(t) & 0 \\ -\mu_{12}\epsilon_P(t) & E_2 & -\mu_{23}\epsilon_S(t) \\ 0 & -\mu_{23}\epsilon_S(t) & E_2 \end{pmatrix}$$

in the lab frame.

However, we now also include that the level $|2\rangle$ decays incoherently. This is the primary motivation of the STIRAP scheme: through destructive interference it can keep the dynamics in a "dark state" where the population is transferred from $|1\rangle$ to $|3\rangle$ without ever populating the $|2\rangle$ state. A rigorous treatment would be to include the dissipation as a Lindblad operator, and to simulate the dynamics and perform the optimization in Liouville space. The Lindblad operator for spontaneous decay from level $|2\rangle$ with decay rate 2γ is $\hat{L} = \sqrt{2\gamma}|1\rangle\langle 2|$. However, this is numerically expensive. For the optimization, it is sufficient to find a way to penalize population in the $|2\rangle$ state.

Motivated by the Monte-Carlo Wave Function (MCWF) method, we define the non-Hermitian *effective Hamiltonian*

$$\hat{H}_{\text{eff}} = \hat{H}_{\text{lab}} - \frac{i}{2}\hat{L}^\dagger\hat{L}$$

In explicit form, this is

$$\hat{H}_{\text{eff}} = \begin{pmatrix} E_1 & -\mu_{12}\epsilon_P(t) & 0 \\ -\mu_{12}\epsilon_P(t) & E_2 - i\gamma & -\mu_{23}\epsilon_S(t) \\ 0 & -\mu_{23}\epsilon_S(t) & E_2 \end{pmatrix}$$

The only change is that the energy of level $|2\rangle$ now has an imaginary part $-\gamma$, which causes an exponential decay of any population amplitude in $|2\rangle$, and thus a decay in the norm of the state. In the MCWF, this decay of the norm is used to track the probability that quantum jump occurs (otherwise, the state is re-normalized). Here, we do not perform quantum jumps or renormalize the state. Instead, we use the decay in the norm to steer the optimization. Using the functional

$$J_{T,\mathrm{re}} = 1 - \mathrm{Re}\left\langle \Psi(T) \mid \Psi^{\mathrm{tgt}} \right\rangle$$

to be minimized, we find that the value of the functional increases if $\||\Psi(T)\rangle\| < 1$. Thus, population in $|2\rangle$ is penalized, without any significant numerical overhead.

The decay rate 2γ does not necessarily need to correspond to the actual physical lifetime of the $|2\rangle$ state: we can choose an artificially high decay rate to put a stronger penalty on the $|2\rangle$ level. Or, if the physical decay is so strong that the norm of the state reaches effectively zero, we could decrease γ to avoid numerical instability. The use of a non-Hermitian Hamiltonian with artificial decay is generally a useful trick to penalize population in a subspace.

The new non-Hermitian decay term remains unchanged when we make the rotating wave approximation. The RWA Hamiltonian now reads

$$\hat{H}_{\mathrm{RWA}} = \begin{pmatrix} \Delta_P & -\frac{1}{2}\Omega_P(t) & 0 \\ -\frac{1}{2}\Omega_P^*(t) & -i\gamma & -\frac{1}{2}\Omega_S(t) \\ 0 & -\frac{1}{2}\Omega_S^*(t) & \Delta_S \end{pmatrix},$$

with complex control fields $\Omega_P(t)$ and $\Omega_S(t)$, see the *previous example*. Again, we split these complex pulses into an independent real and imaginary part for the purpose of optimization.

The guess controls are

[2]:
```
def Omega_P1(t, args):
    """Guess for the real part of the pump pulse"""
    Ω0 = 5.0
    return Ω0 * krotov.shapes.blackman(t, t_start=2.0, t_stop=5.0)


def Omega_P2(t, args):
    """Guess for the imaginary part of the pump pulse"""
    return 0.0


def Omega_S1(t, args):
    """Guess for the real part of the Stokes pulse"""
    Ω0 = 5.0
    return Ω0 * krotov.shapes.blackman(t, t_start=0.0, t_stop=3.0)


def Omega_S2(t, args):
    """Guess for the imaginary part of the Stokes pulse"""
    return 0.0
```

and the Hamiltonian is instantiated as

[3]:
```
def hamiltonian(E1=0.0, E2=10.0, E3=5.0, omega_P=9.5, omega_S=4.5, gamma=0.5):
    """Lambda-system Hamiltonian in the RWA"""

    # detunings
    ΔP = E1 + omega_P - E2
```

(continues on next page)

(continued from previous page)

```
    ΔS = E3 + omega_S - E2

    H0 = Qobj([[ΔP, 0.0, 0.0], [0.0, -1j * gamma, 0.0], [0.0, 0.0, ΔS]])

    HP_re = -0.5 * Qobj([[0.0, 1.0, 0.0], [1.0, 0.0, 0.0], [0.0, 0.0, 0.0]])
    HP_im = -0.5 * Qobj([[0.0, 1.0j, 0.0], [-1.0j, 0.0, 0.0], [0.0, 0.0, 0.0]])

    HS_re = -0.5 * Qobj([[0.0, 0.0, 0.0], [0.0, 0.0, 1.0], [0.0, 1.0, 0.0]])
    HS_im = -0.5 * Qobj([[0.0, 0.0, 0.0], [0.0, 0.0, 1.0j], [0.0, -1.0j, 0.0]])

    return [
        H0,
        [HP_re, Omega_P1],
        [HP_im, Omega_P2],
        [HS_re, Omega_S1],
        [HS_im, Omega_S2],
    ]
```

[4]:
```
H = hamiltonian()
```

We check the hermiticity of the Hamiltonian:

[5]:
```
print("H0 is Hermitian: " + str(H[0].isherm))
print("H1 is Hermitian: "+ str(
        H[1][0].isherm
    and H[2][0].isherm
    and H[3][0].isherm
    and H[4][0].isherm))
```

```
H0 is Hermitian: False
H1 is Hermitian: True
```

9.3.2 Define the optimization target

We optimize for the phase-sensitive transition $|1\rangle \rightarrow |3\rangle$. As we are working in the rotating frame, the target state must be adjusted with an appropriate phase factor:

[6]:
```
ket1 = qutip.Qobj(np.array([1.0, 0.0, 0.0]))
ket2 = qutip.Qobj(np.array([0.0, 1.0, 0.0]))
ket3 = qutip.Qobj(np.array([0.0, 0.0, 1.0]))

def rwa_target_state(ket3, E2=10.0, omega_S=4.5, T=5):
    return np.exp(1j * (E2 - omega_S) * T) * ket3

psi_target = rwa_target_state(ket3)
```

The objective is now instantiated as

[7]:
```
objectives = [krotov.Objective(initial_state=ket1, target=psi_target, H=H)]
objectives
```

[7]:
```
[Objective[|Ψ₀(3)⟩ to |Ψ₁(3)⟩ via [A₀[3,3], [H₁[3,3], u₁(t)], [H₂[3,3], u₂(t)], [H₃[3,
↪3], u₃(t)], [H₄[3,3], u₄(t)]]]]
```

9.3.3 Simulate dynamics under the guess field

We use a time grid with 500 steps between $t = 0$ and $T = 5$:

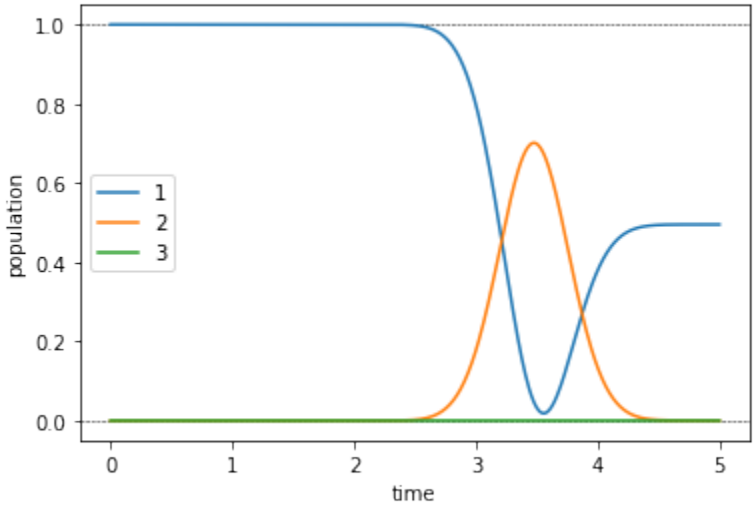
[8]:
```
tlist = np.linspace(0, 5, 500)
```

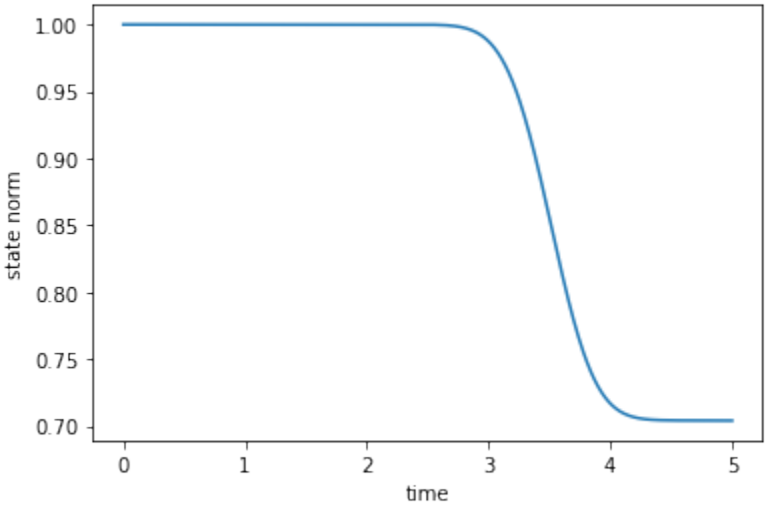
We propagate once for the population dynamics, and once to obtain the propagated states for each point on the time grid:

[9]:
```
proj1 = qutip.ket2dm(ket1)
proj2 = qutip.ket2dm(ket2)
proj3 = qutip.ket2dm(ket3)

guess_dynamics = objectives[0].propagate(
    tlist, propagator=krotov.propagators.expm, e_ops=[proj1, proj2, proj3]
)
guess_states = objectives[0].propagate(
    tlist, propagator=krotov.propagators.expm
)
```

[10]:
```
def plot_population(result):
    fig, ax = plt.subplots()
    ax.axhline(y=1.0, color='black', lw=0.5, ls='dashed')
    ax.axhline(y=0.0, color='black', lw=0.5, ls='dashed')
    ax.plot(result.times, result.expect[0], label='1')
    ax.plot(result.times, result.expect[1], label='2')
    ax.plot(result.times, result.expect[2], label='3')
    ax.legend()
    ax.set_xlabel('time')
    ax.set_ylabel('population')
    plt.show(fig)


def plot_norm(result):

    state_norm = lambda i: result.states[i].norm()
    states_norm=np.vectorize(state_norm)

    fig, ax = plt.subplots()
    ax.plot(result.times, states_norm(np.arange(len(result.states))))
    ax.set_xlabel('time')
    ax.set_ylabel('state norm')
    plt.show(fig)
```

[11]:
```
plot_population(guess_dynamics)
plot_norm(guess_states)
```





The population dynamics and the norm-plot show the effect the non-Hermitian term in the Hamiltonian, resulting in a 30% loss.

9.3.4 Optimize

For each control, we define the update shape and the λ_a parameter that determines the magnitude of the update:

[12]:
```
def S(t):
    """Scales the Krotov methods update of the pulse value at the time t"""
    return krotov.shapes.flattop(
        t, t_start=0.0, t_stop=5.0, t_rise=0.3, func='sinsq'
    )
```

[13]:
```
pulse_options = {
    H[1][1]: dict(lambda_a=2.0, update_shape=S),
    H[2][1]: dict(lambda_a=2.0, update_shape=S),
    H[3][1]: dict(lambda_a=2.0, update_shape=S),
```

(continues on next page)

(continued from previous page)

```
    H[4][1]: dict(lambda_a=2.0, update_shape=S)
}
```

We now run the optimization for 40 iterations, printing out the fidelity

$$F_{\mathrm{re}} = \mathrm{Re}\left\langle \Psi(T) \mid \Psi^{\mathrm{tgt}} \right\rangle$$

after each iteration.

[14]:
```
def print_fidelity(**args):
    F_re = np.average(np.array(args['tau_vals']).real)
    print("   F = %f" % F_re)
    return F_re
```

[15]:
```
opt_result = krotov.optimize_pulses(
    objectives, pulse_options, tlist,
    propagator=krotov.propagators.expm,
    chi_constructor=krotov.functionals.chis_re,
    info_hook=print_fidelity,
    iter_stop=40
)
```

```
   F = -0.007812
   F = 0.055166
   F = 0.117604
   F = 0.178902
   F = 0.238507
   F = 0.295926
   F = 0.350749
   F = 0.402648
   F = 0.451388
   F = 0.496822
   F = 0.538882
   F = 0.577573
   F = 0.612961
   F = 0.645161
   F = 0.674324
   F = 0.700629
   F = 0.724268
   F = 0.745445
   F = 0.764364
   F = 0.781226
   F = 0.796224
   F = 0.809541
   F = 0.821349
   F = 0.831809
   F = 0.841064
   F = 0.849250
   F = 0.856486
   F = 0.862881
   F = 0.868532
   F = 0.873527
   F = 0.877942
   F = 0.881847
   F = 0.885302
   F = 0.888362
   F = 0.891074
```

(continues on next page)

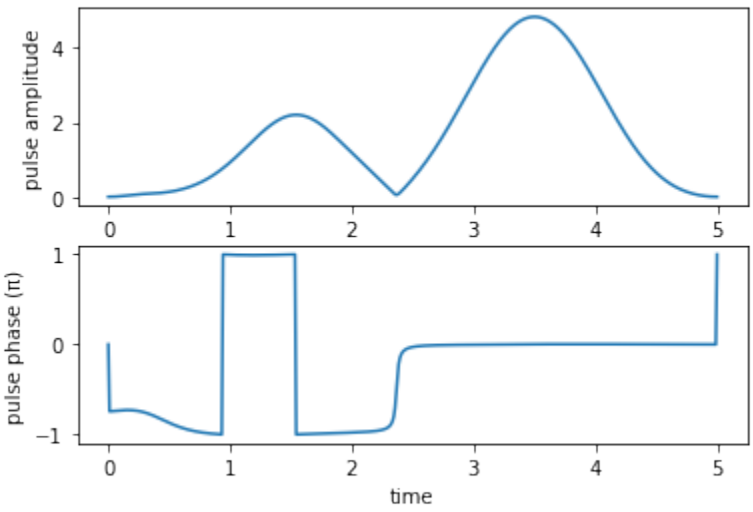
(continued from previous page)

```
   F = 0.893481
   F = 0.895618
   F = 0.897519
   F = 0.899211
   F = 0.900721
   F = 0.902071
```

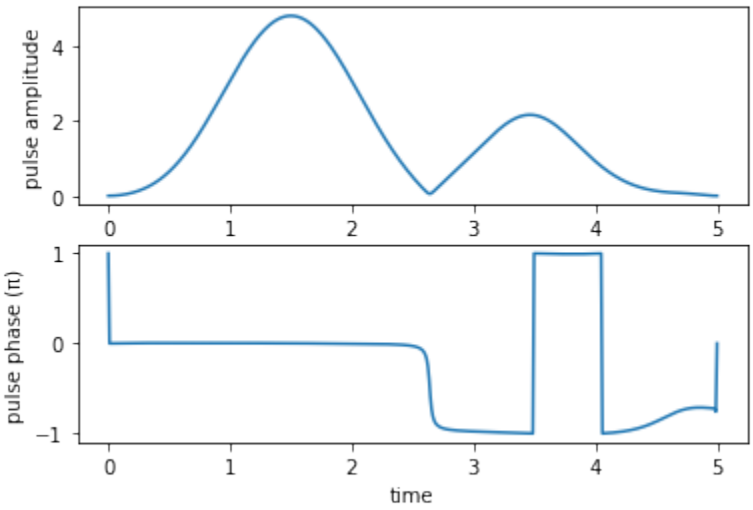
We look at the optimized controls and the population dynamics they induce:

[16]:
```
def plot_pulse_amplitude_and_phase(pulse_real, pulse_imaginary,tlist):
    ax1 = plt.subplot(211)
    ax2 = plt.subplot(212)
    amplitudes = [np.sqrt(x*x + y*y) for x,y in zip(pulse_real,pulse_imaginary)]
    phases = [np.arctan2(y,x)/np.pi for x,y in zip(pulse_real,pulse_imaginary)]
    ax1.plot(tlist,amplitudes)
    ax1.set_xlabel('time')
    ax1.set_ylabel('pulse amplitude')
    ax2.plot(tlist,phases)
    ax2.set_xlabel('time')
    ax2.set_ylabel('pulse phase (π)')
    plt.show()

print("pump pulse amplitude and phase:")
plot_pulse_amplitude_and_phase(
    opt_result.optimized_controls[0], opt_result.optimized_controls[1], tlist)
print("Stokes pulse amplitude and phase:")
plot_pulse_amplitude_and_phase(
    opt_result.optimized_controls[2], opt_result.optimized_controls[3], tlist)
```

```
pump pulse amplitude and phase:
```



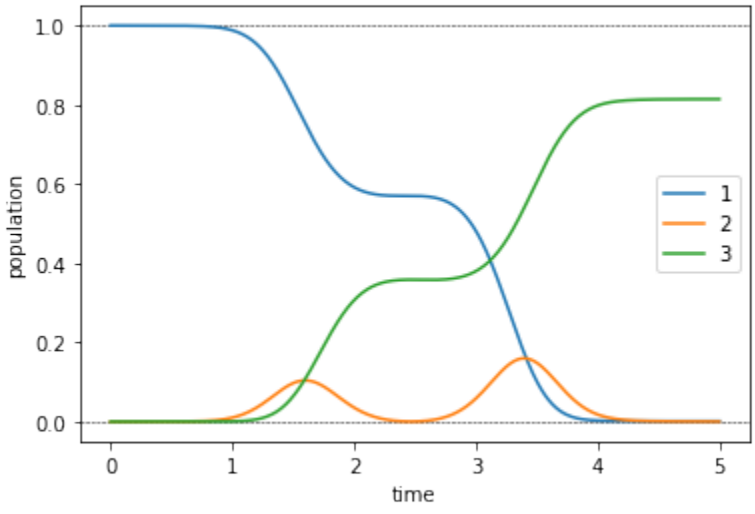

```
Stokes pulse amplitude and phase:
```

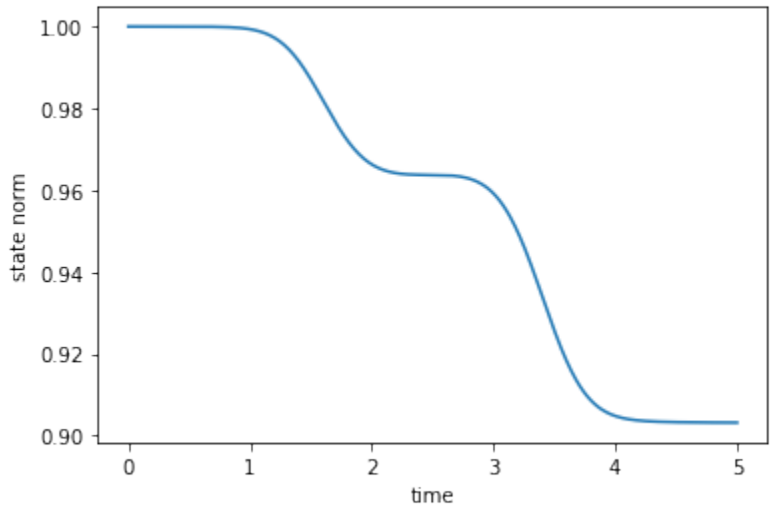


We check the evolution of the population due to our optimized pulses.

[17]:
```
opt_dynamics = opt_result.optimized_objectives[0].propagate(
    tlist, propagator=krotov.propagators.expm, e_ops=[proj1, proj2, proj3])
opt_states = opt_result.optimized_objectives[0].propagate(
    tlist, propagator=krotov.propagators.expm)
```

[18]:
```
plot_population(opt_dynamics)
plot_norm(opt_states)
```





These dynamics show that the non-Hermitian Hamiltonian has the desired effect: The population is steered out of the decaying $|2\rangle$ state, with the resulting loss in norm down to 10% from the 30% loss of the guess pulses. Indeed, these 10% are exactly the value of the error $1 - F_{\mathrm{re}}$, indicating that avoiding population in the $|2\rangle$ part is the difficult part of the optimization. Convergence towards this goal is slow, so we continue the optimization up to iteration 2000.

[19]:
```
dumpfile = "./non_herm_opt_result.dump"
if os.path.isfile(dumpfile):
    opt_result = krotov.result.Result.load(dumpfile, objectives)
else:
    opt_result = krotov.optimize_pulses(
        objectives, pulse_options, tlist,
        propagator=krotov.propagators.expm,
        chi_constructor=krotov.functionals.chis_re,
        info_hook=krotov.info_hooks.chain(print_fidelity),
        iter_stop=2000,
        continue_from=opt_result
    )
    opt_result.dump(dumpfile)
```

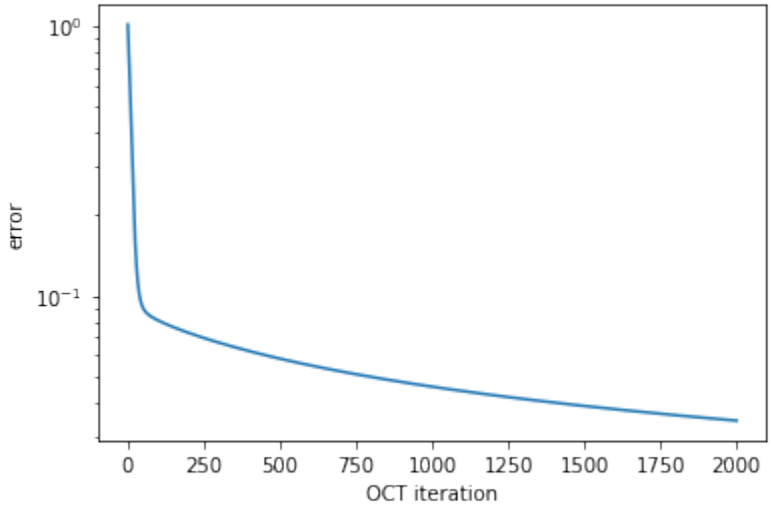
[20]:
```
print("Final fidelity: %.3f" % opt_result.info_vals[-1])
```

```
Final fidelity: 0.966
```

[21]:
```
def plot_convergence(result):
    fig, ax = plt.subplots()
    ax.semilogy(result.iters, 1-np.array(result.info_vals))
    ax.set_xlabel('OCT iteration')
    ax.set_ylabel('error')
    plt.show(fig)
```

To get a feel for the convergence, we can plot the optimization error over the iteration number:

[22]:
```
plot_convergence(opt_result)
```

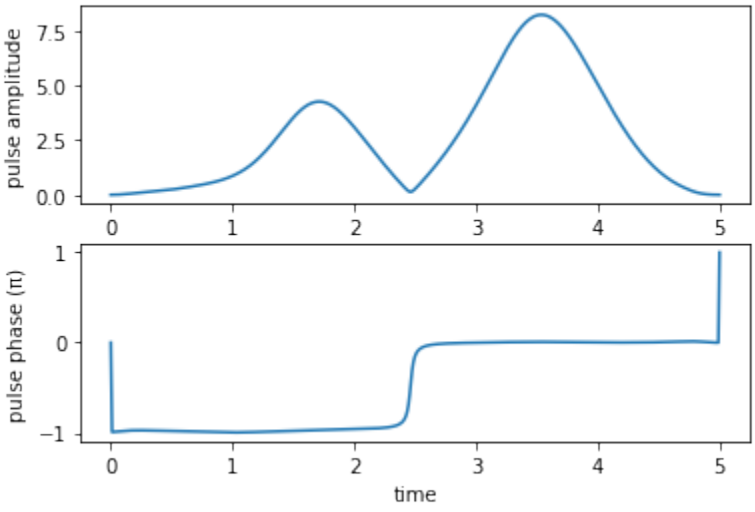


We have used here that the return value of the routine `print_fidelity` that was passed to the `optimize_pulses` routine as an `info_hook` is automatically accumulated in `result.info_vals`.

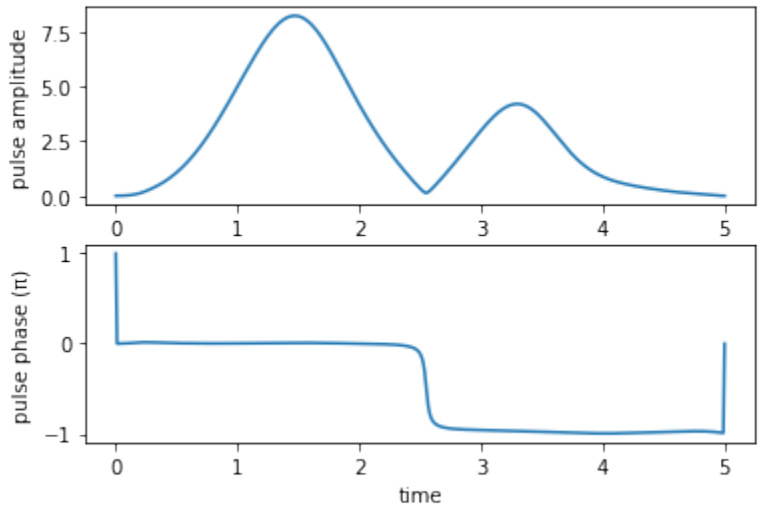
We also look at optimized controls and the dynamics they induce:

[23]:
```
print("pump pulse amplitude and phase:")
plot_pulse_amplitude_and_phase(
    opt_result.optimized_controls[0], opt_result.optimized_controls[1], tlist)
print("Stokes pulse amplitude and phase:")
plot_pulse_amplitude_and_phase(
    opt_result.optimized_controls[2], opt_result.optimized_controls[3], tlist)
```

```
pump pulse amplitude and phase:
```

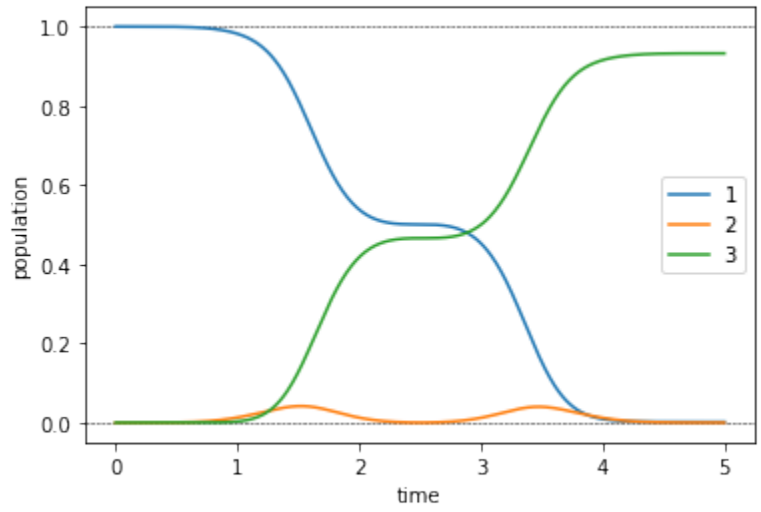


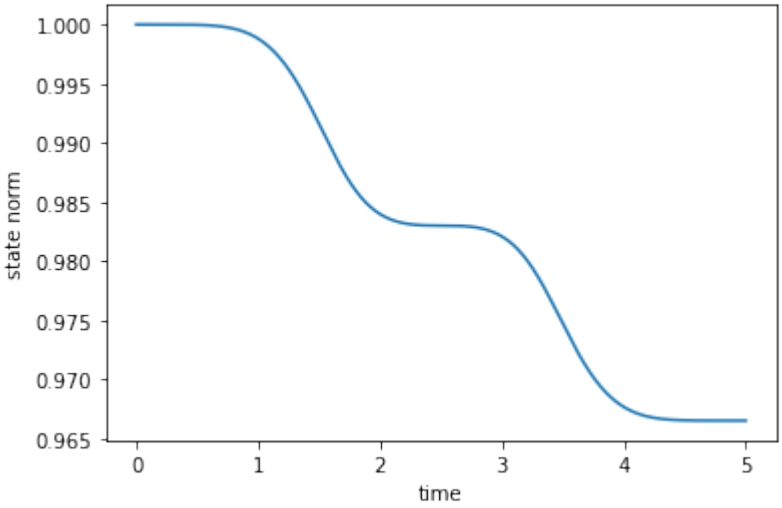

```
Stokes pulse amplitude and phase:
```



[24]:
```
opt_dynamics = opt_result.optimized_objectives[0].propagate(
    tlist, propagator=krotov.propagators.expm, e_ops=[proj1, proj2, proj3])
opt_states = opt_result.optimized_objectives[0].propagate(
    tlist, propagator=krotov.propagators.expm)
```

[25]:
```
plot_population(opt_dynamics)
plot_norm(opt_states)
```



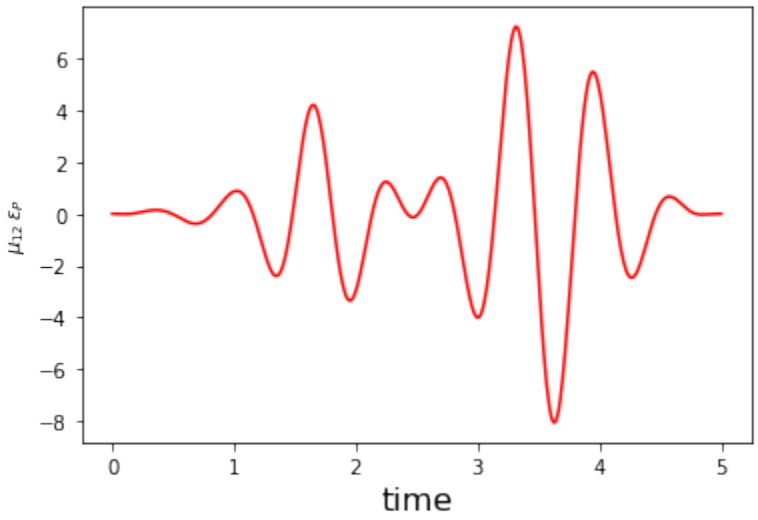


In accordance with the lower optimization error, the population dynamics now show a reasonably efficient transfer, and a significantly reduced population in state $|2\rangle$.

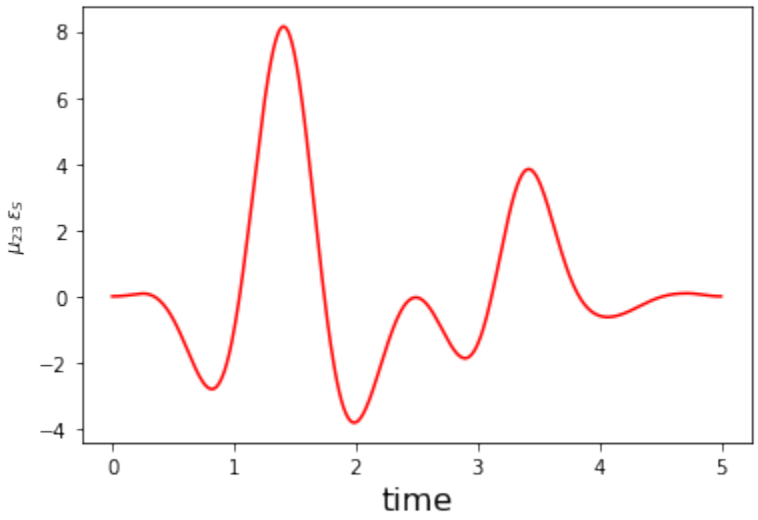
Finally, we can convert the complex-valued Ω_P and Ω_S functions to the physical electric fields ϵ_P and ϵ_S:

[26]:
```
def plot_physical_field(pulse_re, pulse_im, tlist, case=None):

    if case == 'pump':
        w = 9.5
    elif case == 'stokes':
        w = 4.5
    else:
        print('Error: selected case is not a valid option')
        return

    ax = plt.subplot(111)
    ax.plot(tlist,pulse_re*np.cos(w*tlist)-pulse_im*np.sin(w*tlist), 'r')
    ax.set_xlabel('time', fontsize = 16)
    if case == 'pump':
        ax.set_ylabel(r'$\mu_{12}\,\epsilon_{P}$')
    elif case == 'stokes':
        ax.set_ylabel(r'$ \mu_{23}\,\epsilon_{S}$')
    plt.show()


print('Physical electric pump pulse in the lab frame:')
plot_physical_field(
    opt_result.optimized_controls[0], opt_result.optimized_controls[1], tlist, case =
↪'pump')


print('Physical electric Stokes pulse in the lab frame:')
plot_physical_field(
    opt_result.optimized_controls[2], opt_result.optimized_controls[3], tlist, case =
↪'stokes')
```

```
Physical electric pump pulse in the lab frame:
```



Physical electric Stokes pulse in the lab frame:



9.4 Optimization of Dissipative Qubit Reset

[1]:
```
# NBVAL_IGNORE_OUTPUT
%load_ext watermark
import qutip
import numpy as np
import scipy
import matplotlib
import matplotlib.pylab as plt
import krotov

%watermark -v --iversions
```

```
matplotlib.pylab 1.17.2
matplotlib       3.1.1
```

(continues on next page)

(continued from previous page)

```
krotov           0.4.1
scipy            1.3.1
numpy            1.17.2
qutip            4.4.1
CPython 3.7.3
IPython 7.8.0
```

This example illustrates an optimization in an *open* quantum system, where the dynamics is governed by the Liouville-von Neumann equation. Hence, states are represented by density matrices $\hat{\rho}(t)$ and the time-evolution operator is given by a general dynamical map $\mathcal{E}$.

9.4.1 Define parameters

The system consists of a qubit with Hamiltonian $\hat{H}_q(t) = -\frac{\omega_q}{2}\hat{\sigma}_z - \frac{\epsilon(t)}{2}\hat{\sigma}_z$, where ω_q is an energy level splitting that can be dynamically adjusted by the control $\epsilon(t)$. This qubit couples strongly to another two-level system (TLS) with Hamiltonian $\hat{H}_t = -\frac{\omega_t}{2}\hat{\sigma}_z$ with static energy level splitting ω_t. The coupling strength between both systems is given by J with the interaction Hamiltonian given by $\hat{H}_\int = J\hat{\sigma}_x \otimes \hat{\sigma}_x$.

The Hamiltonian for the system of qubit and TLS is

$$\hat{H}(t) = \hat{H}_q(t) \otimes \mathbf{1}_t + \mathbf{1}_q \otimes \hat{H}_t + \hat{H}_\int.$$

In addition, the TLS is embedded in a heat bath with inverse temperature β. The TLS couples to the bath with rate κ. In order to simulate the dissipation arising from this coupling, we consider the two Lindblad operators

$$\hat{L}_1 = \sqrt{\kappa(N_{th}+1)}\mathbf{1}_q \otimes |0\rangle\langle 1|$$
$$\hat{L}_2 = \sqrt{\kappa N_{th}}\mathbf{1}_q \otimes |1\rangle\langle 0|$$

with $N_{th} = 1/(e^{\beta\omega_t} - 1)$.

[2]:
```
omega_q = 1.0  # qubit level splitting
omega_T = 3.0  # TLS level splitting
J = 0.1  # qubit-TLS coupling
kappa = 0.04  # TLS decay rate
beta = 1.0  # inverse bath temperature
T = 25.0  # final time
nt = 2500  # number of time steps
```

9.4.2 Define the Liouvillian

The dynamics of the qubit-TLS system state $\hat{\rho}(t)$ is governed by the Liouville-von Neumann equation

$$\begin{aligned}\frac{\partial}{\partial t}\hat{\rho}(t) &= \mathcal{L}(t)\hat{\rho}(t) \\ &= -i\left[\hat{H}(t), \hat{\rho}(t)\right] + \sum_{k=1,2}\left(\hat{L}_k\hat{\rho}(t)\hat{L}_k^\dagger - \frac{1}{2}\hat{L}_k^\dagger\hat{L}_k\hat{\rho}(t) - \frac{1}{2}\hat{\rho}(t)\hat{L}_k^\dagger\hat{L}_k\right).\end{aligned}$$

[3]:
```
def liouvillian(omega_q, omega_T, J, kappa, beta):
    """Liouvillian for the coupled system of qubit and TLS"""

    # drift qubit Hamiltonian
    H0_q = 0.5 * omega_q * np.diag([-1, 1])
    # drive qubit Hamiltonian
    H1_q = 0.5 * np.diag([-1, 1])

    # drift TLS Hamiltonian
    H0_T = 0.5 * omega_T * np.diag([-1, 1])

    # Lift Hamiltonians to joint system operators
    H0 = np.kron(H0_q, np.identity(2)) + np.kron(np.identity(2), H0_T)
    H1 = np.kron(H1_q, np.identity(2))

    # qubit-TLS interaction
    H_int = J * np.fliplr(np.diag([0, 1, 1, 0]))

    # convert Hamiltonians to QuTiP objects
    H0 = qutip.Qobj(H0 + H_int)
    H1 = qutip.Qobj(H1)

    # Define Lindblad operators
    N = 1.0 / (np.exp(beta * omega_T) - 1.0)
    # Cooling on TLS
    L1 = np.sqrt(kappa * (N + 1)) * np.kron(
        np.identity(2), np.array([[0, 1], [0, 0]])
    )
    # Heating on TLS
    L2 = np.sqrt(kappa * N) * np.kron(
        np.identity(2), np.array([[0, 0], [1, 0]])
    )

    # convert Lindblad operators to QuTiP objects
    L1 = qutip.Qobj(L1)
    L2 = qutip.Qobj(L2)

    # generate the Liouvillian
    L0 = qutip.liouvillian(H=H0, c_ops=[L1, L2])
    L1 = qutip.liouvillian(H=H1)

    # Shift the qubit and TLS into resonance by default
    eps0 = lambda t, args: omega_T - omega_q

    return [L0, [L1, eps0]]


L = liouvillian(omega_q=omega_q, omega_T=omega_T, J=J, kappa=kappa, beta=beta)
```

9.4.3 Define the optimization target

The initial state of qubit and TLS are assumed to be in thermal equilibrium with the heat bath (although only the TLS is directly interacting with the bath). Both states are given by

$$\hat{\rho}_\alpha^{th} = \frac{e^{x_\alpha} |0\rangle\langle 0| + e^{-x_\alpha} |1\rangle\langle 1|}{2\cosh(x_\alpha)}, \qquad x_\alpha = \frac{\omega_\alpha \beta}{2},$$

with $\alpha = q, t$. The initial state of the bipartite system of qubit and TLS is given by the thermal state $\hat{\rho}_{th} = \hat{\rho}_q^{th} \otimes \hat{\rho}_t^{th}$.

[4]:
```
x_q = omega_q * beta / 2.0
rho_q_th = np.diag([np.exp(x_q), np.exp(-x_q)]) / (2 * np.cosh(x_q))

x_T = omega_T * beta / 2.0
rho_T_th = np.diag([np.exp(x_T), np.exp(-x_T)]) / (2 * np.cosh(x_T))

rho_th = qutip.Qobj(np.kron(rho_q_th, rho_T_th))
```

Since we are ultimately only interested in the state of the qubit, we define `trace_TLS`. It returns the reduced state of the qubit $\hat{\rho}_q = \mathrm{tr}_t\{\hat{\rho}\}$ when passed the state $\hat{\rho}$ of the bipartite system.

[5]:
```
def trace_TLS(rho):
    """Partial trace over the TLS degrees of freedom"""
    rho_q = np.zeros(shape=(2, 2), dtype=np.complex_)
    rho_q[0, 0] = rho[0, 0] + rho[1, 1]
    rho_q[0, 1] = rho[0, 2] + rho[1, 3]
    rho_q[1, 0] = rho[2, 0] + rho[3, 1]
    rho_q[1, 1] = rho[2, 2] + rho[3, 3]
    return qutip.Qobj(rho_q)
```

The target state is (temporarily) the ground state of the bipartite system, i.e., $\hat{\rho}_{\text{tgt}} = |00\rangle\langle 00|$. Note that in the end we will only optimize the reduced state of the qubit.

[6]:
```
rho_q_trg = np.diag([1, 0])
rho_T_trg = np.diag([1, 0])
rho_trg = np.kron(rho_q_trg, rho_T_trg)
rho_trg = qutip.Qobj(rho_trg)
```

Next, the list of `objectives` is defined, which contains the initial and target state and the Liouvillian $\mathcal{L}(t)$ that determines the system dynamics.

[7]:
```
objectives = [krotov.Objective(initial_state=rho_th, target=rho_trg, H=L)]
objectives
```

[7]:
```
[Objective[ρ₀[4,4] to ρ₁[4,4] via [₀[[4,4],[4,4]], [₁[[4,4],[4,4]], u₁(t)]]]]
```

In the following, we define the shape function $S(t)$, which we use in order to ensure a smooth switch on and off in the beginning and end. Note that at times t where $S(t)$ vanishes, the updates of the field is suppressed.

[8]:
```
def S(t):
    """Shape function for the field update"""
    return krotov.shapes.flattop(
        t, t_start=0, t_stop=T, t_rise=0.05 * T, t_fall=0.05 * T, func='sinsq'
    )
```

We re-use this function to also shape the guess control $\epsilon_0(t)$ to be zero at $t = 0$ and $t = T$. This is on top of the originally defined constant value shifting the qubit and TLS into resonance.

[9]:
```
def shape_field(eps0):
    """Applies the shape function S(t) to the guess field"""
    eps0_shaped = lambda t, args: eps0(t, args) * S(t)
    return eps0_shaped
```

(continues on next page)

(continued from previous page)

```
L[1][1] = shape_field(L[1][1])
```

At last, before heading to the actual optimization below, we assign the shape function $S(t)$ to the OCT parameters of the control and choose `lambda_a`, a numerical parameter that controls the field update magnitude in each iteration.

[10]:
```
pulse_options = {L[1][1]: dict(lambda_a=0.01, update_shape=S)}
```

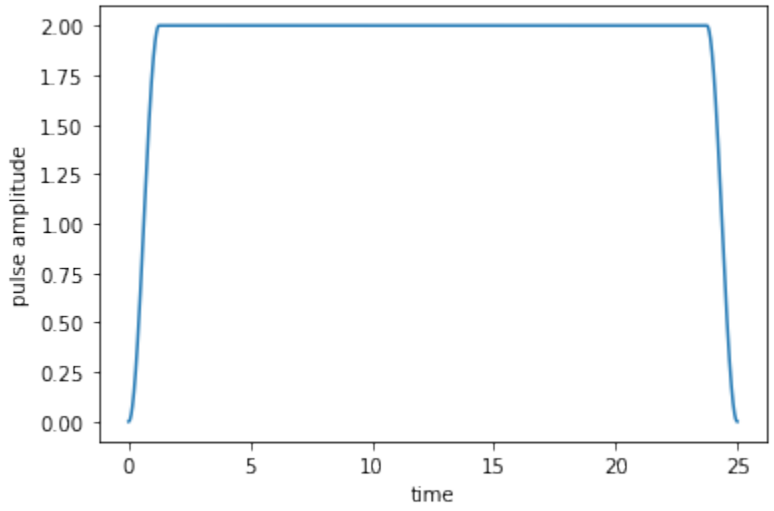
9.4.4 Simulate the dynamics of the guess field

[11]:
```
tlist = np.linspace(0, T, nt)
```

[12]:
```
def plot_pulse(pulse, tlist):
    fig, ax = plt.subplots()
    if callable(pulse):
        pulse = np.array([pulse(t, args=None) for t in tlist])
    ax.plot(tlist, pulse)
    ax.set_xlabel('time')
    ax.set_ylabel('pulse amplitude')
    plt.show(fig)
```

The following plot shows the guess field $\epsilon_0(t)$ as a constant that puts qubit and TLS into resonance, but with a smooth switch-on and switch-off.

[13]:
```
plot_pulse(L[1][1], tlist)
```



We solve the equation of motion for this guess field, storing the expectation values for the population in the bipartite levels:

[14]:
```
psi00 = qutip.Qobj(np.kron(np.array([1,0]), np.array([1,0])))
psi01 = qutip.Qobj(np.kron(np.array([1,0]), np.array([0,1])))
psi10 = qutip.Qobj(np.kron(np.array([0,1]), np.array([1,0])))
psi11 = qutip.Qobj(np.kron(np.array([0,1]), np.array([0,1])))
```

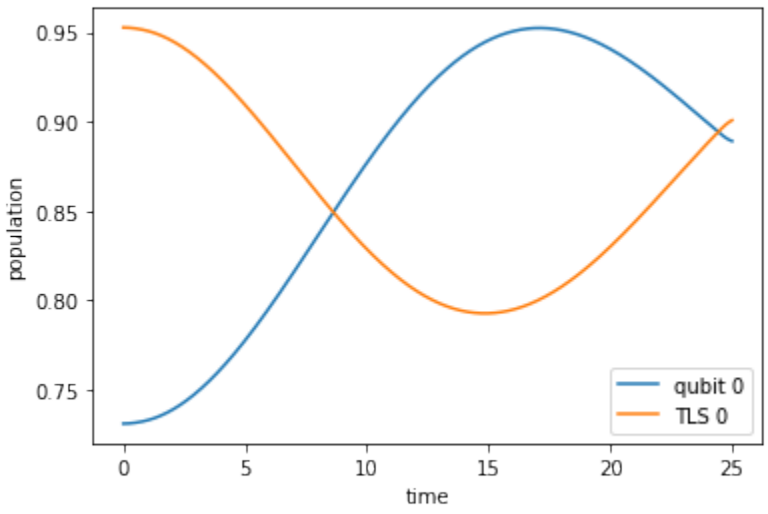
(continues on next page)

(continued from previous page)

```
proj_00 = qutip.ket2dm(psi00)
proj_01 = qutip.ket2dm(psi01)
proj_10 = qutip.ket2dm(psi10)
proj_11 = qutip.ket2dm(psi11)
```

[15]:
```
guess_dynamics = objectives[0].mesolve(
    tlist, e_ops=[proj_00, proj_01, proj_10, proj_11]
)
```

[16]:
```
def plot_population(result):
    fig, ax = plt.subplots()
    ax.plot(
        result.times,
        np.array(result.expect[0]) + np.array(result.expect[1]),
        label='qubit 0',
    )
    ax.plot(
        result.times,
        np.array(result.expect[0]) + np.array(result.expect[2]),
        label='TLS 0',
    )
    ax.legend()
    ax.set_xlabel('time')
    ax.set_ylabel('population')
    plt.show(fig)


plot_population(guess_dynamics)
```



The population dynamics of qubit and TLS ground state show that both are oscillating and especially the qubit's ground state population reaches a maximal value at intermediate times $t < T$. This maximum is indeed the maximum that is physically possible. It corresponds to a perfect swap of the initial qubit and TLS purities. However, we want to reach this maximum at final time T (not before), so the guess control is not yet working as desired.

9.4.5 Optimize

Our optimization target is the ground state $|\Psi_q^{\text{tgt}}\rangle = |0\rangle$ of the qubit, irrespective of the state of the TLS. Thus, our optimization functional reads

$$F_{re} = 1 - \left\langle \Psi_q^{\text{tgt}} \right| \text{tr}_t\{\hat{\rho}(T)\} \left| \Psi_q^{\text{tgt}} \right\rangle ,$$

and we first define `print_qubit_error`, which prints out the above functional after each iteration.

[17]:
```
def print_qubit_error(**args):
    """Utility function writing the qubit error to screen"""
    taus = []
    for state_T in args['fw_states_T']:
        state_q_T = trace_TLS(state_T)
        taus.append(state_q_T[0, 0].real)
    J_T_re = 1 - np.average(taus)
    print("    qubit error: %.1e" % J_T_re)
    return J_T_re
```

In order to minimize the above functional, we need to provide the correct `chi_constructor` for the Krotov optimization. This is the only place where the functional (implicitly) enters the optimization. Given our bipartite system and choice of F_{re}, the equation for $\hat{\chi}(T)$ reads

$$\hat{\chi}(T) = \sum_{k=0,1} a_k \hat{\rho}_q^{\text{tgt}} \otimes |k\rangle \langle k|$$

with $\{|k\rangle\}$ a basis for the TLS Hilbert space.

[18]:
```
def TLS_onb_trg():
    """Returns the tensor product of qubit target state
    and a basis for the TLS Hilbert space"""
    rho1 = qutip.Qobj(np.kron(rho_q_trg, np.diag([1, 0])))
    rho2 = qutip.Qobj(np.kron(rho_q_trg, np.diag([0, 1])))
    return [rho1, rho2]


TLS_onb = TLS_onb_trg()


def chis_qubit(fw_states_T, objectives, tau_vals):
    """Calculate chis for the chosen functional"""
    chis = []
    for state_i_T in fw_states_T:
        chis_i = np.zeros(shape=(4, 4), dtype=np.complex_)
        for state_k in TLS_onb:
            a_i_k = krotov.optimize._overlap(state_i_T, state_k)
            chis_i += a_i_k * state_k
        chis.append(qutip.Qobj(chis_i))
    return chis
```

We now carry out the optimization for five iterations.

[19]:
```
# NBVAL_IGNORE_OUTPUT
# the DensityMatrixODEPropagator is not sufficiently exact to guarantee that
# you won't get slightly different results in the optimization when
# running this on different systems
```

(continues on next page)

(continued from previous page)

```
opt_result = krotov.optimize_pulses(
    objectives,
    pulse_options,
    tlist,
    propagator=krotov.propagators.DensityMatrixODEPropagator(
        atol=1e-10, rtol=1e-8
    ),
    chi_constructor=chis_qubit,
    info_hook=krotov.info_hooks.chain(
        krotov.info_hooks.print_debug_information, print_qubit_error
    ),
    check_convergence=krotov.convergence.check_monotonic_error,
    iter_stop=5,
)
```

```
Iteration 0
    objectives:
        1:ρ₀[4,4] to ρ₁[4,4] via [₀[[4,4],[4,4]], [₁[[4,4],[4,4]], u₂(t)]]
    adjoint objectives:
        1:ρ₂[4,4] to ρ₃[4,4] via [₂[[4,4],[4,4]], [₃[[4,4],[4,4]], u₂(t)]]
    S(t) (ranges): [0.000000, 1.000000]
    duration: 0.5 secs (started at 2019-10-11 12:21:58)
    optimized pulses (ranges): [0.00, 2.00]
    ∫gₐ(t)dt: 0.00e+00
    λₐ: 1.00e-02
    storage (bw, fw, fw0): None, None, None
    fw_states_T norm: 1.000000
    τ: (7.97e-01:0.00π)
    qubit error: 1.1e-01
Iteration 1
    duration: 2.8 secs (started at 2019-10-11 12:21:58)
    optimized pulses (ranges): [0.00, 2.06]
    ∫gₐ(t)dt: 7.72e-02
    λₐ: 1.00e-02
    storage (bw, fw, fw0): [1 * ndarray(2500)] (1.3 MB), None, None
    fw_states_T norm: 1.000000
    τ: (7.98e-01:0.00π)
    qubit error: 1.1e-01
Iteration 2
    duration: 2.8 secs (started at 2019-10-11 12:22:01)
    optimized pulses (ranges): [0.00, 2.23]
    ∫gₐ(t)dt: 5.72e-01
    λₐ: 1.00e-02
    storage (bw, fw, fw0): [1 * ndarray(2500)] (1.3 MB), None, None
    fw_states_T norm: 1.000000
    τ: (8.01e-01:0.00π)
    qubit error: 6.7e-02
Iteration 3
    duration: 3.2 secs (started at 2019-10-11 12:22:04)
    optimized pulses (ranges): [0.00, 2.33]
    ∫gₐ(t)dt: 8.11e-02
    λₐ: 1.00e-02
    storage (bw, fw, fw0): [1 * ndarray(2500)] (1.3 MB), None, None
    fw_states_T norm: 1.000000
    τ: (7.99e-01:0.00π)
    qubit error: 5.0e-02
Iteration 4
```

(continues on next page)

(continued from previous page)

```
    duration: 2.7 secs (started at 2019-10-11 12:22:07)
    optimized pulses (ranges): [0.00, 2.19]
    ∫gₐ(t)dt: 2.16e-01
    λₐ: 1.00e-02
    storage (bw, fw, fw0): [1 * ndarray(2500)] (1.3 MB), None, None
    fw_states_T norm: 1.000000
    τ: (8.02e-01:0.00π)
    qubit error: 4.9e-02
Iteration 5
    duration: 3.5 secs (started at 2019-10-11 12:22:10)
    optimized pulses (ranges): [0.00, 2.15]
    ∫gₐ(t)dt: 6.38e-02
    λₐ: 1.00e-02
    storage (bw, fw, fw0): [1 * ndarray(2500)] (1.3 MB), None, None
    fw_states_T norm: 1.000000
    τ: (8.03e-01:0.00π)
    qubit error: 4.9e-02
```

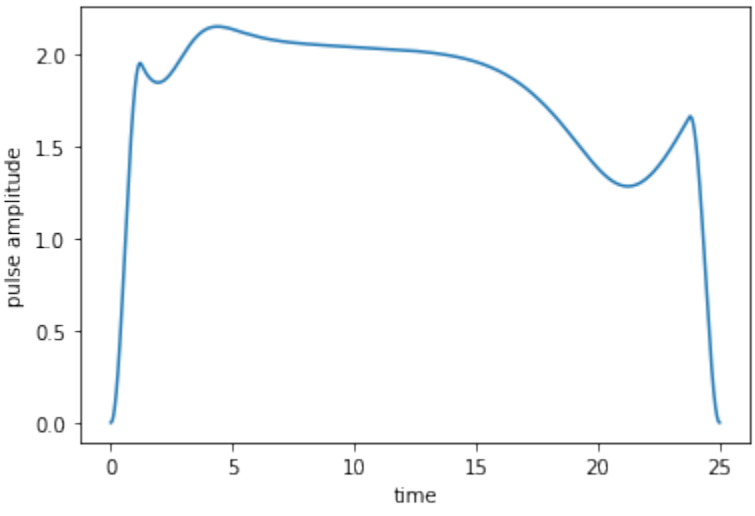
[20]:
```
opt_result
```

[20]:
```
Krotov Optimization Result
--------------------------
- Started at 2019-10-11 12:21:58
- Number of objectives: 1
- Number of iterations: 5
- Reason for termination: Reached 5 iterations
- Ended at 2019-10-11 12:22:13
```

9.4.6 Simulate the dynamics of the optimized field

The plot of the optimized field shows that the optimization slightly shifts the field such that qubit and TLS are no longer perfectly in resonance.

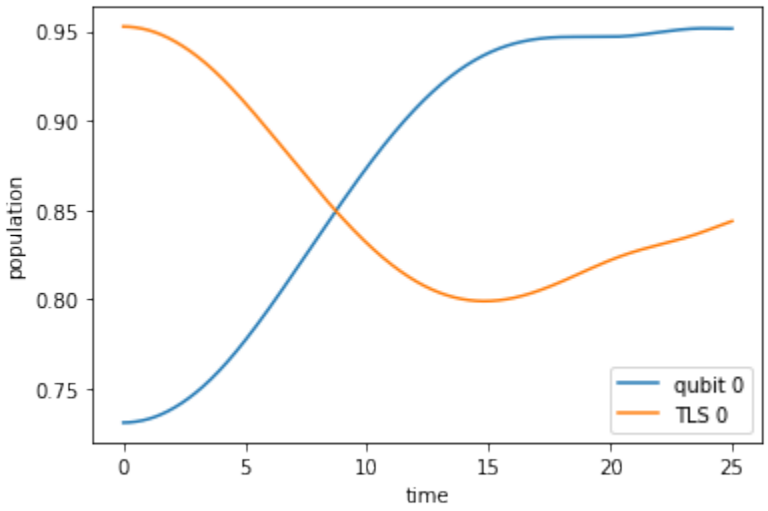
[21]:
```
plot_pulse(opt_result.optimized_controls[0], tlist)
```



This slight shift of qubit and TLS out of resonance delays the population oscillations between qubit and TLS ground state such that the qubit ground state is maximally populated at final

time T.

[22]:
```
optimized_dynamics = opt_result.optimized_objectives[0].mesolve(
    tlist, e_ops=[proj_00, proj_01, proj_10, proj_11]
)

plot_population(optimized_dynamics)
```



9.5 Optimization of an X-Gate for a Transmon Qubit

[1]:
```
# NBVAL_IGNORE_OUTPUT
%load_ext watermark
import os
import qutip
import numpy as np
import scipy
import matplotlib
import matplotlib.pylab as plt
import krotov
from scipy.fftpack import fft
from scipy.interpolate import interp1d
%watermark -v --iversions
```

```
matplotlib       3.1.1
qutip            4.4.1
krotov           0.4.1
numpy            1.17.2
matplotlib.pylab 1.17.2
scipy            1.3.1
CPython 3.7.3
IPython 7.8.0
```

In the previous examples, we have only optimized for state-to-state transitions, i.e., for a single objective. This example shows the optimization of a simple quantum gate, which requires multiple objectives to be fulfilled simultaneously (one for each state in the logical basis). We consider a superconducting "transmon" qubit and implement a single-qubit Pauli-X gate.

9.5.1 The transmon Hamiltonian

The effective Hamiltonian of a single transmon depends on the capacitive energy $E_C = e^2/2C$ and the Josephson energy E_J, an energy due to the Josephson junction working as a nonlinear inductor periodic with the flux Φ. In the so-called transmon limit, the ratio between these two energies lies around $E_J/E_C \approx 45$. The Hamiltonian for the transmon is

$$\hat{H}_0 = 4E_C(\hat{n} - n_g)^2 - E_J \cos(\hat{\Phi})$$

where $\hat{n}$ is the number operator, which counts the relative number of Cooper pairs capacitively stored in the junction, and n_g is the effective offset charge measured in Cooper pair charge units. The equation can be written in a truncated charge basis defined by the number operator $\hat{n}\,|n\rangle = n\,|n\rangle$ such that

$$\hat{H}_0 = 4E_C \sum_{j=-N}^{N} (j - n_g)^2\,|j\rangle\langle j| - \frac{E_J}{2} \sum_{j=-N}^{N-1} (|j+1\rangle\langle j| + |j\rangle\langle j+1|).$$

A voltage $V(t)$ applied to the circuit couples to the charge Hamiltonian $\hat{q}$, which in the (truncated) charge basis reads

$$\hat{H}_1 = \hat{q} = \sum_{j=-N}^{N} -2n\,|n\rangle\langle n|\ .$$

The factor 2 is due to the charge carriers in a superconductor being Cooper pairs. The total Hamiltonian is

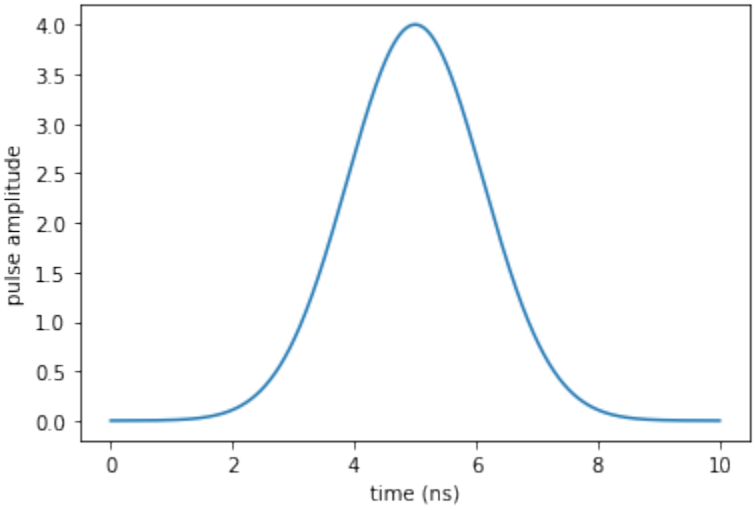
$$\hat{H} = \hat{H}_0 + V(t) \cdot \hat{H}_1$$

We use a Gaussian voltage profile as the guess pulse:

[2]:
```
tlist = np.linspace(0, 10, 1000)

def eps0(t, args):
    T = tlist[-1]
    return 4 * np.exp(-40.0 * (t / T - 0.5) ** 2)
```

[3]:
```
def plot_pulse(pulse, tlist, xlimit=None):
    fig, ax = plt.subplots()
    if callable(pulse):
        pulse = np.array([pulse(t, None) for t in tlist])
    ax.plot(tlist, pulse)
    ax.set_xlabel('time (ns)')
    ax.set_ylabel('pulse amplitude')
    if xlimit is not None:
        ax.set_xlim(xlimit)
    plt.show(fig)
```

[4]:
```
plot_pulse(eps0, tlist)
```



The complete Hamiltonian is instantiated as

[5]:
```
def transmon_hamiltonian(Ec=0.386, EjEc=45, nstates=8, ng=0.0, T=10.0):
    """Transmon Hamiltonian

    Args:
        Ec: capacitive energy
        EjEc: ratio `Ej` / `Ec`
        nstates: defines the maximum and minimum states for the basis. The
            truncated basis will have a total of ``2*nstates + 1`` states

        ng: offset charge
        T: gate duration
    """

    Ej = EjEc * Ec
    n = np.arange(-nstates, nstates + 1)
    up = np.diag(np.ones(2 * nstates), k=-1)
    do = up.T
    H0 = qutip.Qobj(np.diag(4 * Ec * (n - ng) ** 2) - Ej * (up + do) / 2.0)
    H1 = qutip.Qobj(-2 * np.diag(n))

    return [H0, [H1, eps0]]
```

[6]:
```
H = transmon_hamiltonian()
```

We define the logical basis $|0_l\rangle$ and $|1_l\rangle$ (not to be confused with the charge states $|n=0\rangle$ and $|n=1\rangle$) as the eigenstates of the drift Hamiltonian $\hat{H}_0$ with the lowest energy. The optimization goal is to find a potential $V_{opt}(t)$ such that after a given final time T implements an X-gate on this logical basis.

[7]:
```
def logical_basis(H):
    H0 = H[0]
    eigenvals, eigenvecs = scipy.linalg.eig(H0.full())
    ndx = np.argsort(eigenvals.real)
    E = eigenvals[ndx].real
    V = eigenvecs[:, ndx]
    psi0 = qutip.Qobj(V[:, 0])
```

(continues on next page)

(continued from previous page)

```
    psi1 = qutip.Qobj(V[:, 1])
    w01 = E[1] - E[0]  # Transition energy between states
    print("Energy of qubit transition is %.3f" % w01)
    return psi0, psi1

psi0, psi1 = logical_basis(H)
```

```
Energy of qubit transition is 6.914
```

We also introduce the projectors $P_i = |\psi_i\rangle \langle\psi_i|$ for the logical states $|\psi_i\rangle \in \{|0_l\rangle, |1_l\rangle\}$

[8]:
```
proj0 = qutip.ket2dm(psi0)
proj1 = qutip.ket2dm(psi1)
```

9.5.2 Optimization target

The key insight for the realization of a quantum gate $\hat{O}$ is that (by virtue of linearity)

$$|\Psi(t=0)\rangle \rightarrow |\Psi(t=T)\rangle = \hat{U}(T, \epsilon(t))\,|\Psi(0)\rangle = \hat{O}\,|\Psi(0)\rangle$$

is fulfilled for an arbitrary state $|\Psi(t=0)\rangle$ if an only if $\hat{U}(T, \epsilon(t))\,|k\rangle = \hat{O}\,|k\rangle$ for every state $|k\rangle$ in logical basis, for the time evolution operator $\hat{U}(T, \epsilon(t))$ from $t = 0$ to $t = T$ under the same control $\epsilon(t)$.

The function `krotov.gate_objectives` automatically sets up the corresponding objectives $\forall\,|k\rangle : |k\rangle \rightarrow \hat{O}\,|k\rangle$:

[9]:
```
objectives = krotov.gate_objectives(
    basis_states=[psi0, psi1], gate=qutip.operators.sigmax(), H=H
)

objectives
```

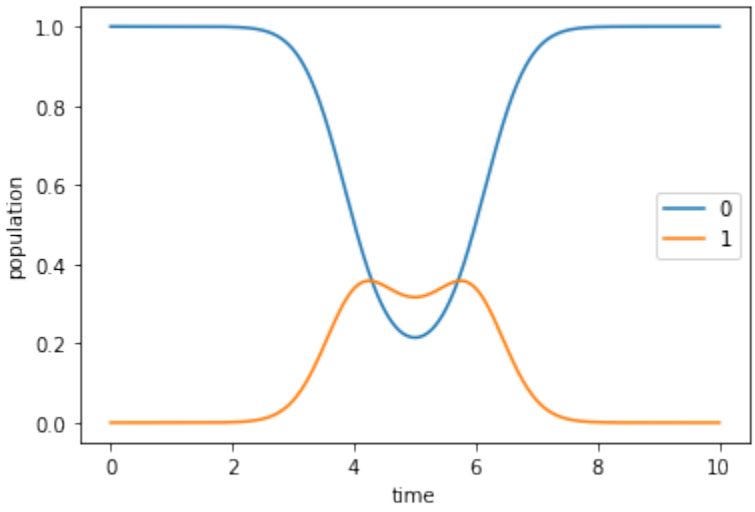
[9]:
```
[Objective[|Ψ₀(17)⟩ to |Ψ₁(17)⟩ via [H₀[17,17], [H₁[17,17], u₁(t)]]],
 Objective[|Ψ₁(17)⟩ to |Ψ₀(17)⟩ via [H₀[17,17], [H₁[17,17], u₁(t)]]]]
```

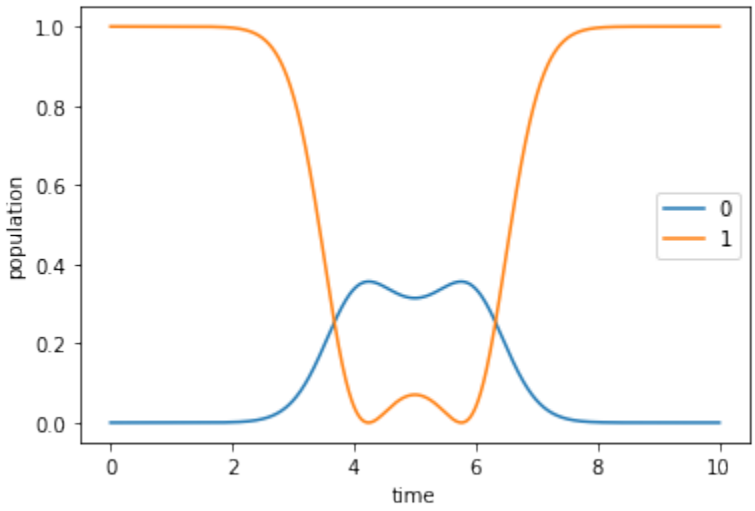
9.5.3 Dynamics of the guess pulse

[10]:
```
guess_dynamics = [
    objectives[x].mesolve(tlist, e_ops=[proj0, proj1]) for x in [0, 1]
]
```

[11]:
```
def plot_population(result):
    '''Representation of the expected values for the initial states'''
    fig, ax = plt.subplots()
    ax.plot(result.times, result.expect[0], label='0')
    ax.plot(result.times, result.expect[1], label='1')
    ax.legend()
    ax.set_xlabel('time')
    ax.set_ylabel('population')
    plt.show(fig)
```

[12]:
```
plot_population(guess_dynamics[0])
plot_population(guess_dynamics[1])
```





9.5.4 Optimization

We define the desired shape of the update and the factor λ_a, and then start the optimization

[13]:
```
def S(t):
    """Scales the Krotov methods update of the pulse value at the time t"""
    return krotov.shapes.flattop(
        t, t_start=0.0, t_stop=10.0, t_rise=0.5, func='sinsq'
    )

pulse_options = {H[1][1]: dict(lambda_a=1, update_shape=S)}
```

[14]:
```
opt_result = krotov.optimize_pulses(
    objectives,
```

(continues on next page)

(continued from previous page)

```
    pulse_options,
    tlist,
    propagator=krotov.propagators.expm,
    chi_constructor=krotov.functionals.chis_re,
    info_hook=krotov.info_hooks.print_table(
        J_T=krotov.functionals.J_T_re,
        show_g_a_int_per_pulse=True,
        unicode=False,
    ),
    check_convergence=krotov.convergence.Or(
        krotov.convergence.value_below(1e-3, name='J_T'),
        krotov.convergence.delta_below(1e-5),
        krotov.convergence.check_monotonic_error,
    ),
    iter_stop=5,
    parallel_map=(
        qutip.parallel_map,
        qutip.parallel_map,
        krotov.parallelization.parallel_map_fw_prop_step,
    ),
)
```

```
  iter.        J_T     g_a_int           J  Delta_J_T    Delta J  secs
     0   1.00e+00    0.00e+00   1.00e+00         n/a        n/a     2
     1   2.80e-01    3.41e-01   6.22e-01  -7.20e-01  -3.78e-01     6
     2   2.12e-01    3.06e-02   2.43e-01  -6.81e-02  -3.75e-02     7
     3   1.35e-01    3.28e-02   1.68e-01  -7.72e-02  -4.44e-02     6
     4   9.79e-02    1.56e-02   1.13e-01  -3.71e-02  -2.15e-02     5
     5   7.13e-02    1.11e-02   8.25e-02  -2.65e-02  -1.54e-02     6
```

(this takes a while ...)

[15]:
```
dumpfile = "./transmonxgate_opt_result.dump"
if os.path.isfile(dumpfile):
    opt_result = krotov.result.Result.load(dumpfile, objectives)
else:
    opt_result = krotov.optimize_pulses(
        objectives,
        pulse_options,
        tlist,
        propagator=krotov.propagators.expm,
        chi_constructor=krotov.functionals.chis_re,
        info_hook=krotov.info_hooks.print_table(
            J_T=krotov.functionals.J_T_re,
            show_g_a_int_per_pulse=True,
            unicode=False,
        ),
        check_convergence=krotov.convergence.Or(
            krotov.convergence.value_below(1e-3, name='J_T'),
            krotov.convergence.delta_below(1e-5),
            krotov.convergence.check_monotonic_error,
        ),
        iter_stop=1000,
        parallel_map=(
            qutip.parallel_map,
            qutip.parallel_map,
            krotov.parallelization.parallel_map_fw_prop_step,
```

(continues on next page)

(continued from previous page)

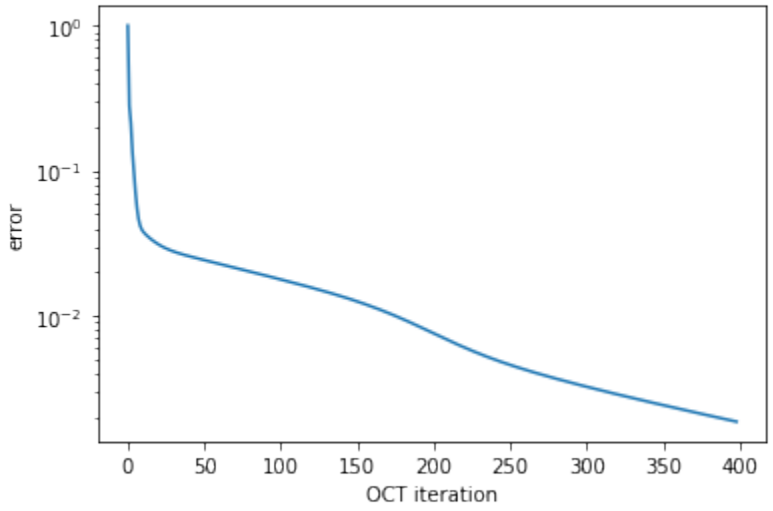
```
        ),
        continue_from=opt_result
    )
    opt_result.dump(dumpfile)
```

[16]:
```
opt_result
```

[16]:
```
Krotov Optimization Result
--------------------------
- Started at 2019-04-12 17:45:43
- Number of objectives: 2
- Number of iterations: 398
- Reason for termination: Reached convergence: Δ(('info_vals', T[-1]),('info_vals',␣
↪T[-2])) < 1e-05
- Ended at 2019-04-12 18:20:47
```

[17]:
```
def plot_convergence(result):
    fig, ax = plt.subplots()
    ax.semilogy(result.iters, np.array(result.info_vals))
    ax.set_xlabel('OCT iteration')
    ax.set_ylabel('error')
    plt.show(fig)
```

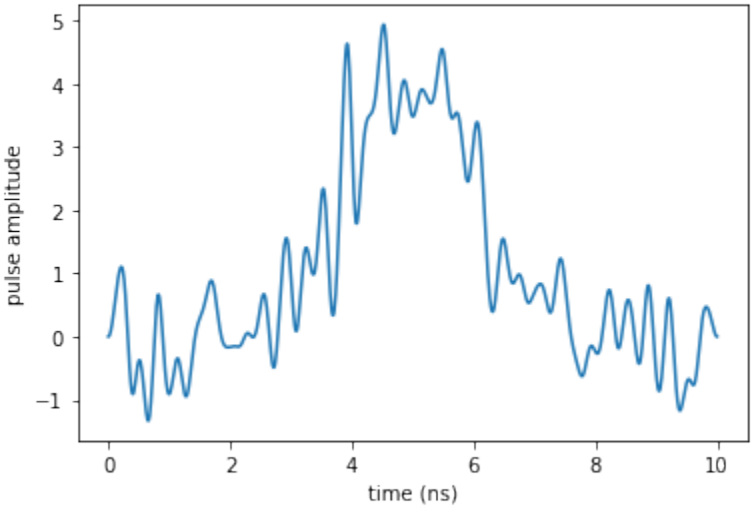
[18]:
```
plot_convergence(opt_result)
```



9.5.5 Optimized pulse and dynamics

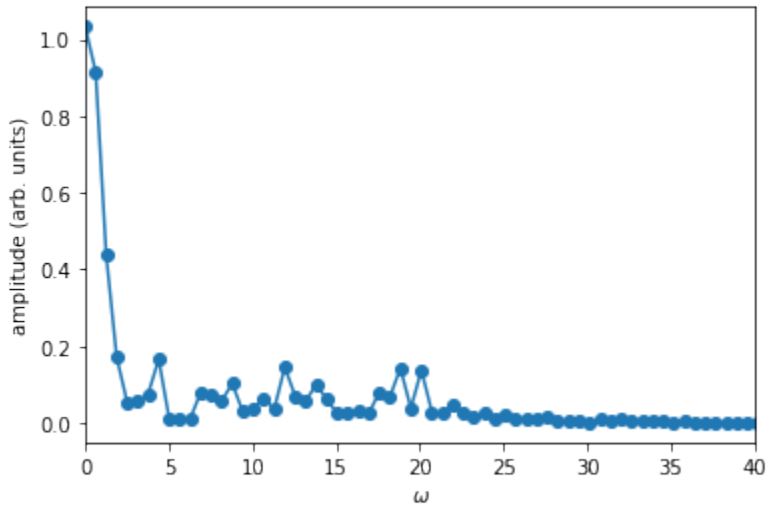
We obtain the following optimized pulse:

[19]:
```
plot_pulse(opt_result.optimized_controls[0], tlist)
```



The oscillations in the control shape indicate non-negligible spectral broadening:

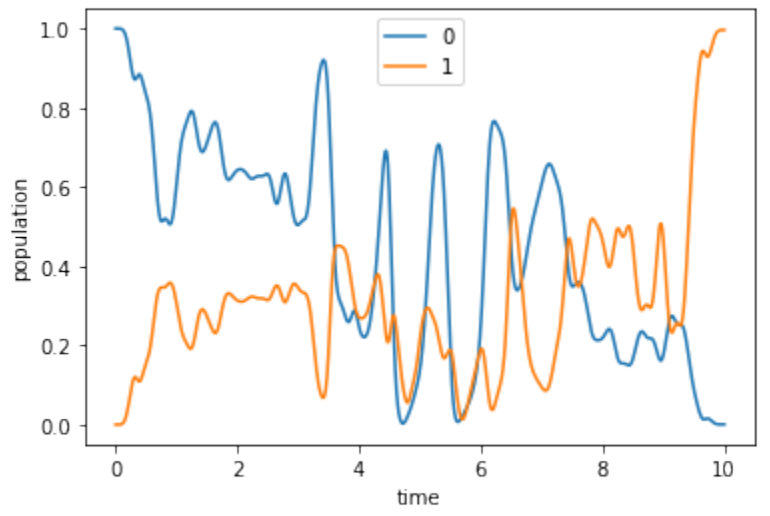
[20]:
```
def plot_spectrum(pulse, tlist, xlim=None):

    if callable(pulse):
        pulse = np.array([pulse(t, None) for t in tlist])

    dt = tlist[1] - tlist[0]
    n = len(tlist)

    w = np.fft.fftfreq(n, d=dt/(2.0*np.pi))
    # the factor 2π in the normalization means that
    # the spectrum is in units of angular frequency,
    # which is normally what we want

    spectrum = np.fft.fft(pulse) / n
    # normalizing the spectrum with n means that
    # the y-axis is independent of dt

    # we assume a real-valued pulse, so we throw away
    # the half of the spectrum with negative frequencies
    w = w[range(int(n / 2))]
    spectrum = np.abs(spectrum[range(int(n / 2))])

    fig, ax = plt.subplots()
    ax.plot(w, spectrum, '-o')
    ax.set_xlabel(r'$\omega$')
    ax.set_ylabel('amplitude (arb. units)')
    if xlim is not None:
        ax.set_xlim(*xlim)
    plt.show(fig)


plot_spectrum(opt_result.optimized_controls[0], tlist, xlim=(0, 40))
```



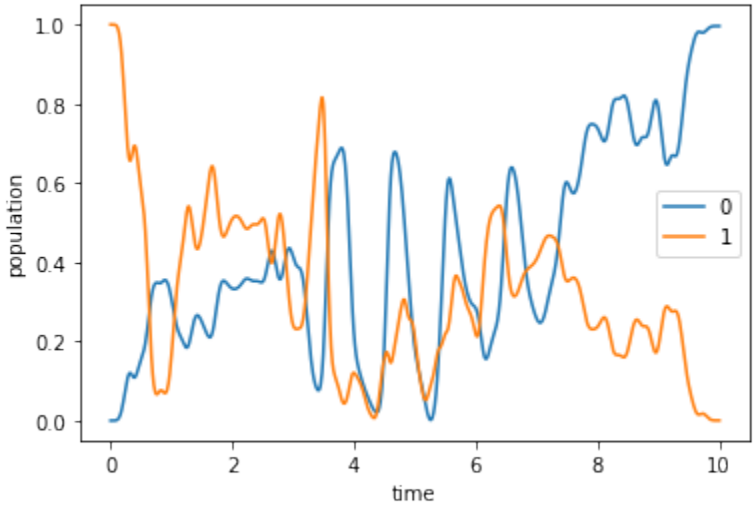
Lastly, we verify that the pulse produces the desired dynamics $|0_l\rangle \rightarrow |1_l\rangle$ and $|1_l\rangle \rightarrow |0_l\rangle$:

[21]:
```
opt_dynamics = [
    opt_result.optimized_objectives[x].mesolve(tlist, e_ops=[proj0, proj1])
    for x in [0, 1]
]
```

[22]:
```
plot_population(opt_dynamics[0])
```



[23]:
```
plot_population(opt_dynamics[1])
```



Since the optimized pulse shows some oscillations (cf. the spectrum above), it is a good idea to check for any discretization error. To this end, we also propagate the optimization result using the same propagator that was used in the optimization (instead of `qutip.mesolve`). The main difference between the two propagations is that `mesolve` assumes piecewise constant pulses that switch between two points in `tlist`, whereas `propagate` assumes that pulses are constant on the intervals of `tlist`, and thus switches *on* the points in `tlist`.

[24]:
```
opt_dynamics2 = [
    opt_result.optimized_objectives[x].propagate(
        tlist, e_ops=[proj0, proj1], propagator=krotov.propagators.expm
    )
    for x in [0, 1]
]
```

The difference between the two propagations gives an indication of the error due to the choice of the piecewise constant time discretization. If this error were unacceptably large, we would need a smaller time step.

[25]:
```
# NBVAL_IGNORE_OUTPUT
# Note: the particular error value may depend on the version of QuTiP
print(
    "Time discretization error = %.1e" %
    abs(opt_dynamics2[0].expect[1][-1] - opt_dynamics[0].expect[1][-1])
)
```

```
Time discretization error = 2.9e-05
```

9.6 Optimization of a Dissipative Quantum Gate

[1]:
```
# NBVAL_IGNORE_OUTPUT
%load_ext watermark
import os
import qutip
import numpy as np
import scipy
```

(continues on next page)

(continued from previous page)

```
import matplotlib
import matplotlib.pylab as plt
import krotov
import copy
from functools import partial
from itertools import product
%watermark -v --iversions
```

```
numpy            1.17.2
qutip            4.4.1
matplotlib.pylab 1.17.2
scipy            1.3.1
krotov           0.4.1
matplotlib       3.1.1
CPython 3.7.3
IPython 7.8.0
```

This example illustrates the optimization for a quantum gate in an open quantum system, where the dynamics is governed by the Liouville-von Neumann equation. A naive extension of a gate optimization to Liouville space would seem to imply that it is necessary to optimize over the full basis of Liouville space (16 matrices, for a two-qubit gate). However, Goerz et al., New J. Phys. 16, 055012 (2014) showed that is not necessary, but that a set of 3 density matrices is sufficient to track the optimization.

This example reproduces the "Example II" from that paper, considering the optimization towards a $\sqrt{\text{iSWAP}}$ two-qubit gate on a system of two transmons with a shared transmission line resonator.

Note: This notebook uses some parallelization features (`qutip.parallel_map`/`multiprocessing.Pool`). Unfortunately, on Windows, `multiprocessing.Pool` does not work correctly for functions defined in a Jupyter notebook (due to the `spawn method <https://docs.python.org/3/library/multiprocessing.html#contexts-and-start-methods>`__ being used on Windows, instead of Unix-`fork`, see also https://stackoverflow.com/questions/45719956). We therefore replace `parallel_map` with `serial_map` when running on Windows.

[2]:
```
import sys
if sys.platform == 'win32':
    from qutip import serial_map as parallel_map
else:
    from qutip import parallel_map
```

9.6.1 The two-transmon system

We consider the Hamiltonian from Eq (17) in the paper, in the rotating wave approximation, together with spontaneous decay and dephasing of each qubit. Alltogether, we define the Liouvillian as follows:

[3]:
```
def two_qubit_transmon_liouvillian(
    ω1, ω2, ωd, δ1, δ2, J, q1T1, q2T1, q1T2, q2T2, T, Omega, n_qubit
):
    from qutip import tensor, identity, destroy

    b1 = tensor(identity(n_qubit), destroy(n_qubit))
    b2 = tensor(destroy(n_qubit), identity(n_qubit))
```

(continues on next page)

(continued from previous page)

```
    H0 = (
        (ω1 - ωd - δ1 / 2) * b1.dag() * b1
        + (δ1 / 2) * b1.dag() * b1 * b1.dag() * b1
        + (ω2 - ωd - δ2 / 2) * b2.dag() * b2
        + (δ2 / 2) * b2.dag() * b2 * b2.dag() * b2
        + J * (b1.dag() * b2 + b1 * b2.dag())
    )

    H1_re = 0.5 * (b1 + b1.dag() + b2 + b2.dag())  # 0.5 is due to RWA
    H1_im = 0.5j * (b1.dag() - b1 + b2.dag() - b2)

    H = [H0, [H1_re, Omega], [H1_im, ZeroPulse]]

    A1 = np.sqrt(1 / q1T1) * b1  # decay of qubit 1
    A2 = np.sqrt(1 / q2T1) * b2  # decay of qubit 2
    A3 = np.sqrt(1 / q1T2) * b1.dag() * b1  # dephasing of qubit 1
    A4 = np.sqrt(1 / q2T2) * b2.dag() * b2  # dephasing of qubit 2

    L = krotov.objectives.liouvillian(H, c_ops=[A1, A2, A3, A4])
    return L
```

We will use internal units GHz and ns. Values in GHz contain an implicit factor 2π, and MHz and µs are converted to GHz and ns, respectively:

[4]:
```
GHz = 2 * np.pi
MHz = 1e-3 * GHz
ns = 1
µs = 1000 * ns
```

This implicit factor 2π is because frequencies (ν) convert to energies as $E = h\nu$, but our propagation routines assume a unit $\hbar = 1$ for energies. Thus, the factor $h/\hbar = 2\pi$.

We will use the same parameters as those given in Table 2 of the paper:

[5]:
```
ω1 = 4.3796 * GHz  # qubit frequency 1
ω2 = 4.6137 * GHz  # qubit frequency 2
ωd = 4.4985 * GHz  # drive frequency
δ1 = -239.3 * MHz  # anharmonicity 1
δ2 = -242.8 * MHz  # anharmonicity 2
J = -2.3 * MHz     # effective qubit-qubit coupling
q1T1 = 38.0 * µs   # decay time for qubit 1
q2T1 = 32.0 * µs   # decay time for qubit 2
q1T2 = 29.5 * µs   # dephasing time for qubit 1
q2T2 = 16.0 * µs   # dephasing time for qubit 2
T = 400 * ns       # gate duration
```

[6]:
```
tlist = np.linspace(0, T, 2000)
```

While in the original paper, each transmon was cut off at 6 levels, here we truncate at 5 levels. This makes the propagation faster, while potentially introducing a slightly larger truncation error.

[7]:
```
n_qubit = 5        # number of transmon levels to consider
```

In the Liouvillian, note the control being split up into a separate real and imaginary part. As a guess control we use a real-valued constant pulse with an amplitude of 35 MHz, acting over

400 ns, with a switch-on and switch-off in the first 20 ns (see plot below)

[8]:
```
def Omega(t, args):
    E0 = 35.0 * MHz
    return E0 * krotov.shapes.flattop(t, 0, T, t_rise=(20 * ns), func='sinsq')
```

The imaginary part start out as zero:

[9]:
```
def ZeroPulse(t, args):
    return 0.0
```

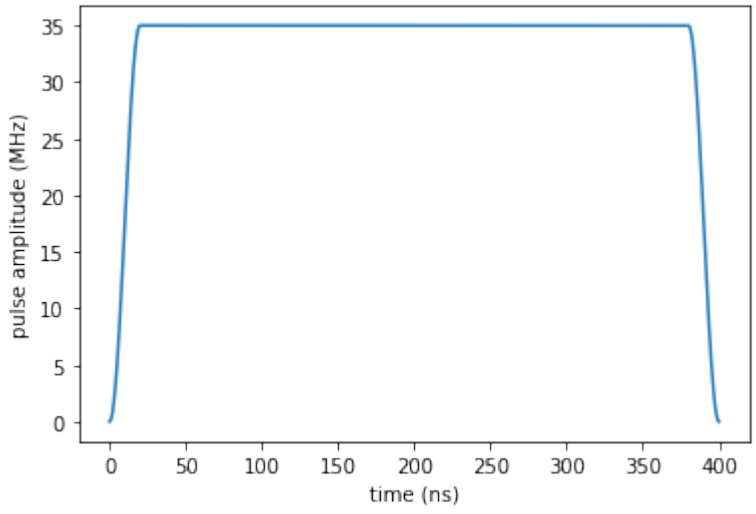
We can now instantiate the Liouvillian:

[10]:
```
L = two_qubit_transmon_liouvillian(
    ω1, ω2, ωd, δ1, δ2, J, q1T1, q2T1, q1T2, q2T2, T, Omega, n_qubit
)
```

The guess pulse looks as follows:

[11]:
```
def plot_pulse(pulse, tlist, xlimit=None):
    fig, ax = plt.subplots()
    if callable(pulse):
        pulse = np.array([pulse(t, None) for t in tlist])
    ax.plot(tlist, pulse/MHz)
    ax.set_xlabel('time (ns)')
    ax.set_ylabel('pulse amplitude (MHz)')
    if xlimit is not None:
        ax.set_xlim(xlimit)
    plt.show(fig)
```

[12]:
```
plot_pulse(L[1][1], tlist)
```



9.6.2 Optimization objectives

Our target gate is $\hat{O} = \sqrt{\mathrm{iSWAP}}$:

[13]:
```
gate = qutip.gates.sqrtiswap()
```

[14]:
```
# NBVAL_IGNORE_OUTPUT
gate
```

[14]:
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper, isherm = False

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.707 & 0.707j & 0.0 \\ 0.0 & 0.707j & 0.707 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

The key idea explored in the paper is that a set of three density matrices is sufficient to track the optimization

$$\hat{\rho}_1 = \sum_{i=1}^{d} \frac{2(d-i+1)}{d(d+1)} |i\rangle\langle i|$$

$$\hat{\rho}_2 = \sum_{i,j=1}^{d} \frac{1}{d} |i\rangle\langle j|$$

$$\hat{\rho}_3 = \sum_{i=1}^{d} \frac{1}{d} |i\rangle\langle i|$$

In our case, $d = 4$ for a two qubit-gate, and the $|i\rangle$, $|j\rangle$ are the canonical basis states $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$

[15]:
```
ket00 = qutip.ket((0, 0), dim=(n_qubit, n_qubit))
ket01 = qutip.ket((0, 1), dim=(n_qubit, n_qubit))
ket10 = qutip.ket((1, 0), dim=(n_qubit, n_qubit))
ket11 = qutip.ket((1, 1), dim=(n_qubit, n_qubit))
basis = [ket00, ket01, ket10, ket11]
```

The three density matrices play different roles in the optimization, and, as shown in the paper, convergence may improve significantly by weighing the states relatively to each other. For this example, we place a strong emphasis on the optimization $\hat{\rho}_1 \to \hat{O}^\dagger \hat{\rho}_1 \hat{O}$, by a factor of 20. This reflects that the hardest part of the optimization is identifying the basis in which the gate is diagonal. We will be using the real-part functional ($J_{T,\mathrm{re}}$) to evaluate the success of $\hat{\rho}_i \to \hat{O}\hat{\rho}_i\hat{O}^\dagger$. Because $\hat{\rho}_1$ and $\hat{\rho}_3$ are mixed states, the Hilbert-Schmidt overlap will take values smaller than one in the optimal case. To compensate, we divide the weights by the purity of the respective states.

[16]:
```
weights = np.array([20, 1, 1], dtype=np.float64)
weights *= len(weights) / np.sum(weights) # manual normalization
weights /= np.array([0.3, 1.0, 0.25]) # purities
```

The `krotov.gate_objectives` routine can initialize the density matrices $\hat{\rho}_1$, $\hat{\rho}_2$, $\hat{\rho}_3$ automatically, via the parameter `liouville_states_set`. Alternatively, we could also use the `'full'` basis of 16 matrices or the extended set of $d + 1 = 5$ pure-state density matrices.

[17]:
```
objectives = krotov.gate_objectives(
    basis,
    gate,
    L,
    liouville_states_set='3states',
```

(continues on next page)

(continued from previous page)

```
    weights=weights,
    normalize_weights=False,
)
objectives
```

[17]:
```
[Objective[ρ₀[5⊗5,5⊗5] to ρ₁[5⊗5,5⊗5] via [₀[[5⊗5,5⊗5],[5⊗5,5⊗5]], [₁[[5⊗5,5⊗5],[5⊗5,
↪5⊗5]], u₁(t)], [₂[[5⊗5,5⊗5],[5⊗5,5⊗5]], u₂(t)]]],
 Objective[ρ₂[5⊗5,5⊗5] to ρ₃[5⊗5,5⊗5] via [₀[[5⊗5,5⊗5],[5⊗5,5⊗5]], [₁[[5⊗5,5⊗5],[5⊗5,
↪5⊗5]], u₁(t)], [₂[[5⊗5,5⊗5],[5⊗5,5⊗5]], u₂(t)]]],
 Objective[ρ₄[5⊗5,5⊗5] to ρ₅[5⊗5,5⊗5] via [₀[[5⊗5,5⊗5],[5⊗5,5⊗5]], [₁[[5⊗5,5⊗5],[5⊗5,
↪5⊗5]], u₁(t)], [₂[[5⊗5,5⊗5],[5⊗5,5⊗5]], u₂(t)]]]]
```

The use of `normalize_weights=False` is because we have included the purities in the weights, as discussed above.

9.6.3 Dynamics under the Guess Pulse

For numerical efficiency, both for the analysis of the guess/optimized controls, we will use a stateful density matrix propagator:

A true physical measure for the success of the optimization is the "average gate fidelity". Evaluating the fidelity requires to simulate the dynamics of the full basis of Liouville space:

[18]:
```
full_liouville_basis = [psi * phi.dag() for (psi, phi) in product(basis, basis)]
```

We propagate these under the guess control:

[19]:
```
def propagate_guess(initial_state):
    return objectives[0].mesolve(
        tlist,
        rho0=initial_state,
    ).states[-1]
```

[20]:
```
full_states_T = parallel_map(
    propagate_guess, values=full_liouville_basis,
)
```

[21]:
```
print("F_avg = %.3f" % krotov.functionals.F_avg(full_states_T, basis, gate))
```

```
F_avg = 0.344
```

Note that we use $F_{T,\mathrm{re}}$, not F_{avg} to steer the optimization, as the Krotov boundary condition $\frac{\partial F_{\mathrm{avg}}}{\partial \rho^\dagger}$ would be non-trivial.

Before doing the optimization, we can look the population dynamics under the guess pulse. For this purpose we propagate the pure-state density matrices corresponding to the canonical logical basis in Hilbert space, and obtain the expectation values for the projection onto these same states:

[22]:
```
rho00, rho01, rho10, rho11 = [qutip.ket2dm(psi) for psi in basis]
```

[23]:
```
def propagate_guess_for_expvals(initial_state):
    return objectives[0].propagate(
        tlist,
```

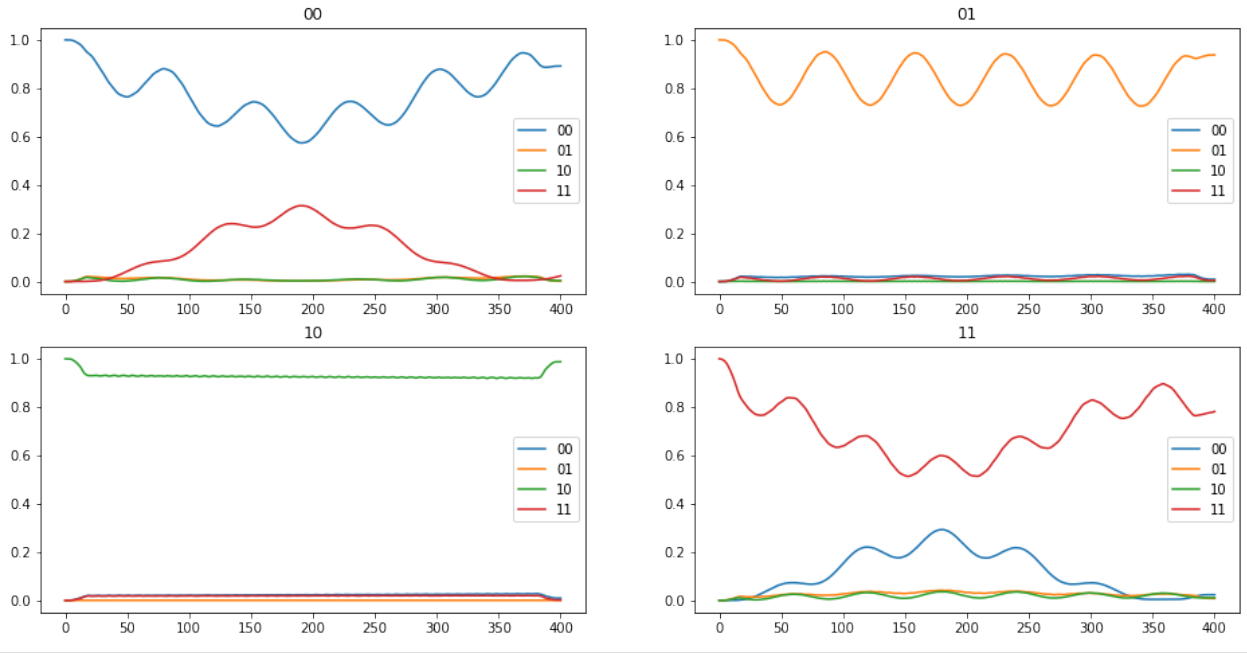
(continues on next page)

(continued from previous page)

```
        propagator=krotov.propagators.DensityMatrixODEPropagator(),
        rho0=initial_state,
        e_ops=[rho00, rho01, rho10, rho11]
    )
```

[24]:
```
def plot_population_dynamics(dyn00, dyn01, dyn10, dyn11):
    fig, axs = plt.subplots(ncols=2, nrows=2, figsize=(16, 8))
    axs = np.ndarray.flatten(axs)
    labels = ['00', '01', '10', '11']
    dyns = [dyn00, dyn01, dyn10, dyn11]
    for (ax, dyn, title) in zip(axs, dyns, labels):
        for (i, label) in enumerate(labels):
            ax.plot(dyn.times, dyn.expect[i], label=label)
        ax.legend()
        ax.set_title(title)
    plt.show(fig)
```

[25]:
```
plot_population_dynamics(
    *parallel_map(
        propagate_guess_for_expvals,
        values=[rho00, rho01, rho10, rho11],
    )
)
```



9.6.4 Optimization

We now define the optimization parameters for the controls, the Krotov step size λ_a and the update-shape that will ensure that the pulse switch-on and switch-off stays intact.

[26]:
```
pulse_options = {
    L[i][1]: dict(
```

(continues on next page)

(continued from previous page)

```
        lambda_a=1.0,
        update_shape=partial(
            krotov.shapes.flattop, t_start=0, t_stop=T, t_rise=(20 * ns))
        )
    for i in [1, 2]
}
```

Then we run the optimization for 2000 iterations

[27]:
```
opt_result = krotov.optimize_pulses(
    objectives,
    pulse_options,
    tlist,
    propagator=krotov.propagators.DensityMatrixODEPropagator(reentrant=True),
    chi_constructor=krotov.functionals.chis_re,
    info_hook=krotov.info_hooks.print_table(J_T=krotov.functionals.J_T_re),
    iter_stop=3,
)
```

```
  iter.        J_T   ∑∫gₐ(t)dt          J       ΔJ_T         ΔJ  secs
     0   1.22e-01    0.00e+00   1.22e-01        n/a        n/a     8
     1   7.49e-02    2.26e-02   9.75e-02  -4.67e-02  -2.41e-02    31
     2   7.41e-02    3.98e-04   7.45e-02  -8.12e-04  -4.14e-04    33
     3   7.33e-02    3.70e-04   7.37e-02  -7.55e-04  -3.85e-04    29
```

(this takes a while)...

[28]:
```
dumpfile = "./3states_opt_result.dump"
if os.path.isfile(dumpfile):
    opt_result = krotov.result.Result.load(dumpfile, objectives)
else:
    opt_result = krotov.optimize_pulses(
        objectives,
        pulse_options,
        tlist,
        propagator=krotov.propagators.DensityMatrixODEPropagator(reentrant=True),
        chi_constructor=krotov.functionals.chis_re,
        info_hook=krotov.info_hooks.print_table(J_T=krotov.functionals.J_T_re),
        iter_stop=5,
        continue_from=opt_result
    )
    opt_result.dump(dumpfile)
```

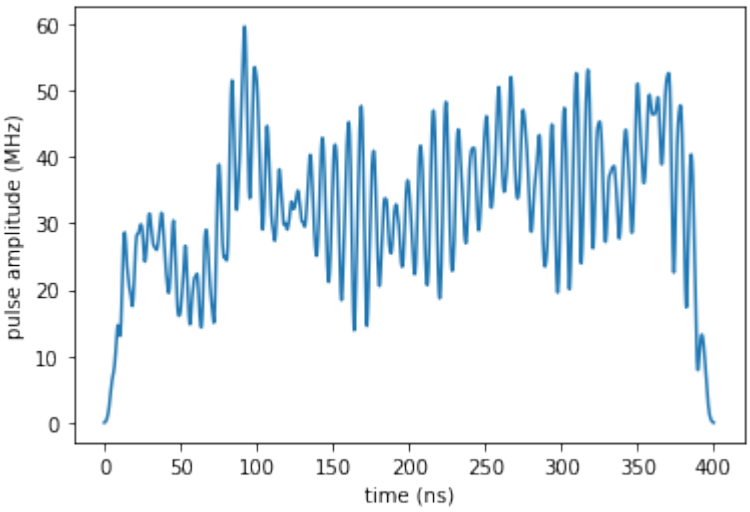
[29]:
```
opt_result
```

[29]:
```
Krotov Optimization Result
--------------------------
- Started at 2019-02-25 00:43:31
- Number of objectives: 3
- Number of iterations: 2000
- Reason for termination: Reached 2000 iterations
- Ended at 2019-02-25 23:19:34
```

9.6.5 Optimization result

[30]:
```
optimized_control = opt_result.optimized_controls[0] + 1j * opt_result.optimized_
↪controls[1]
```

[31]:
```
plot_pulse(np.abs(optimized_control), tlist)
```



[32]:
```
def propagate_opt(initial_state):
    return opt_result.optimized_objectives[0].propagate(
        tlist,
        propagator=krotov.propagators.DensityMatrixODEPropagator(),
        rho0=initial_state,
    ).states[-1]
```

[33]:
```
opt_full_states_T = parallel_map(
    propagate_opt, values=full_liouville_basis,
)
```

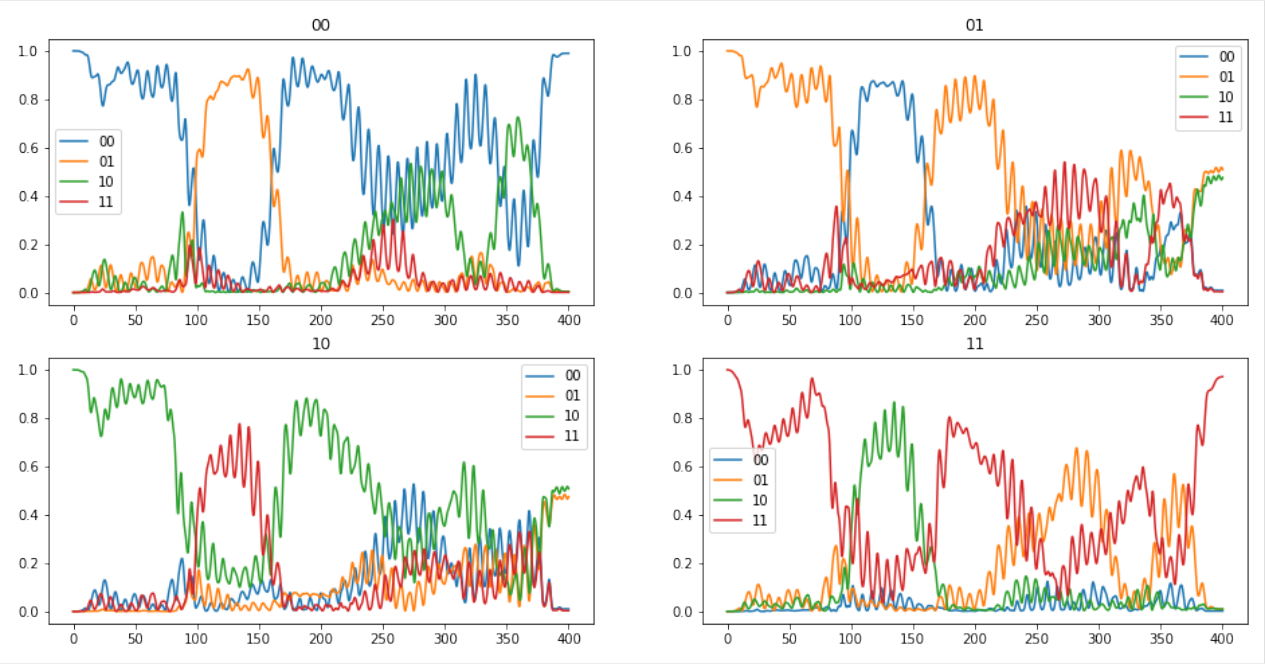
[34]:
```
print("F_avg = %.3f" % krotov.functionals.F_avg(opt_full_states_T, basis, gate))
```

```
F_avg = 0.977
```

[35]:
```
def propagate_opt_for_expvals(initial_state):
    return opt_result.optimized_objectives[0].propagate(
        tlist,
        propagator=krotov.propagators.DensityMatrixODEPropagator(),
        rho0=initial_state,
        e_ops=[rho00, rho01, rho10, rho11]
    )
```

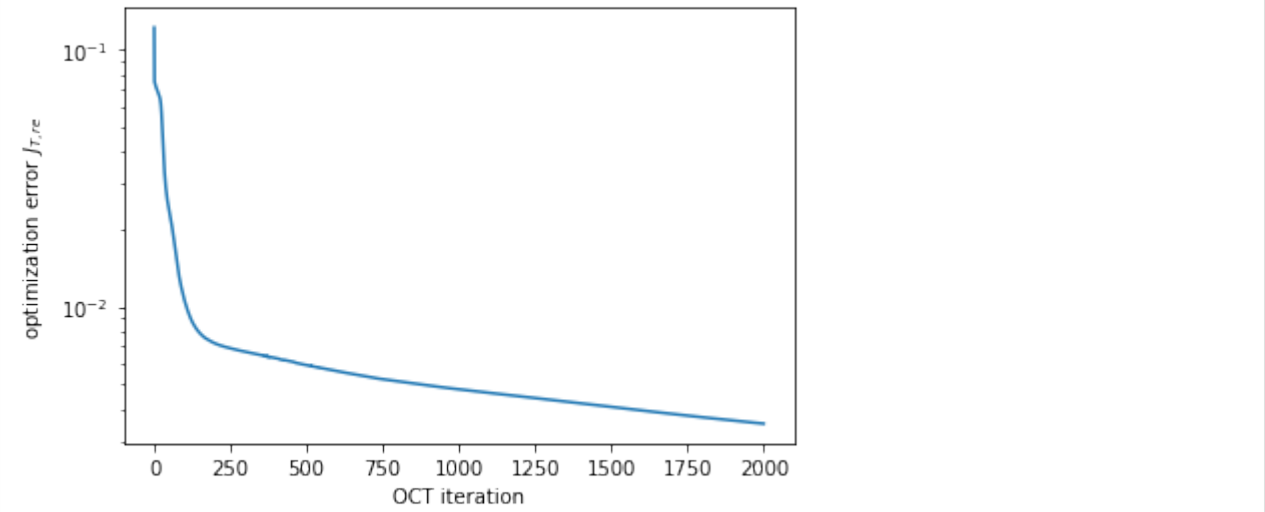
Plotting the population dynamics, we see the expected behavior for the $\sqrt{\text{iSWAP}}$ gate.

[36]:
```
plot_population_dynamics(
    *parallel_map(
        propagate_opt_for_expvals,
        values=[rho00, rho01, rho10, rho11],
    )
)
```



[37]:
```
def plot_convergence(result):
    fig, ax = plt.subplots()
    ax.semilogy(result.iters, result.info_vals)
    ax.set_xlabel('OCT iteration')
    ax.set_ylabel(r'optimization error $J_{T, re}$')
    plt.show(fig)
```

[38]:
```
plot_convergence(opt_result)
```



9.7 Optimization towards a Perfect Entangler

[1]:
```
# NBVAL_IGNORE_OUTPUT
%load_ext watermark
import qutip
import numpy as np
import scipy
import matplotlib
import matplotlib.pylab as plt
import krotov
from IPython.display import display
import weylchamber as wc
from weylchamber.visualize import WeylChamber
from weylchamber.coordinates import from_magic

%watermark -v --iversions
```

```
weylchamber      0.3.2
numpy            1.17.2
krotov           0.4.1
matplotlib.pylab 1.17.2
matplotlib       3.1.1
scipy            1.3.1
qutip            4.4.1
CPython 3.7.3
IPython 7.8.0
```

This example demonstrates the optimization with an "unconventional" optimization target. Instead of a state-to-state transition, or the realization of a specific quantum gate, we optimize for an arbitrary perfectly entangling gate. See

- P. Watts, et al., Phys. Rev. A 91, 062306 (2015)
- M. H. Goerz, et al., Phys. Rev. A 91, 062307 (2015)

for details.

9.7.1 Hamiltonian

We consider a generic two-qubit Hamiltonian (motivated from the example of two superconducting transmon qubits, truncated to the logical subspace),

$$\hat{H}(t) = -\frac{\omega_1}{2}\hat{\sigma}_z^{(1)} - \frac{\omega_2}{2}\hat{\sigma}_z^{(2)} + 2J\left(\hat{\sigma}_x^{(1)}\hat{\sigma}_x^{(2)} + \hat{\sigma}_y^{(1)}\hat{\sigma}_y^{(2)}\right) + u(t)\left(\hat{\sigma}_x^{(1)} + \lambda\hat{\sigma}_x^{(2)}\right),$$

where ω_1 and ω_2 are the energy level splitting of the respective qubit, J is the effective coupling strength and $u(t)$ is the control field. λ defines the strength of the qubit-control coupling for qubit 2, relative to qubit 1.

We use the following parameters:

[2]:
```
w1 = 1.1  # qubit 1 level splitting
w2 = 2.1  # qubit 2 level splitting
J = 0.2  # effective qubit coupling
u0 = 0.3  # initial driving strength
la = 1.1  # relative pulse coupling strength of second qubit
T = 25.0  # final time
```

(continues on next page)

(continued from previous page)

```
nt = 250  # number of time steps

tlist = np.linspace(0, T, nt)
```

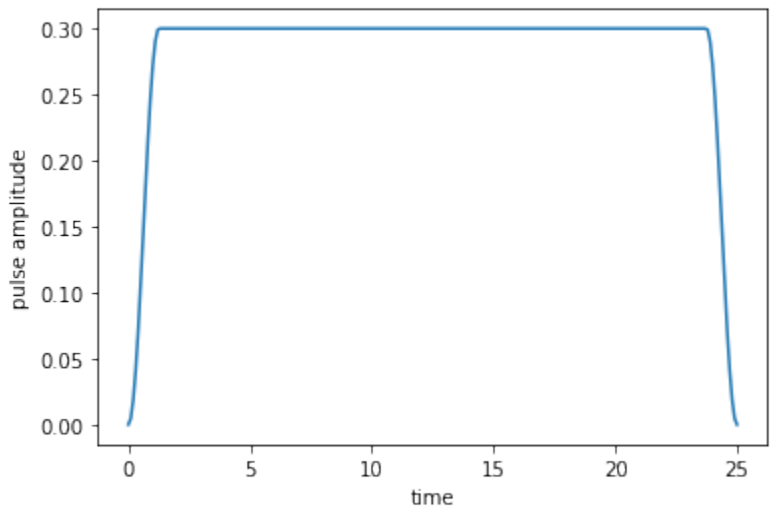
These are for illustrative purposes only, and do not correspond to any particular physical system.

The initial guess is defined as

[3]:
```
def eps0(t, args):
    return u0 * krotov.shapes.flattop(
        t, t_start=0, t_stop=T, t_rise=(T / 20), t_fall=(T / 20), func='sinsq'
    )
```

[4]:
```
def plot_pulse(pulse, tlist):
    fig, ax = plt.subplots()
    if callable(pulse):
        pulse = np.array([pulse(t, args=None) for t in tlist])
    ax.plot(tlist, pulse)
    ax.set_xlabel('time')
    ax.set_ylabel('pulse amplitude')
    plt.show(fig)
```

[5]:
```
plot_pulse(eps0, tlist)
```



We instantiate the Hamiltonian with this guess pulse

[6]:
```
def hamiltonian(w1=w1, w2=w2, J=J, la=la, u0=u0):
    """Two qubit Hamiltonian

    Args:
        w1 (float): energy separation of the first qubit levels
        w2 (float): energy separation of the second qubit levels
        J (float): effective coupling between both qubits
        la (float): factor that pulse coupling strength differs for second qubit
        u0 (float): constant amplitude of the driving field
    """
```

(continues on next page)

(continued from previous page)

```
    # local qubit Hamiltonians
    Hq1 = 0.5 * w1 * np.diag([-1, 1])
    Hq2 = 0.5 * w2 * np.diag([-1, 1])

    # lift Hamiltonians to joint system operators
    H0 = np.kron(Hq1, np.identity(2)) + np.kron(np.identity(2), Hq2)

    # define the interaction Hamiltonian
    sig_x = np.array([[0, 1], [1, 0]])
    sig_y = np.array([[0, -1j], [1j, 0]])
    Hint = 2 * J * (np.kron(sig_x, sig_x) + np.kron(sig_y, sig_y))
    H0 = H0 + Hint

    # define the drive Hamiltonian
    H1 = np.kron(np.array([[0, 1], [1, 0]]), np.identity(2)) + la * np.kron(
        np.identity(2), np.array([[0, 1], [1, 0]])
    )

    # convert Hamiltonians to QuTiP objects
    H0 = qutip.Qobj(H0)
    H1 = qutip.Qobj(H1)

    return [H0, [H1, eps0]]


H = hamiltonian(w1=w1, w2=w2, J=J, la=la, u0=u0)
```

As well as the canonical two-qubit logical basis,

[7]:
```
psi_00 = qutip.Qobj(np.kron(np.array([1, 0]), np.array([1, 0])))
psi_01 = qutip.Qobj(np.kron(np.array([1, 0]), np.array([0, 1])))
psi_10 = qutip.Qobj(np.kron(np.array([0, 1]), np.array([1, 0])))
psi_11 = qutip.Qobj(np.kron(np.array([0, 1]), np.array([0, 1])))
```

with the corresponding projectors to calculate population dynamics below.

[8]:
```
proj_00 = qutip.ket2dm(psi_00)
proj_01 = qutip.ket2dm(psi_01)
proj_10 = qutip.ket2dm(psi_10)
proj_11 = qutip.ket2dm(psi_11)
```

9.7.2 Objectives for a perfect entangler

Our optimization target is the closest perfectly entangling gate, quantified by the perfect-entangler functional

$$F_{PE} = g_3 \sqrt{g_1^2 + g_2^2} - g_1,$$

where g_1, g_2, g_3 are the local invariants of the implemented gate that uniquely identify its non-local content. The local invariants are closely related to the Weyl coordinates c_1, c_2, c_3, which provide a useful geometric visualization in the Weyl chamber. The perfectly entangling gates lie within a polyhedron in the Weyl chamber and F_{PE} becomes zero at its boundaries. We define $F_{PE} \equiv 0$ for *all* perfect entanglers (inside the polyhedron)

A list of four objectives that encode the minimization of F_{PE} are generated by calling the `gate_objectives` function with the canonical basis, and `"PE"` as target "gate".

[9]:
```
objectives = krotov.gate_objectives(
    basis_states=[psi_00, psi_01, psi_10, psi_11], gate="PE", H=H
)
```

[10]:
```
objectives
```

[10]:
```
[Objective[|Ψ₀(4)⟩ to PE via [H₀[4,4], [H₁[4,4], u₁(t)]]],
 Objective[|Ψ₁(4)⟩ to PE via [H₀[4,4], [H₁[4,4], u₁(t)]]],
 Objective[|Ψ₂(4)⟩ to PE via [H₀[4,4], [H₁[4,4], u₁(t)]]],
 Objective[|Ψ₃(4)⟩ to PE via [H₀[4,4], [H₁[4,4], u₁(t)]]]]
```

The initial states in these objectives are not the canonical basis states, but a Bell basis,

[11]:
```
# NBVAL_IGNORE_OUTPUT
for obj in objectives:
    display(obj.initial_state)
```

Quantum object: dims = [[4], [1]], shape = (4, 1), type = ket

$$\begin{pmatrix} 0.707 \\ 0.0 \\ 0.0 \\ 0.707 \end{pmatrix}$$

Quantum object: dims = [[4], [1]], shape = (4, 1), type = ket

$$\begin{pmatrix} 0.0 \\ 0.707j \\ 0.707j \\ 0.0 \end{pmatrix}$$

Quantum object: dims = [[4], [1]], shape = (4, 1), type = ket

$$\begin{pmatrix} 0.0 \\ 0.707 \\ -0.707 \\ 0.0 \end{pmatrix}$$

Quantum object: dims = [[4], [1]], shape = (4, 1), type = ket

$$\begin{pmatrix} 0.707j \\ 0.0 \\ 0.0 \\ -0.707j \end{pmatrix}$$

Since we don't know *which* perfect entangler the optimization result will implement, we cannot associate any "target state" with each objective, and the `target` attribute is set to the string 'PE'.

We can treat the above objectives as a "black box"; the only important consideration is that the `chi_constructor` that we will pass to `optimize_pulses` to calculating the boundary condition for the backwards propagation,

$$|\chi_k\rangle = \frac{\partial F_{PE}}{\partial \langle\phi_k|}\bigg|_{|\phi_k(T)\rangle},$$

must be consistent with how the `objectives` are set up. For the perfect entanglers functional, the calculation of the $|\chi_k\rangle$ is relatively complicated. The `weylchamber` package (https://github.com/qucontrol/weylchamber) contains a suitable routine that works on the

`objectives` exactly as defined above (specifically, under the assumption that the $|\phi_k\rangle$ are the appropriate Bell states):

[12]:
```
help(wc.perfect_entanglers.make_PE_krotov_chi_constructor)
```

```
Help on function make_PE_krotov_chi_constructor in module weylchamber.
↪perfect_entanglers:

make_PE_krotov_chi_constructor(canonical_basis, unitarity_weight=0)
    Return a constructor for the χ's in a PE optimization.

    Return a `chi_constructor` that determines the boundary condition of the
    backwards propagation in an optimization towards a perfect entangler in
    Krotov's method, based on the foward-propagtion of the Bell states. In
    detail, the function returns a callable function that calculates

    .. math::

        \ket{\chi_{i}}
        =
        \frac{\partial F_{PE}}{\partial \bra{\phi_i}}
        \Bigg|_{\ket{\phi_{i}(T)}}

    for all $i$ with $\ket{\phi_{0}(T)}, …, \ket{\phi_{3}(T)}$ the forward
    propagated Bell states at final time $T$, cf. Eq. (33b) in Ref. [1].
    $F_{PE}$ is the perfect-entangler functional
    :func:`~weylchamber.perfect_entanglers.F_PE`. For the details of the
    derivative see Appendix G in Ref. [2].

    References:

    [1] `M. H. Goerz, et al., Phys. Rev. A 91, 062307 (2015)
    <https://doi.org/10.1103/PhysRevA.91.062307>`_

    [2] `M. H. Goerz, Optimizing Robust Quantum Gates in Open Quantum Systems.
    PhD thesis, University of Kassel, 2015
    <https://michaelgoerz.net/research/diss_goerz.pdf>`_

    Args:
        canonical_basis (list[qutip.Qobj]): A list of four basis states that
            define the canonical basis $\ket{00}$, $\ket{01}$, $\ket{10}$, and
            $\ket{11}$ of the logical subspace.
        unitarity_weight (float): A weight in [0, 1] that determines how much
            emphasis is placed on  maintaining population in the logical
            subspace.

    Returns:
        callable: a function ``chi_constructor(fw_states_T, **kwargs)`` that
        receives the result of a foward propagation of the Bell states
        (obtained from `canonical_basis` via
        :func:`weylchamber.gates.bell_basis`), and returns a list of statex
        $\ket{\chi_{i}}$ that are the boundary condition for the backward
        propagation in Krotov's method. Positional arguments beyond
        `fw_states_T` are ignored.
```

[13]:
```
chi_constructor = wc.perfect_entanglers.make_PE_krotov_chi_constructor(
    [psi_00, psi_01, psi_10, psi_11]
```

(continues on next page)

(continued from previous page)

```
)
```

Again, the key point to take from this is that when defining a new or unusual functional, **the ``chi_constructor`` must be congruent with the way the objectives are defined**. As a user, you can choose whatever definition of objectives and implementation of `chi_constructor` is most suitable, as long they are compatible.

9.7.3 Second Order Update Equation

As the perfect-entangler functional F_{PE} is non-linear in the states, Krotov's method requires the second-order contribution in order to guarantee monotonic convergence (see D. M. Reich, et al., J. Chem. Phys. 136, 104103 (2012) for details). The second order update equation reads

$$\epsilon^{(i+1)}(t) = \epsilon^{ref}(t) + \frac{S(t)}{\lambda_a} \mathrm{Im}\left\{ \sum_{k=1}^{N} \left\langle \chi_k^{(i)}(t) \right| \left. \frac{\partial \hat{H}}{\partial \epsilon} \right|_{\substack{\phi^{(i+1)}(t) \\ \epsilon^{(i+1)}(t)}} \left| \phi_k^{(i+1)}(t) \right\rangle \right.$$
$$\left. + \frac{1}{2}\sigma(t) \left\langle \Delta\phi_k(t) \right| \left. \frac{\partial \hat{H}}{\partial \epsilon} \right|_{\substack{\phi^{(i+1)}(t) \\ \epsilon^{(i+1)}(t)}} \left| \phi_k^{(i+1)}(t) \right\rangle \right\},$$

where the term proportional to $\sigma(t)$ defines the second-order contribution. In order to use the second-order term, we need to pass a function to evaluate this $\sigma(t)$ as `sigma` to `optimize_pulses`. We use the equation

$$\sigma(t) = -\max\left(\varepsilon_A, 2A + \varepsilon_A\right)$$

with ε_A a small non-negative number, and A a parameter that can be recalculated numerically after each iteration (see D. M. Reich, et al., J. Chem. Phys. 136, 104103 (2012) for details).

Generally, $\sigma(t)$ has parametric dependencies like A in this example, which should be refreshed for each iteration. Thus, since `sigma` holds internal state, it must be implemented as an object subclassing from `krotov.second_order.Sigma`:

[14]:
```
class sigma(krotov.second_order.Sigma):
    def __init__(self, A, epsA=0):
        self.A = A
        self.epsA = epsA

    def __call__(self, t):
        ϵ, A = self.epsA, self.A
        return -max(ϵ, 2 * A + ϵ)

    def refresh(
        self,
        forward_states,
        forward_states0,
        chi_states,
        chi_norms,
        optimized_pulses,
        guess_pulses,
        objectives,
        result,
    ):
```

(continues on next page)

(continued from previous page)

```
        try:
            Delta_J_T = result.info_vals[-1][0] - result.info_vals[-2][0]
        except IndexError:  # first iteration
            Delta_J_T = 0
        self.A = krotov.second_order.numerical_estimate_A(
            forward_states, forward_states0, chi_states, chi_norms, Delta_J_T
        )
```

This combines the evaluation of the function, `sigma(t)`, with the recalculation of A (or whatever parametrizations another $\sigma(t)$ function might contain) in `sigma.refresh`, which `optimize_pulses` invokes automatically at the end of each iteration.

9.7.4 Optimization

Before running the optimization, we define the shape function $S(t)$ to maintain the smooth switch-on and switch-off, and the λ_a parameter that determines the overall magnitude of the pulse update in each iteration:

[15]:
```
def S(t):
    """Shape function for the field update"""
    return krotov.shapes.flattop(
        t, t_start=0, t_stop=T, t_rise=T / 20, t_fall=T / 20, func='sinsq'
    )

pulse_options = {H[1][1]: dict(lambda_a=1.0e2, update_shape=S)}
```

In previous examples, we have used `info_hook` routines that display and store the value of the functional J_T. Here, we will also want to analyze the optimization in terms of the Weyl chamber coordinates (c_1, c_2, c_3). We therefore write a custom `print_fidelity` routine that prints F_{PE} as well as the gate concurrence (as an alternative measure for the entangling power of quantum gates), and results in the storage of a nested tuple `(F_PE, (c1, c2, c3))` for each iteration, in `Result.info_vals`.

[16]:
```
def print_fidelity(**args):
    basis = [objectives[i].initial_state for i in [0, 1, 2, 3]]
    states = [args['fw_states_T'][i] for i in [0, 1, 2, 3]]
    U = wc.gates.gate(basis, states)
    c1, c2, c3 = wc.coordinates.c1c2c3(from_magic(U))
    g1, g2, g3 = wc.local_invariants.g1g2g3_from_c1c2c3(c1, c2, c3)
    conc = wc.perfect_entanglers.concurrence(c1, c2, c3)
    F_PE = wc.perfect_entanglers.F_PE(g1, g2, g3)
    print("    F_PE: %f\n    gate conc.: %f" % (F_PE, conc))
    return F_PE, [c1, c2, c3]
```

This structure must be taken into account in a `check_convergence` routine. This would affect routines like `krotov.convergence.value_below` that assume that `Result.info_vals` contains the values of J_T only. Here, we define a check that stops the optimization as soon as we reach a perfect entangler:

[17]:
```
def check_PE(result):
    # extract F_PE from (F_PE, [c1, c2, c3])
    F_PE = result.info_vals[-1][0]
    if F_PE <= 0:
        return "achieved perfect entangler"
```

(continues on next page)

(continued from previous page)

```
    else:
        return None
```

[18]:
```
opt_result = krotov.optimize_pulses(
    objectives,
    pulse_options=pulse_options,
    tlist=tlist,
    propagator=krotov.propagators.expm,
    chi_constructor=chi_constructor,
    info_hook=krotov.info_hooks.chain(
        krotov.info_hooks.print_debug_information, print_fidelity
    ),
    check_convergence=check_PE,
    sigma=sigma(A=0.0),
    iter_stop=20,
)
```

```
Iteration 0
    objectives:
        1:|Ψ₀(4)) to PE via [H₀[4,4], [H₁[4,4], u₁(t)]]
        2:|Ψ₁(4)) to PE via [H₀[4,4], [H₁[4,4], u₁(t)]]
        3:|Ψ₂(4)) to PE via [H₀[4,4], [H₁[4,4], u₁(t)]]
        4:|Ψ₃(4)) to PE via [H₀[4,4], [H₁[4,4], u₁(t)]]
    adjoint objectives:
        1:(Ψ₀(4)| to PE via [H₂[4,4], [H₃[4,4], u₁(t)]]
        2:(Ψ₁(4)| to PE via [H₄[4,4], [H₅[4,4], u₁(t)]]
        3:(Ψ₂(4)| to PE via [H₆[4,4], [H₇[4,4], u₁(t)]]
        4:(Ψ₃(4)| to PE via [H₈[4,4], [H₉[4,4], u₁(t)]]
    S(t) (ranges): [0.000000, 1.000000]
    duration: 1.5 secs (started at 2019-10-11 12:26:08)
    optimized pulses (ranges): [0.00, 0.30]
    ∫gₐ(t)dt: 0.00e+00
    λₐ: 1.00e+02
    storage (bw, fw, fw0): None, [4 * ndarray(250)] (0.5 MB), [4 * ndarray(250)] (0.5␣
↪MB)
    fw_states_T norm: 1.000000, 1.000000, 1.000000, 1.000000
    F_PE: 1.447666
    gate conc.: 0.479571
Iteration 1
    duration: 4.4 secs (started at 2019-10-11 12:26:10)
    optimized pulses (ranges): [0.00, 0.32]
    ∫gₐ(t)dt: 2.56e-03
    λₐ: 1.00e+02
    storage (bw, fw, fw0): [4 * ndarray(250)] (0.5 MB), [4 * ndarray(250)] (0.5 MB),␣
↪[4 * ndarray(250)] (0.5 MB)
    fw_states_T norm: 1.000000, 1.000000, 1.000000, 1.000000
    F_PE: 1.000458
    gate conc.: 0.645400
Iteration 2
    duration: 4.3 secs (started at 2019-10-11 12:26:14)
    optimized pulses (ranges): [0.00, 0.34]
    ∫gₐ(t)dt: 2.18e-03
    λₐ: 1.00e+02
    storage (bw, fw, fw0): [4 * ndarray(250)] (0.5 MB), [4 * ndarray(250)] (0.5 MB),␣
↪[4 * ndarray(250)] (0.5 MB)
    fw_states_T norm: 1.000000, 1.000000, 1.000000, 1.000000
    F_PE: 0.587432
```

(continues on next page)

(continued from previous page)

```
    gate conc.: 0.782093
Iteration 3
    duration: 4.2 secs (started at 2019-10-11 12:26:18)
    optimized pulses (ranges): [0.00, 0.36]
    ∫gₐ(t)dt: 1.44e-03
    λₐ: 1.00e+02
    storage (bw, fw, fw0): [4 * ndarray(250)] (0.5 MB), [4 * ndarray(250)] (0.5 MB),␣
↪[4 * ndarray(250)] (0.5 MB)
    fw_states_T norm: 1.000000, 1.000000, 1.000000, 1.000000
    F_PE: 0.309838
    gate conc.: 0.889907
Iteration 4
    duration: 4.2 secs (started at 2019-10-11 12:26:23)
    optimized pulses (ranges): [0.00, 0.37]
    ∫gₐ(t)dt: 7.71e-04
    λₐ: 1.00e+02
    storage (bw, fw, fw0): [4 * ndarray(250)] (0.5 MB), [4 * ndarray(250)] (0.5 MB),␣
↪[4 * ndarray(250)] (0.5 MB)
    fw_states_T norm: 1.000000, 1.000000, 1.000000, 1.000000
    F_PE: 0.158278
    gate conc.: 0.949411
Iteration 5
    duration: 4.7 secs (started at 2019-10-11 12:26:27)
    optimized pulses (ranges): [0.00, 0.38]
    ∫gₐ(t)dt: 3.92e-04
    λₐ: 1.00e+02
    storage (bw, fw, fw0): [4 * ndarray(250)] (0.5 MB), [4 * ndarray(250)] (0.5 MB),␣
↪[4 * ndarray(250)] (0.5 MB)
    fw_states_T norm: 1.000000, 1.000000, 1.000000, 1.000000
    F_PE: 0.079730
    gate conc.: 0.979050
Iteration 6
    duration: 4.8 secs (started at 2019-10-11 12:26:32)
    optimized pulses (ranges): [0.00, 0.38]
    ∫gₐ(t)dt: 2.07e-04
    λₐ: 1.00e+02
    storage (bw, fw, fw0): [4 * ndarray(250)] (0.5 MB), [4 * ndarray(250)] (0.5 MB),␣
↪[4 * ndarray(250)] (0.5 MB)
    fw_states_T norm: 1.000000, 1.000000, 1.000000, 1.000000
    F_PE: 0.037817
    gate conc.: 0.992901
Iteration 7
    duration: 4.4 secs (started at 2019-10-11 12:26:36)
    optimized pulses (ranges): [0.00, 0.39]
    ∫gₐ(t)dt: 1.16e-04
    λₐ: 1.00e+02
    storage (bw, fw, fw0): [4 * ndarray(250)] (0.5 MB), [4 * ndarray(250)] (0.5 MB),␣
↪[4 * ndarray(250)] (0.5 MB)
    fw_states_T norm: 1.000000, 1.000000, 1.000000, 1.000000
    F_PE: 0.014284
    gate conc.: 0.998580
Iteration 8
    duration: 5.5 secs (started at 2019-10-11 12:26:41)
    optimized pulses (ranges): [0.00, 0.39]
    ∫gₐ(t)dt: 6.83e-05
    λₐ: 1.00e+02
```

(continues on next page)

(continued from previous page)

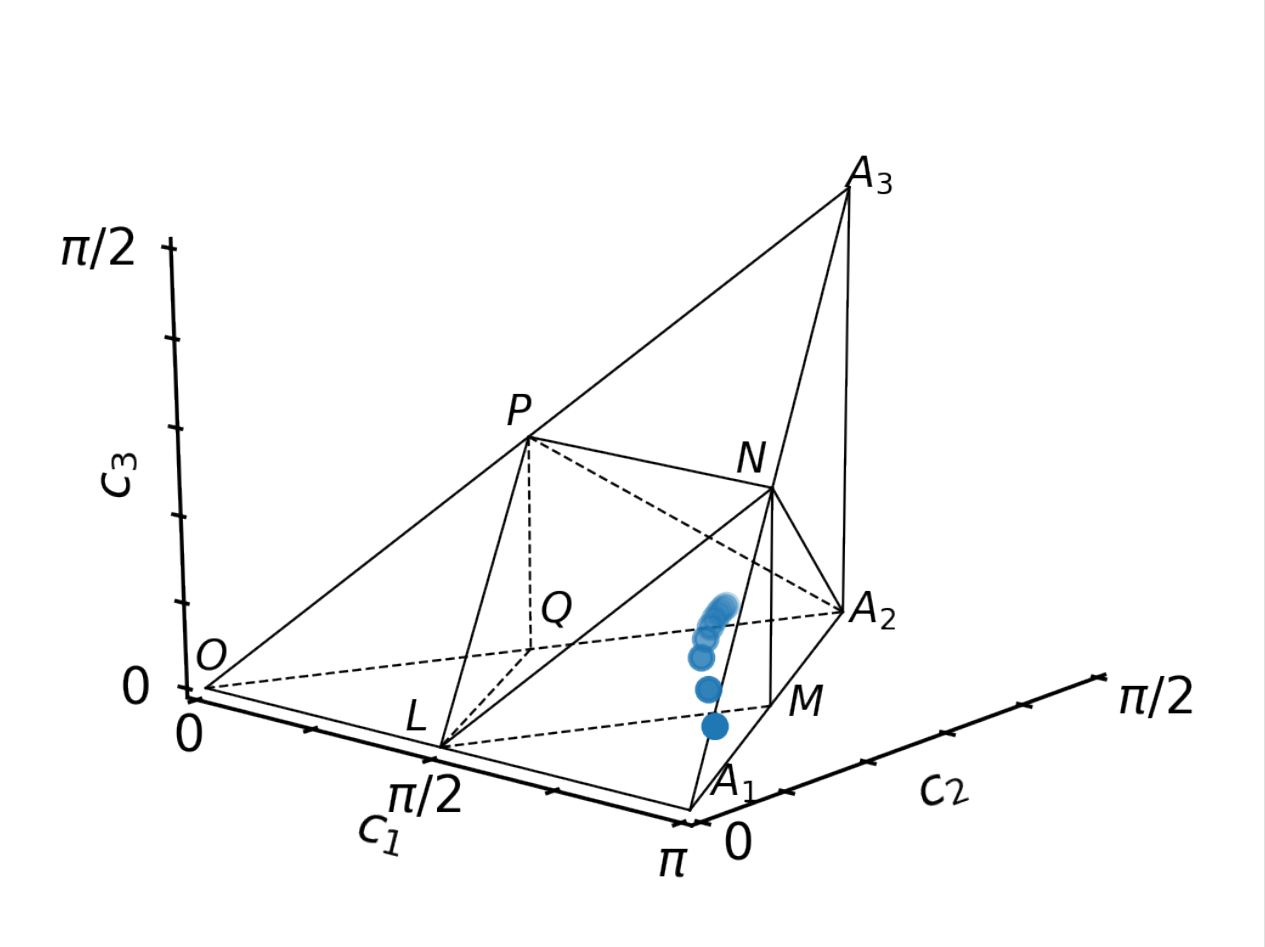
```
    storage (bw, fw, fw0): [4 * ndarray(250)] (0.5 MB), [4 * ndarray(250)] (0.5 MB),␣
↪[4 * ndarray(250)] (0.5 MB)
    fw_states_T norm: 1.000000, 1.000000, 1.000000, 1.000000
    F_PE: 0.000359
    gate conc.: 0.999999
Iteration 9
    duration: 4.6 secs (started at 2019-10-11 12:26:46)
    optimized pulses (ranges): [0.00, 0.39]
    ∫gₐ(t)dt: 4.26e-05
    λₐ: 1.00e+02
    storage (bw, fw, fw0): [4 * ndarray(250)] (0.5 MB), [4 * ndarray(250)] (0.5 MB),␣
↪[4 * ndarray(250)] (0.5 MB)
    fw_states_T norm: 1.000000, 1.000000, 1.000000, 1.000000
    F_PE: -0.008316
    gate conc.: 1.000000
```

[19]:
```
opt_result
```

[19]:
```
Krotov Optimization Result
--------------------------
- Started at 2019-10-11 12:26:08
- Number of objectives: 4
- Number of iterations: 9
- Reason for termination: Reached convergence: achieved perfect entangler
- Ended at 2019-10-11 12:26:51
```

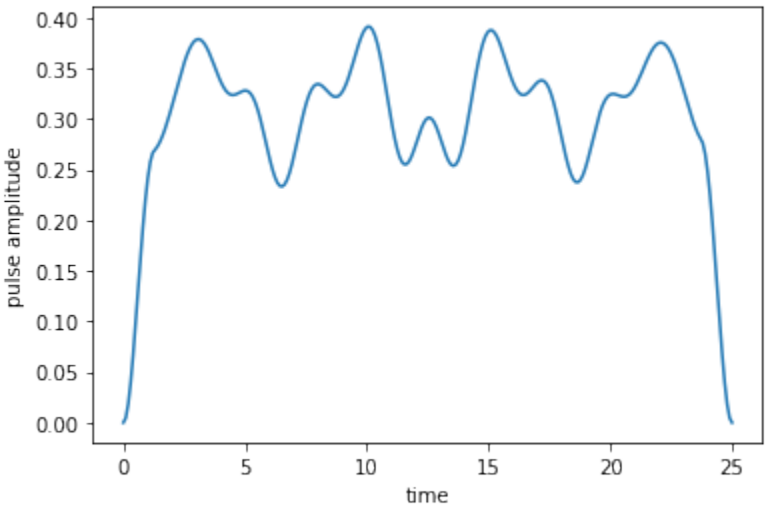
We can visualize how each iteration of the optimization brings the dynamics closer to the polyhedron of perfect entanglers (using the Weyl chamber coordinates that we calculated in the `info_hook` routine `print_fidelity`, and that were stored in `Result.info_vals`).

[20]:
```
w = WeylChamber()
c1c2c3 = [opt_result.info_vals[i][1] for i in range(len(opt_result.iters))]
for i in range(len(opt_result.iters)):
    w.add_point(c1c2c3[i][0], c1c2c3[i][1], c1c2c3[i][2])
w.plot()
```



The final optimized control field looks like this:

[21]:
```
plot_pulse(opt_result.optimized_controls[0], tlist)
```



9.8 Ensemble Optimization for Robust Pulses

[1]:
```
# NBVAL_IGNORE_OUTPUT
%load_ext watermark
import os
import qutip
import numpy as np
import scipy
import matplotlib
import matplotlib.pylab as plt
import krotov
from qutip import Qobj
import pickle

%watermark -v --iversions
```

```
krotov           0.4.1
matplotlib.pylab 1.17.2
matplotlib       3.1.1
numpy            1.17.2
qutip            4.4.1
scipy            1.3.1
CPython 3.7.3
IPython 7.8.0
```

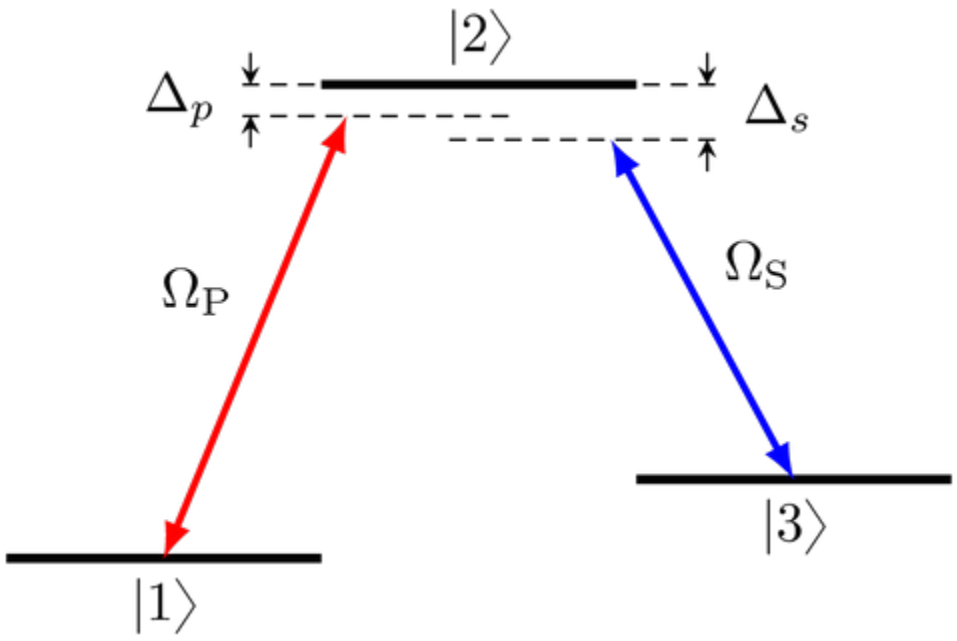
This example revisits the *Optimization of a State-to-State Transfer in a Lambda System in the RWA*, attempting to make the control pulses robustness with respect to variations in the pulse amplitude, through "ensemble optimization".

Note: This notebook uses some parallelization features (`qutip.parallel_map`/`multiprocessing.Pool`). Unfortunately, on Windows, `multiprocessing.Pool` does not work correctly for functions defined in a Jupyter notebook (due to the `spawn method <https://docs.python.org/3/library/multiprocessing.html#contexts-and-start-methods>`__ being used on Windows, instead of Unix-`fork`, see also https://stackoverflow.com/questions/45719956). We therefore replace `parallel_map` with `serial_map` when running on Windows.

[2]:
```
import sys
if sys.platform == 'win32':
    from qutip import serial_map as parallel_map
else:
    from qutip import parallel_map
```

9.8.1 Control objectives for population transfer in the Lambda system

As in the original example, we define the Hamiltonian for a Lambda system in the rotating wave approximation, like this:



We set up the control fields and the Hamiltonian exactly as before:

[3]:
```
def Omega_P1(t, args):
    """Guess for the real part of the pump pulse"""
    Ω0 = 5.0
    return Ω0 * krotov.shapes.blackman(t, t_start=2.0, t_stop=5.0)


def Omega_P2(t, args):
    """Guess for the imaginary part of the pump pulse"""
    return 0.0


def Omega_S1(t, args):
    """Guess for the real part of the Stokes pulse"""
    Ω0 = 5.0
    return Ω0 * krotov.shapes.blackman(t, t_start=0.0, t_stop=3.0)


def Omega_S2(t, args):
    """Guess for the imaginary part of the Stokes pulse"""
    return 0.0
```

[4]:
```
tlist = np.linspace(0, 5, 500)
```

[5]:
```
def hamiltonian(E1=0.0, E2=10.0, E3=5.0, omega_P=9.5, omega_S=4.5):
    """Lambda-system Hamiltonian in the RWA"""

    # detunings
    ΔP = E1 + omega_P - E2
    ΔS = E3 + omega_S - E2

    H0 = Qobj([[ΔP, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, ΔS]])
```

(continues on next page)

(continued from previous page)

```
    HP_re = -0.5 * Qobj([[0.0, 1.0, 0.0], [1.0, 0.0, 0.0], [0.0, 0.0, 0.0]])
    HP_im = -0.5 * Qobj([[0.0, 1.0j, 0.0], [-1.0j, 0.0, 0.0], [0.0, 0.0, 0.0]])

    HS_re = -0.5 * Qobj([[0.0, 0.0, 0.0], [0.0, 0.0, 1.0], [0.0, 1.0, 0.0]])
    HS_im = -0.5 * Qobj([[0.0, 0.0, 0.0], [0.0, 0.0, 1.0j], [0.0, -1.0j, 0.0]])

    return [
        H0,
        [HP_re, Omega_P1],
        [HP_im, Omega_P2],
        [HS_re, Omega_S1],
        [HS_im, Omega_S2],
    ]
```

[6]:
```
H = hamiltonian()
```

The control objective is the realization of a phase sensitive $|1\rangle \rightarrow |3\rangle$ transition in the lab frame. Thus, in the rotating frame, we must take into account an additional phase factor.

[7]:
```
ket1 = qutip.Qobj(np.array([1.0, 0.0, 0.0]))
ket2 = qutip.Qobj(np.array([0.0, 1.0, 0.0]))
ket3 = qutip.Qobj(np.array([0.0, 0.0, 1.0]))
```

[8]:
```
def rwa_target_state(ket3, E2=10.0, omega_S=4.5, T=5):
    return np.exp(1j * (E2 - omega_S) * T) * ket3
```

[9]:
```
psi_target = rwa_target_state(ket3)
```

[10]:
```
objective = krotov.Objective(initial_state=ket1, target=psi_target, H=H)
objectives = [objective]
objectives
```

[10]:
```
[Objective[|Ψ₀(3)⟩ to |Ψ₁(3)⟩ via [H₀[3,3], [H₁[3,3], u₁(t)], [H₂[3,3], u₂(t)], [H₃[3,
↪3], u₃(t)], [H₄[3,3], u₄(t)]]]]
```

9.8.2 Robustness to amplitude fluctuations

A potential source of error is fluctuations in the pulse amplitude between different runs of the experiment. To account for this, the `hamiltonian` function above include a parameter `mu` that scales the pulse amplitudes by the given factor.

We can analyze the result of the *Optimization of a State-to-State Transfer in a Lambda System in the RWA* with respect to such fluctuations. We load the earlier optimization result from disk, and verify that the optimized controls produce the $|1\rangle \rightarrow |3\rangle$ transition as desired.

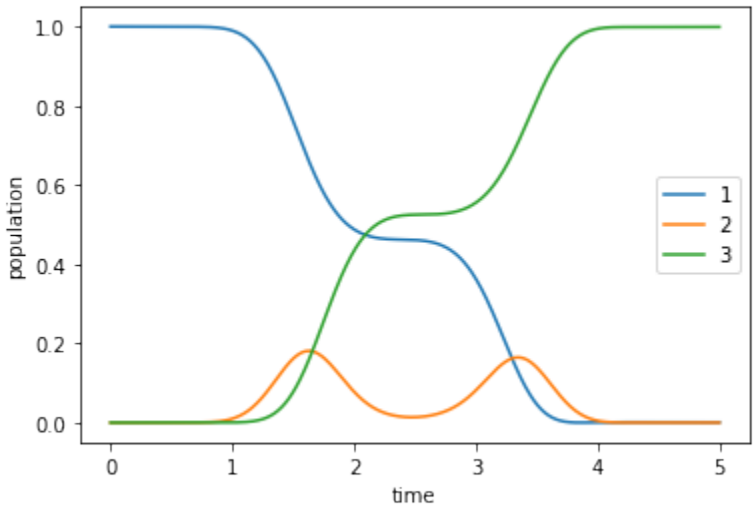
[11]:
```
opt_result_unperturbed = krotov.result.Result.load(
    'lambda_rwa_opt_result.dump', objectives=[objective]
)
```

[12]:
```
proj1 = qutip.ket2dm(ket1)
proj2 = qutip.ket2dm(ket2)
proj3 = qutip.ket2dm(ket3)
```

[13]:
```
opt_unperturbed_dynamics = opt_result_unperturbed.optimized_objectives[
    0
].mesolve(tlist, e_ops=[proj1, proj2, proj3])
```

[14]:
```
def plot_population(result):
    fig, ax = plt.subplots()
    ax.plot(result.times, result.expect[0], label='1')
    ax.plot(result.times, result.expect[1], label='2')
    ax.plot(result.times, result.expect[2], label='3')
    ax.legend()
    ax.set_xlabel('time')
    ax.set_ylabel('population')
    plt.show(fig)
```

[15]:
```
plot_population(opt_unperturbed_dynamics)
```



Now we can analyze how robust this control is for variations of $\pm 20\%$ of the pulse amplitude. Numerically, this is achieved by scaling the control Hamiltonians with a pre-factor μ.

[16]:
```
def scale_control(H, *, mu):
    """Scale all control Hamiltonians by `mu`."""
    H_scaled = []
    for spec in H:
        if isinstance(spec, list):
            H_scaled.append([mu * spec[0], spec[1]])
        else:
            H_scaled.append(spec)
    return H_scaled
```

For the analysis, we take the following sample of μ values:

[17]:
```
mu_vals = np.linspace(0.75, 1.25, 33)
```

We measure the success of the transfer via the "population error", i.e., the deviation from 1.0 of the population in state $|3\rangle$ at final time T.

[18]:
```
def pop_error(obj, mu):
    res = obj.mesolve(tlist, H=scale_control(obj.H, mu=mu), e_ops=[proj3])
```

(continues on next page)

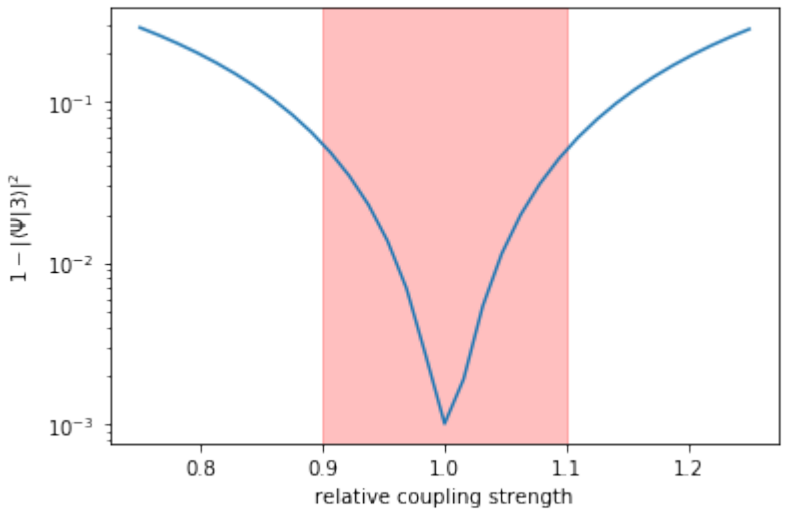
(continued from previous page)

```
    return 1 - res.expect[0][-1]
```

[19]:
```
def _f(mu):
    # parallel_map needs a global function
    return pop_error(opt_result_unperturbed.optimized_objectives[0], mu=mu)


pop_errors_norobust = parallel_map(_f, mu_vals)
```

[20]:
```
def plot_robustness(mu_vals, pop_errors, pop_errors0=None):
    fig, ax = plt.subplots()
    ax.plot(mu_vals, pop_errors, label='1')
    if pop_errors0 is not None:
        ax.set_prop_cycle(None)  # reset colors
        if isinstance(pop_errors0, list):
            for (i, pop_errors_prev) in enumerate(pop_errors0):
                ax.plot(
                    mu_vals, pop_errors_prev, ls='dotted', label=("%d" % (-i))
                )
        else:
            ax.plot(mu_vals, pop_errors0, ls='dotted', label='0')
    ax.set_xlabel("relative coupling strength")
    ax.set_ylabel(r"$1 - \vert \langle \Psi \vert 3 \rangle \vert^2$")
    ax.axvspan(0.9, 1.1, alpha=0.25, color='red')
    ax.set_yscale('log')
    if pop_errors0 is not None:
        ax.legend()
    plt.show(fig)
```

[21]:
```
plot_robustness(mu_vals, pop_errors_norobust)
```



The plot shows that as the pulse amplitude deviates from the optimal value, the error rises quickly: our previous optimization result is not robust.

The highlighted region of ±10% is our “region of interest” within which we would like the control to be robust by applying optimal control.

9.8.3 Setting the ensemble objectives

They central idea of optimizing for robustness is to take multiple copies of the Hamiltonian, sampling over the space of variations to which would like to be robust, and optimize over the average of this ensemble.

Here, we sample 5 values of μ (including the unperturbed $\mu = 1$) in the region of interest, $\mu \in [0.9, 1.1]$.

[22]:
```
ensemble_mu = [1.0, 0.9, 0.95, 1.05, 1.1]
```

The corresponding perturbed Hamiltonians (excluding $\mu = 1$) are

[23]:
```
ham_ensemble = [scale_control(objective.H, mu=mu) for mu in ensemble_mu[1:]]
```

The `krotov.objectives.ensemble_objectives` extends the original objective of a single unperturbed state-to-state transition with one additional objective for each ensemble Hamiltonian:

[24]:
```
ensemble_objectives = krotov.objectives.ensemble_objectives(
    objectives, ham_ensemble
)
ensemble_objectives
```

[24]:
```
[Objective[|Ψ₀(3)⟩ to |Ψ₁(3)⟩ via [H₀[3,3], [H₁[3,3], u₁(t)], [H₂[3,3], u₂(t)], [H₃[3,
↪3], u₃(t)], [H₄[3,3], u₄(t)]]],
 Objective[|Ψ₀(3)⟩ to |Ψ₁(3)⟩ via [H₀[3,3], [H₅[3,3], u₁(t)], [H₆[3,3], u₂(t)], [H₇[3,
↪3], u₃(t)], [H₈[3,3], u₄(t)]]],
 Objective[|Ψ₀(3)⟩ to |Ψ₁(3)⟩ via [H₀[3,3], [H₉[3,3], u₁(t)], [H₁₀[3,3], u₂(t)], [H₁₁[3,
↪3], u₃(t)], [H₁₂[3,3], u₄(t)]]],
 Objective[|Ψ₀(3)⟩ to |Ψ₁(3)⟩ via [H₀[3,3], [H₁₃[3,3], u₁(t)], [H₁₄[3,3], u₂(t)], [H₁₅[3,
↪3], u₃(t)], [H₁₆[3,3], u₄(t)]]],
 Objective[|Ψ₀(3)⟩ to |Ψ₁(3)⟩ via [H₀[3,3], [H₁₇[3,3], u₁(t)], [H₁₈[3,3], u₂(t)], [H₁₉[3,
↪3], u₃(t)], [H₂₀[3,3], u₄(t)]]]]
```

It is important that all five objectives reference the same four control pulses, as is the case here.

9.8.4 Optimize

We use the same update shape $S(t)$ and λ_a value as in the original optimization:

[25]:
```
def S(t):
    """Scales the Krotov methods update of the pulse value at the time t"""
    return krotov.shapes.flattop(t, 0.0, 5, 0.3, func='sinsq')


λ = 0.5

pulse_options = {
    H[1][1]: dict(lambda_a=λ, update_shape=S),
    H[2][1]: dict(lambda_a=λ, update_shape=S),
    H[3][1]: dict(lambda_a=λ, update_shape=S),
    H[4][1]: dict(lambda_a=λ, update_shape=S),
}
```

It will be interesting to see how the optimization progresses for each individual element of the ensemble. Thus, we write an `info_hook` routine that prints out a tabular overview of $1 - \mathrm{Re}\langle\Psi(T) \mid 3\rangle_{\hat{H}_i}$ for all $\hat{H}_i$ in the ensemble, as well as their average (the total functional J_T that is being minimized)

[26]:
```
def print_J_T_per_target(**kwargs):
    iteration = kwargs['iteration']
    N = len(ensemble_mu)
    if iteration == 0:
        print(
            "iteration "
            + "%11s " % "J_T(avg)"
            + " ".join([("J_T(μ=%.2f)" % μ) for μ in ensemble_mu])
        )
    J_T_vals = 1 - kwargs['tau_vals'].real
    J_T = np.sum(J_T_vals) / N
    print(
        ("%9d " % iteration)
        + ("%11.2e " % J_T)
        + " ".join([("%11.2e" % v) for v in J_T_vals])
    )
```

We'll also want to look at the output of `krotov.info_hooks.print_table`, but in order to keep the output orderly, we will write that information to a file `ensemble_opt.log`.

[27]:
```
log_fh = open("ensemble_opt.log", "w", encoding="utf-8")
```

To speed up the optimization slightly, we parallelize across the five objectives with appropriate `parallel_map` functions. The optimization starts for the same guess pulses as the original *Optimization of a State-to-State Transfer in a Lambda System in the RWA*. Generally, for a robustness ensemble optimization, this will yield better results than trying to take the optimized pulses for the unperturbed system as a guess.

[28]:
```
opt_result = krotov.optimize_pulses(
    ensemble_objectives,
    pulse_options,
    tlist,
    propagator=krotov.propagators.expm,
    chi_constructor=krotov.functionals.chis_re,
    info_hook=krotov.info_hooks.chain(
        print_J_T_per_target,
        krotov.info_hooks.print_table(
            J_T=krotov.functionals.J_T_re, out=log_fh
        ),
    ),
    check_convergence=krotov.convergence.Or(
        krotov.convergence.value_below(1e-3, name='J_T'),
        krotov.convergence.check_monotonic_error,
    ),
    parallel_map=(
        qutip.parallel_map,
        qutip.parallel_map,
        krotov.parallelization.parallel_map_fw_prop_step,
    ),
    iter_stop=12,
)
```

```
iteration    J_T(avg) J_T(μ=1.00) J_T(μ=0.90) J_T(μ=0.95) J_T(μ=1.05) J_T(μ=1.10)
        0    1.01e+00    1.01e+00    1.01e+00    1.01e+00    1.01e+00    1.01e+00
        1    6.79e-01    6.75e-01    6.94e-01    6.83e-01    6.71e-01    6.71e-01
        2    4.14e-01    4.07e-01    4.41e-01    4.21e-01    4.00e-01    4.00e-01
        3    2.36e-01    2.27e-01    2.68e-01    2.43e-01    2.20e-01    2.23e-01
        4    1.32e-01    1.21e-01    1.63e-01    1.37e-01    1.16e-01    1.22e-01
        5    7.46e-02    6.29e-02    1.04e-01    7.78e-02    5.98e-02    6.86e-02
        6    4.47e-02    3.24e-02    7.13e-02    4.58e-02    3.13e-02    4.26e-02
        7    2.92e-02    1.66e-02    5.32e-02    2.88e-02    1.72e-02    3.04e-02
        8    2.14e-02    8.59e-03    4.32e-02    1.96e-02    1.04e-02    2.50e-02
        9    1.73e-02    4.48e-03    3.74e-02    1.46e-02    7.25e-03    2.28e-02
       10    1.52e-02    2.38e-03    3.41e-02    1.19e-02    5.83e-03    2.21e-02
       11    1.42e-02    1.29e-03    3.20e-02    1.03e-02    5.23e-03    2.20e-02
       12    1.36e-02    7.20e-04    3.07e-02    9.37e-03    5.00e-03    2.20e-02
```

After twelve iterations (which were sufficient to produce an error $< 10^{-3}$ in the original optimization), we find the average error over the ensemble to be still above $> 10^{-2}$. However, the error for $\mu = 1$ is only *slightly* larger than in the original optimization; the lack of success is entirely due to the large error for the other elements of the ensemble for $\mu \neq 1$. Achieving robustness is hard!

We continue the optimization until the *average* error falls below 10^{-3}:

[29]:
```
dumpfile = "./ensemble_opt_result.dump"
if os.path.isfile(dumpfile):
    opt_result = krotov.result.Result.load(dumpfile, objectives)
    print_J_T_per_target(iteration=0, tau_vals=opt_result.tau_vals[12])
    print("      ...")
    n_iters = len(opt_result.tau_vals)
    for i in range(n_iters - 10, n_iters):
        print_J_T_per_target(iteration=i, tau_vals=opt_result.tau_vals[i])
else:
    opt_result = krotov.optimize_pulses(
        ensemble_objectives,
        pulse_options,
        tlist,
        propagator=krotov.propagators.expm,
        chi_constructor=krotov.functionals.chis_re,
        info_hook=krotov.info_hooks.chain(
            print_J_T_per_target,
            krotov.info_hooks.print_table(
                J_T=krotov.functionals.J_T_re, out=log_fh
            ),
        ),
        check_convergence=krotov.convergence.Or(
            krotov.convergence.value_below(1e-3, name='J_T'),
            krotov.convergence.check_monotonic_error,
        ),
        parallel_map=(
            qutip.parallel_map,
            qutip.parallel_map,
            krotov.parallelization.parallel_map_fw_prop_step,
        ),
        iter_stop=1000,
        continue_from=opt_result,
    )
    opt_result.dump(dumpfile)
```

```
iteration    J_T(avg) J_T(μ=1.00) J_T(μ=0.90) J_T(μ=0.95) J_T(μ=1.05) J_T(μ=1.10)
        0    1.36e-02    7.20e-04    3.07e-02    9.37e-03    5.00e-03    2.20e-02
      …
      670    1.05e-03    1.10e-04    2.80e-03    4.83e-04    5.22e-04    1.34e-03
      671    1.05e-03    1.10e-04    2.79e-03    4.79e-04    5.20e-04    1.33e-03
      672    1.04e-03    1.10e-04    2.77e-03    4.76e-04    5.17e-04    1.33e-03
      673    1.03e-03    1.11e-04    2.76e-03    4.73e-04    5.14e-04    1.32e-03
      674    1.03e-03    1.11e-04    2.74e-03    4.70e-04    5.11e-04    1.31e-03
      675    1.02e-03    1.11e-04    2.72e-03    4.66e-04    5.08e-04    1.30e-03
      676    1.02e-03    1.12e-04    2.71e-03    4.63e-04    5.06e-04    1.29e-03
      677    1.01e-03    1.12e-04    2.69e-03    4.60e-04    5.03e-04    1.28e-03
      678    1.00e-03    1.12e-04    2.68e-03    4.57e-04    5.00e-04    1.27e-03
      679    9.99e-04    1.13e-04    2.67e-03    4.54e-04    4.98e-04    1.27e-03
```

[30]:
```
opt_result
```

[30]:
```
Krotov Optimization Result
--------------------------
- Started at 2019-04-11 20:07:54
- Number of objectives: 1
- Number of iterations: 679
- Reason for termination: Reached convergence: J_T < 0.001
- Ended at 2019-04-11 21:45:38
```

[31]:
```
log_fh.close()
```

Even now, the ideal Hamiltonian ($\mu = 1$) has the lowest error in the ensemble by a significant margin. However, notice that the error in the J_T for $\mu = 1$ is actually rising, while the errors for values of $\mu \neq 1$ are falling by a much larger value! This is a good thing: we sacrifice a little bit of fidelity in the unperturbed dynamics to an increase in robustness.

The optimized "robust" pulse looks as follows:

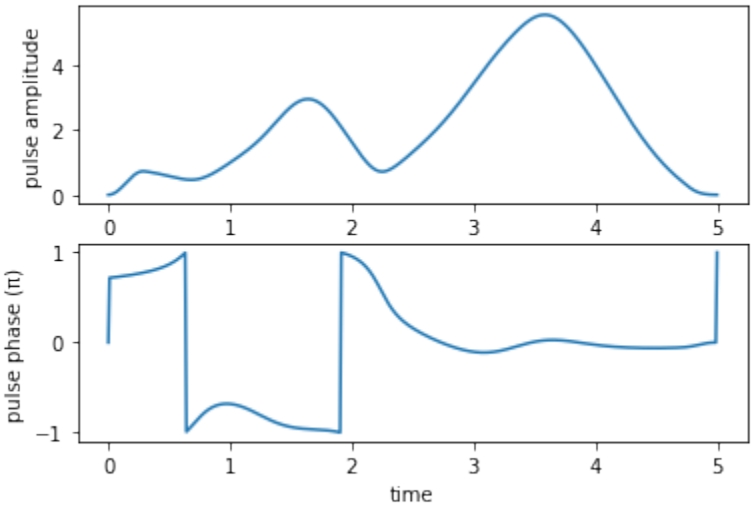
[32]:
```
def plot_pulse_amplitude_and_phase(pulse_real, pulse_imaginary, tlist):
    ax1 = plt.subplot(211)
    ax2 = plt.subplot(212)
    amplitudes = [
        np.sqrt(x * x + y * y) for x, y in zip(pulse_real, pulse_imaginary)
    ]
    phases = [
        np.arctan2(y, x) / np.pi for x, y in zip(pulse_real, pulse_imaginary)
    ]
    ax1.plot(tlist, amplitudes)
    ax1.set_xlabel('time')
    ax1.set_ylabel('pulse amplitude')
    ax2.plot(tlist, phases)
    ax2.set_xlabel('time')
    ax2.set_ylabel('pulse phase (π)')
    plt.show()


print("pump pulse amplitude and phase:")
plot_pulse_amplitude_and_phase(
    opt_result.optimized_controls[0], opt_result.optimized_controls[1], tlist
)
print("Stokes pulse amplitude and phase:")
plot_pulse_amplitude_and_phase(
```

(continues on next page)

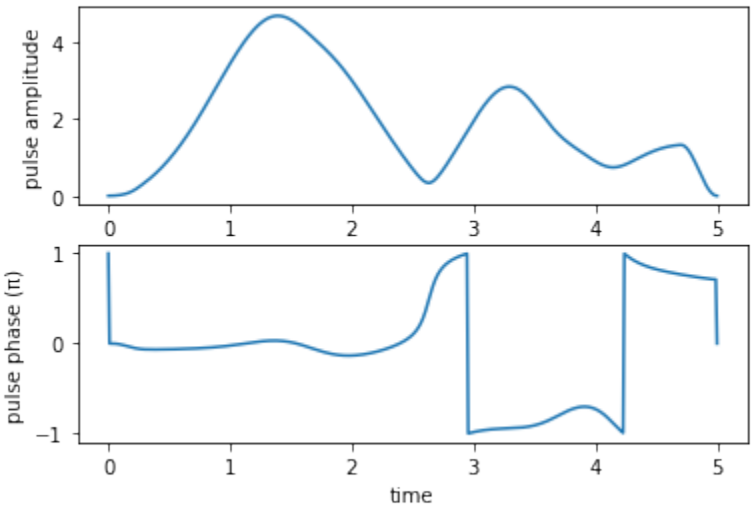
(continued from previous page)

```
    opt_result.optimized_controls[2], opt_result.optimized_controls[3], tlist
)
```

pump pulse amplitude and phase:



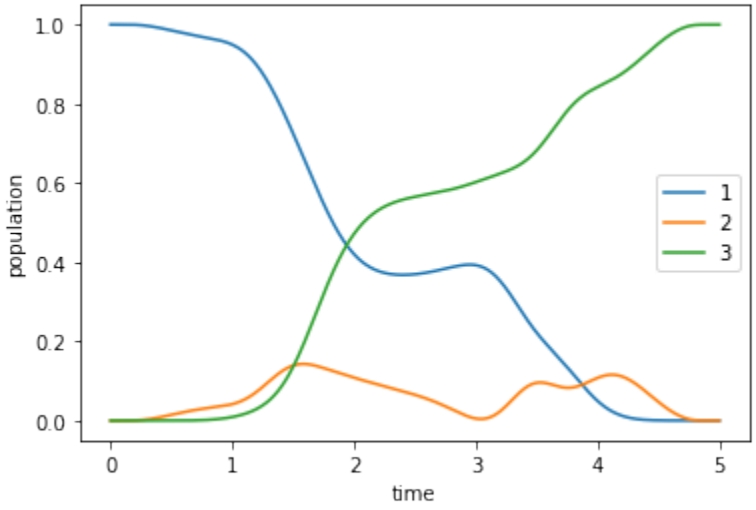
Stokes pulse amplitude and phase:



and produces the dynamics (in the unperturbed system) shown below:

[33]:
```
opt_robust_dynamics = opt_result.optimized_objectives[0].mesolve(
    tlist, e_ops=[proj1, proj2, proj3]
)
```

[34]:
```
plot_population(opt_robust_dynamics)
```



Robustness analysis

When comparing the robustness of the "robust" optimized pulse to that obtained from the original optimization for the unperturbed Hamiltonian, we should make sure that we have converged to a comparable error: We would like to avoid the suspicion that the ensemble error is below our threshold only because the error for $\mu = 1$ is so much lower. Therefore, we continue the original unperturbed optimization for a few more iterations, until we reach the same error $\approx 1.13 \times 10^{-4}$ that we found as the result of the ensemble optimization, looking at $\mu = 1$ only:

[35]:
```
print("J_T(μ=1) = %.2e" % (1 - opt_result.tau_vals[-1][0].real))
```

```
J_T(μ=1) = 1.13e-04
```

[36]:
```
opt_result_unperturbed_cont = krotov.optimize_pulses(
    [objective],
    pulse_options,
    tlist,
    propagator=krotov.propagators.expm,
    chi_constructor=krotov.functionals.chis_re,
    info_hook=krotov.info_hooks.print_table(
        J_T=krotov.functionals.J_T_re,
        show_g_a_int_per_pulse=True,
        unicode=False,
    ),
    check_convergence=krotov.convergence.Or(
        krotov.convergence.value_below(1.13e-4, name='J_T'),
        krotov.convergence.check_monotonic_error,
    ),
    iter_stop=50,
    continue_from=opt_result_unperturbed,
)
```

```
  iter.        J_T   g_a_int_1   g_a_int_2   g_a_int_3   g_a_int_4     g_a_int
↪  J  Delta_J_T    Delta J  secs
     0   5.91e-04    0.00e+00    0.00e+00    0.00e+00    0.00e+00    0.00e+00   5.
↪91e-04        n/a        n/a     1
```

(continues on next page)

(continued from previous page)

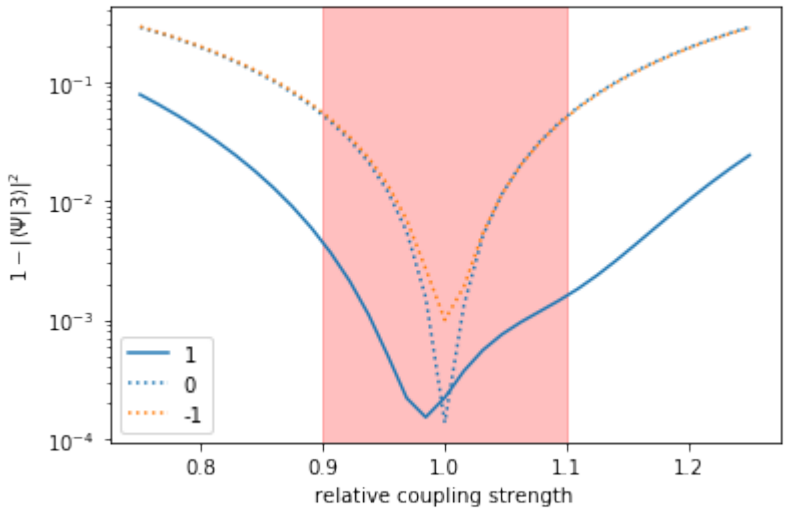
```
     13   3.25e-04    1.26e-04    1.98e-05    1.02e-04    1.84e-05    2.66e-04    5.
↪90e-04   -2.66e-04  -3.54e-07     2
     14   1.83e-04    6.32e-05    1.41e-05    5.12e-05    1.29e-05    1.41e-04    3.
↪24e-04   -1.42e-04  -2.11e-07     2
     15   1.06e-04    3.19e-05    1.00e-05    2.59e-05    9.11e-06    7.69e-05    1.
↪83e-04   -7.70e-05  -1.27e-07     2
```

Now, we can compare the robustness of the optimized pulses from the original unperturbed optimization (label "-1"), the continued unperturbed optimization (label "0"), and the ensemble optimization (label "1"):

[37]:
```
def _f(mu):
    return pop_error(
        opt_result_unperturbed_cont.optimized_objectives[0], mu=mu
    )


pop_errors_norobust_cont = parallel_map(_f, mu_vals)
```

[38]:
```
def _f(mu):
    return pop_error(opt_result.optimized_objectives[0], mu=mu)


pop_errors_robust = parallel_map(_f, mu_vals)
```

[39]:
```
plot_robustness(
    mu_vals,
    pop_errors_robust,
    pop_errors0=[pop_errors_norobust_cont, pop_errors_norobust],
)
```



We see that without the ensemble optimization, we only lower the error for exactly $\mu = 1$: the more we converge, the less robust the result. In contrast, the ensemble optimization results in considerably lower errors (order of magnitude!) throughout the highlighted "region of interest" and beyond.

9.9 Optimization with numpy Arrays

[1]:
```
# NBVAL_IGNORE_OUTPUT
%load_ext watermark
import numpy as np
import scipy
import matplotlib
import matplotlib.pylab as plt
import krotov
# note that qutip is NOT imported
%watermark -v --iversions
```

```
matplotlib       3.1.1
scipy            1.3.1
numpy            1.17.2
krotov           0.4.1
matplotlib.pylab 1.17.2
CPython 3.7.3
IPython 7.8.0
```

The `krotov` package heavily builds on QuTiP. However, in rare circumstances the overhead of `qutip.Qobj` objects might limit numerical efficiency, in particular when QuTiP's automatic sparse storage is inappropriate. If you know what you are doing, it is possible to replace `Qobj`s with low-level objects such as numpy arrays. This example revisits the *Optimization of a State-to-State Transfer in a Two-Level-System*, but exclusively uses numpy objects for states and operators.

9.9.1 Two-level-Hamiltonian

We consider again the standard Hamiltonian of a two-level system, but now we construct the drift Hamiltonian `H0` and the control Hamiltonian `H1` as numpy matrices:

[2]:
```
def hamiltonian(omega=1.0, ampl0=0.2):
    """Two-level-system Hamiltonian

    Args:
        omega (float): energy separation of the qubit levels
        ampl0 (float): constant amplitude of the driving field
    """
    # .full() converts everything to numpy arrays
    H0 = -0.5 * omega * np.array([[-1, 0], [0, 1]], dtype=np.complex128)
    H1 = np.array([[0, 1], [1, 0]], dtype=np.complex128)

    def guess_control(t, args):
        return ampl0 * krotov.shapes.flattop(
            t, t_start=0, t_stop=5, t_rise=0.3, func="sinsq"
        )

    return [H0, [H1, guess_control]]
```

[3]:
```
H = hamiltonian()
```

9.9.2 Optimization target

By default, the `Objective` initializer checks that the objective is expressed with QuTiP objects. If we want to use low-level objects instead, we have to explicitly disable this:

[4]:
```
krotov.Objective.type_checking = False
```

Now, we initialize the initial and target states,

[5]:
```
ket0 = np.array([[1], [0]], dtype=np.complex128)
ket1 = np.array([[0], [1]], dtype=np.complex128)
```

and instantiate the `Objective` for the state-to-state transfer:

[6]:
```
objectives = [krotov.Objective(initial_state=ket0, target=ket1, H=H)]

objectives
```

[6]:
```
[Objective[a₀[2,1] to a₁[2,1] via [a₂[2,2], [a₃[2,2], u₁(t)]]]]
```

Note how all objects are numpy arrays, as indicated by the symbol `a`.

9.9.3 Simulate dynamics under the guess field

To simulate the dynamics under the guess pulse, we can use the objective's `propagator` method. However, the propagator we use must take into account the format of the states and operators. We define a simple propagator that solve the dynamics within a single time step my matrix exponentiation of the Hamiltonian:

[7]:
```
def expm(H, state, dt, c_ops=None, backwards=False, initialize=False):
    eqm_factor = -1j  # factor in front of H on rhs of the equation of motion
    if backwards:
        eqm_factor = eqm_factor.conjugate()
    A = eqm_factor * H[0]
    for part in H[1:]:
        A += (eqm_factor * part[1]) * part[0]
    return scipy.linalg.expm(A * dt) @ state
```

We will want to analyze the population dynamics, and thus define the projectors on the ground and excited levels, again as numpy matrices:

[8]:
```
proj0 = np.array([[1, 0],[0, 0]], dtype=np.complex128)
proj1 = np.array([[0, 0],[0, 1]], dtype=np.complex128)
```

We will pass these as `e_ops` to the `propagate` method, but since `propagate` assumes that `e_ops` contains `Qobj` instances, we will have to teach it how to calculate expectation values:

[9]:
```
def expect(proj, state):
    return complex(state.conj().T @ (proj @ state)).real
```

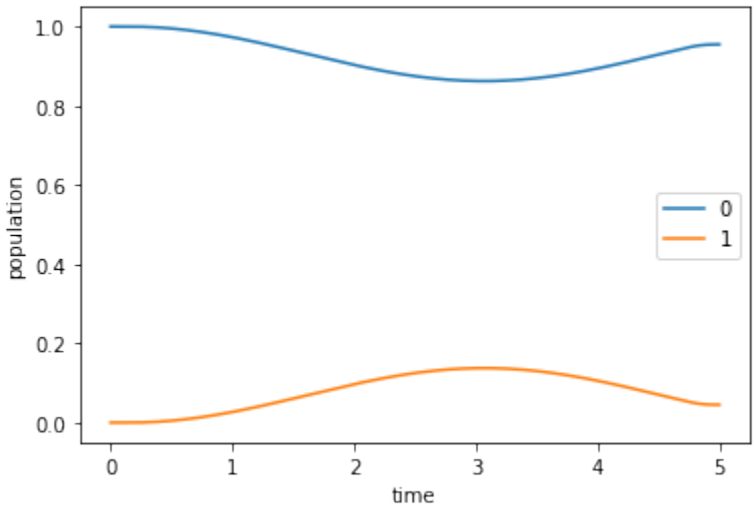
Now we can simulate the dynamics over a time grid from $t = 0$ to $T = 5$ and plot the resulting dynamics.

[10]:
```
tlist = np.linspace(0, 5, 500)
```

[11]:
```
guess_dynamics = objectives[0].propagate(tlist, propagator=expm, e_ops=[proj0, proj1],
↪ expect=expect)
```

[12]:
```
def plot_population(result):
    fig, ax = plt.subplots()
    ax.plot(result.times, result.expect[0], label='0')
    ax.plot(result.times, result.expect[1], label='1')
    ax.legend()
    ax.set_xlabel('time')
    ax.set_ylabel('population')
    plt.show(fig)
```

[13]:
```
plot_population(guess_dynamics)
```



The result is the same as in the original example.

9.9.4 Optimize

First, we define the update shape and step width as before:

[14]:
```
def S(t):
    """Shape function for the field update"""
    return krotov.shapes.flattop(
        t, t_start=0, t_stop=5, t_rise=0.3, t_fall=0.3, func='sinsq'
    )
```

[15]:
```
pulse_options = {
    H[1][1]: dict(lambda_a=5, update_shape=S)
}
```

Now we can run the optimization with only small additional adjustments. This is because Krotov's method internally does very little with the states and operators: nearly all of the numerical effort is in the propagator, which we have already defined above for the specific use of numpy arrays.

Beyond this, the optimization only needs to know three things: First, it must know how to calculate and apply the operator $\partial H/\partial \epsilon$. We can easily teach it how to do this:

[16]:
```
def mu(objectives, i_objective, pulses, pulses_mapping, i_pulse, time_index):
    def _mu(state):
        return H[1][0] @ state
    return _mu
```

Second, the pulse updates are calculated from an overlap of states, and we define an appropriate function for numpy arrays:

[17]:
```
def overlap(psi1, psi2):
    return complex(psi1.conj().T @ psi2)
```

Third, it must know how to calculate the norm of states, for which we can use `np.linalg.norm`.

By passing all these routines to `optimize_pulses`, we get the exact same results as in the original example, except much faster:

[18]:
```
opt_result = krotov.optimize_pulses(
    objectives,
    pulse_options=pulse_options,
    tlist=tlist,
    propagator=expm,
    chi_constructor=krotov.functionals.chis_re,
    info_hook=krotov.info_hooks.print_table(J_T=krotov.functionals.J_T_re),
    check_convergence=krotov.convergence.check_monotonic_error,
    iter_stop=10,
    norm=np.linalg.norm,
    mu=mu,
    overlap=overlap,
)
```

```
  iter.        J_T    ∫gₐ(t)dt          J       ΔJ_T         ΔJ  secs
     0   1.00e+00    0.00e+00   1.00e+00        n/a        n/a     0
     1   7.65e-01    2.33e-02   7.88e-01  -2.35e-01  -2.12e-01     0
     2   5.56e-01    2.07e-02   5.77e-01  -2.09e-01  -1.88e-01     0
     3   3.89e-01    1.66e-02   4.05e-01  -1.67e-01  -1.51e-01     0
     4   2.65e-01    1.23e-02   2.77e-01  -1.24e-01  -1.12e-01     0
     5   1.78e-01    8.63e-03   1.87e-01  -8.68e-02  -7.82e-02     0
     6   1.19e-01    5.86e-03   1.25e-01  -5.89e-02  -5.30e-02     0
     7   8.01e-02    3.91e-03   8.40e-02  -3.92e-02  -3.53e-02     0
     8   5.42e-02    2.58e-03   5.68e-02  -2.59e-02  -2.33e-02     0
     9   3.71e-02    1.70e-03   3.88e-02  -1.71e-02  -1.54e-02     0
    10   2.58e-02    1.13e-03   2.70e-02  -1.13e-02  -1.02e-02     0
```

[19]:
```
opt_result
```

[19]:
```
Krotov Optimization Result
--------------------------
- Started at 2019-10-11 12:29:52
- Number of objectives: 1
- Number of iterations: 10
- Reason for termination: Reached 10 iterations
- Ended at 2019-10-11 12:29:57
```

10 How-Tos

10.1 How to optimize towards a quantum gate

To optimize towards a quantum gate $\hat{O}$ in a *closed* quantum system, set one `Objective` for each state in the logical basis, with the basis state $|\phi_k\rangle$ as the `initial_state` and $\hat{O}|\phi_k\rangle$ as the `target`.

You may use `krotov.gate_objectives()` to construct the appropriate list of objectives. See the *Optimization of an X-Gate for a Transmon Qubit* for an example. For more advanced gate optimizations, also see *How to optimize towards a two-qubit gate up to single-qubit corrections*, *How to optimize towards an arbitrary perfect entangler*, *How to optimize in a dissipative system*, and *How to optimize for robust pulses*.

10.2 How to optimize complex-valued control fields

This implementation of Krotov's method requires real-valued control fields. You must rewrite your Hamiltonian to contain the real part and the imaginary part of the field as two independent controls. This is always possible. For example, for a driven harmonic oscillator in the rotating wave approximation, the interaction Hamiltonian is given by

$$\hat{H}_{\text{int}} = \epsilon^*(t)\hat{a} + \epsilon(t)\hat{a}^\dagger = \epsilon_{\text{re}}(t)(\hat{a} + \hat{a}^\dagger) + \epsilon_{\text{im}}(t)(i\hat{a}^\dagger - i\hat{a})\,,$$

where $\epsilon_{\text{re}}(t) = \text{Re}[\epsilon(t)]$ and $\epsilon_{\text{im}}(t) = \text{Im}[\epsilon(t)]$ are considered as two independent (real-valued) controls.

See the *Optimization of a State-to-State Transfer in a Lambda System in the RWA* for an example.

10.3 How to stop the optimization when the error crosses some threshold

By default, an optimization stops after a predefined number of iterations (*iter_stop* parameter in `optimize_pulses()`). However, through the interplay of the *info_hook* and the *check_convergence* routine passed to `optimize_pulses()`, the optimization can be stopped based on

the optimization success or the rate of convergence: The *info_hook* routine should return the value of the optimization functional or error, which is accessible to *check_convergence* via the `Result.info_vals` attribute, see `krotov.convergence` for details.

Generally, you should use the `krotov.info_hooks.print_table()` function as an *info_hook*, which receives a function to evaluate the optimization functional J_T as a parameter. Then, use `krotov.convergence.value_below()` as a *check_convergence* routine to stop the optimization when J_T falls below some given threshold.

See the *Optimization of a State-to-State Transfer in a Lambda System in the RWA* for an example.

10.4 How to exclude a control from the optimization

In order to force the optimization to leave any particular control field unchanged, set its update shape to `krotov.shapes.zero_shape()` in the *pulse_options* that you pass to `optimize_pulses()`.

10.5 How to define a new optimization functional

In order to define a new optimization functional J_T:

- Decide on what should go in `Objective.target` to best describe the *physical* control target. If the control target is reached when the `Objective.initial_state` evolves to a specific target state under the optimal control fields, that target state should be included in `target`.
- Define a function *chi_constructor* that calculates the boundary condition for the backward-propagation in Krotov's method,

  $$|\chi_k(T)\rangle \equiv -\left.\frac{\partial J_T}{\partial \langle\phi_k(T)|}\right|_{|\phi_k(T)\rangle},$$

  or the equivalent experession in Liouville space. This function should calculate the states $|\chi_k\rangle$ based on the forward-propagated states $|\phi_k(T)\rangle$ and the list of objectives. For convenience, when `target` contains a target state, *chi_constructor* will also receive *tau_vals* containing the overlaps $\tau_k = \left\langle \phi_k^{\text{tgt}} \mid \phi_k(T) \right\rangle$. See `chis_re()` for an example.
- Optionally, define a function that can be used as an *info_hook* in `optimize_pulses()` which returns the value J_T. This is not required to run an optimization since the functional is entirely implicit in *chi_constructor*. However, calculating the value of the functional is useful for convergence analysis (*check_convergence* in `optimize_pulses()`)

See `krotov.functionals` for some standard functionals. An example for a more advanced functional is the *Optimization towards a Perfect Entangler*.

10.6 How to penalize population in a forbidden subspace

In principle, `optimize_pulses()` has a *state_dependent_constraint*. However, this has some caveats. Most notably, it results in an inhomogeneous equation of motion, which is currently not implemented.

The recommended "workaround" is to place artificially high dissipation on the levels in the forbidden subspace. A non-Hermitian Hamiltonian is usually a good way to realize this. See the *Optimization of a Dissipative State-to-State Transfer in a Lambda System* for an example.

10.7 How to optimize towards a two-qubit gate up to single-qubit corrections

On many quantum computing platforms, applying arbitrary single-qubit gates is easy compared to entangling two-qubit gates. A specific entangling gate like CNOT is combined with single-qubit gates to form a universal set of gates. For a given physical system, it can be hard to know a-priori which entangling gates are easy or even possible to realize. For example, trapped neutral atoms only allow for the realization of diagonal two-qubit gates [53][26] like CPHASE. However, the CPHASE gate is "locally equivalent" to CNOT: only additional single-qubit operations are required to obtain one from the other. A "local-invariants functional" [54] defines an optimization with respect to a such a local equivalence class, and thus is free to find the specific realization of a two-qubit gate that is easiest to realize.

Use *krotov.objectives.gate_objectives()* with `local_invariants=True` in order to construct a list of objectives suitable for an optimization using the local-invariant functional [54]. This optimizes towards a point in the Weyl chamber.

The `weylchamber` package contains the suitable *chi_constructor* routines to pass to *optimize_pulses()*.

The optimization towards a local equivalence class may require use of the second-order update equation, see *Second order update*.

10.8 How to optimize towards an arbitrary perfect entangler

The relevant property of a gate is often its entangling power, and the requirement for a two-qubit gate in a universal set of gates is that it is a "perfect entangler". A perfect entangler can produce a maximally entangled state from a separable input state. Since 85% of all two-qubit gates are perfect entanglers [55][56], a functional that targets an arbitrary perfect entangler [27][28] solves the control problem with the least constraints.

The optimization towards an arbitrary perfect entangler is closely related to an optimization towards a point in the Weyl chamber (*How to optimize towards a two-qubit gate up to single-qubit corrections*): It turns out that in the geometric representation of the Weyl chamber, all the perfect entanglers lie within a polyhedron, and we can simply minimize the geometric distance to the surface of this polyhedron.

Use *krotov.objectives.gate_objectives()* with `gate='PE'` in order to construct a list of objectives suitable for an optimization using the perfect entanglers functional [27][28]. This is illustrated in the *Optimization towards a Perfect Entangler*.

Again, the *chi_constructor* is available in the `weylchamber` package.

Both the optimization towards a local equivalence class and an arbitrary perfect entangler may require use of the second-order update equation, see *Second order update*.

10.9 How to optimize in a dissipative system

To optimize a dissipative system, it is sufficient to set an *Objective* with a density matrix for the *initial_state* and *target*, and a Liouvillian in *Objective.H*. See the *Optimization of Dissipative Qubit Reset* for an example.

Instead of a Liouvillian, it is also possible to set *Objective.H* to the system Hamiltonian, and *Objective.c_ops* to the appropriate Lindblad operators. However, it is generally much more efficient to use *krotov.objectives.liouvillian()* to convert a time-dependent Hamiltonian and a list of Lindblad operators into a time-dependent Liouvillian. In either case, the *propagate* routine passed to *optimize_pulses()* must be aware of and compatible with the convention for the objectives.

Specifically for gate optimization, the routine *gate_objectives()* can be used to automatically set appropriate objectives for an optimization in Liouville space. The parameter *liouville_states_set* indicates that the system dynamics are in Liouville space and sets an appropriate choice of matrices that track the optimization according to Ref. [26]. See the *Optimization of a Dissipative Quantum Gate* for an example.

For weak dissipation, it may also be possible to avoid the use of density matrices altogether, and to instead use a non-Hermitian Hamiltonian. For example, you may use the effective Hamiltonian from the MCWF method [57],

$$\hat{H}_{\text{eff}} = \hat{H} - \frac{i}{2} \sum_k \hat{L}_k^\dagger \hat{L}_k \,,$$

for the Hermitian Hamiltonian $\hat{H}$ and the Lindblad operators $\hat{L}_k$. Propagating $\hat{H}_{\text{eff}}$ (without quantum jumps) will lead to a decay in the norm of the state corresponding to how much dissipation the state is subjected to. Numerically, this will usually increase the value of the optimization functional (that is, the error). Thus the optimization can be pushed towards avoiding decoherence, without explicitly performing the optimization in Liouville space. See the *Optimization of a Dissipative State-to-State Transfer in a Lambda System* for an example.

10.10 How to optimize for robust pulses

Control fields can be made robust with respect to variations in the system by performing an "ensemble optimization" [25]. The idea is to sample a representative selection of possible system Hamiltonians, and to optimize over an average of the entire ensemble. In the functional, Eq. (7.1), respectively the update Eq. (7.11), the index k now numbers not only the states, but also different ensemble Hamiltonians: $\hat{H}(\{\epsilon_l(t)\}) \rightarrow \{\hat{H}_k(\{\epsilon_l(t)\})\}$.

The example considered in Ref. [25] is that of a CPHASE two-qubit gate on trapped Rydberg atoms. Two classical fluctuations contribute significantly to the gate error: deviations in the pulse amplitude ($\Omega = 1$ ideally), and fluctuations in the energy of the Rydberg level ($\Delta_{\text{ryd}} = 0$ ideally). Starting from a set of objectives for the unperturbed system, see *How to optimize towards a quantum gate*, *ensemble_objectives()* creates an extended set of objectives that duplicates the original objectives once for each Hamiltonian from a set perturbed Hamiltonian $\hat{H}(\Omega \neq 1, \Delta_{\text{ryd}} \neq 0)$. As shown in Ref. [26], an optimization over the average of all these objectives results in controls that are robust over a wide range of system perturbations.

A simpler example of an ensemble optimization is *Ensemble Optimization for Robust Pulses*, which considers a state-to-state transition in a Lamba-System with a dissipative intermediary state.

10.11 How to apply spectral constraints

In principle, Krotov's method can include spectral constraints while maintaining the guarantee for monotonic convergence [58] . However, the calculation of the pulse update with such spectral constraints requires solving a Fredholm equation of the second kind, which has not yet been implemented numerically. Thus, the `krotov` package does not support this approach (and no such support is planned).

A "cheap" alternative that usually yields good results is to apply a spectral filter to the optimized pulses after each iteration. The *optimize_pulses()* function allows this via the *modify_params_after_iter* argument.

For example, the following function restricts the spectrum of each pulse to a given range:

```
def apply_spectral_filter(tlist, w0, w1):
    """Spectral filter for real-valued pulses.

    The resulting filter function performs a Fast-Fourier-Transform (FFT) of
    each optimized pulse, and sets spectral components for angular
    frequencies below `w0` or above `w1` to zero. The filtered pulse is then
    the result of the inverse FFT, and multiplying again with the update
    shape for the pulse, to ensure that the filtered pulse still fulfills
    the required boundary conditions.

    Args:
        tlist (numpy.ndarray): Array of time grid values. All pulses must be
            defined on the intervals of this time grid
        w0 (float): The lowest allowed (angular) frequency
        w1 (float): The highest allowed (angular) frequency

    Returns:
        callable: A function that can be passed to
        `modify_params_after_iter` to apply the spectral filter.
    """

    dt = tlist[1] - tlist[0]  # assume equi-distant time grid

    n = len(tlist) - 1  # = len(pulse)
    # remember that pulses are defined on intervals of tlist

    w = np.abs(np.fft.fftfreq(n, d=dt / (2.0 * np.pi)))
    # the normalization factor 2π means that w0 and w1 are angular
    # frequencies, corresponding directly to energies in the Hamiltonian
    # (ħ = 1).

    flt = (w0 <= w) * (w <= w1)
    # flt is the (boolean) filter array, equivalent to an array of values 0
    # and 1

    def _filter(**kwargs):
        # same interface as an `info_hook` function
        pulses = kwargs['optimized_pulses']
        shape_arrays = kwargs['shape_arrays']
        for (pulse, shape) in zip(pulses, shape_arrays):
            spectrum = np.fft.fft(pulse)
            # apply the filter by element-wise multiplication
            spectrum[:] *= flt[:]
```

(continues on next page)

(continued from previous page)

```
            # after the inverse fft, we should also multiply with the
            # update shape function. Otherwise, there is no guarantee that
            # the filtered pulse will be zero at t=0 and t=T (assuming that
            # is what the update shape is supposed to enforce). Also, it is
            # important that we overwrite `pulse` in-place (pulse[:] = ...)
            pulse[:] = np.fft.ifft(spectrum).real * shape

    return _filter
```

This function is passed to *optimize_pulses()* as e.g.

```
modify_params_after_iter=apply_spectral_filter(tlist, 0, 7)
```

to constrain the spectrum of the pulse to angular frequencies $\omega \in [0, 7]$. You may want to explore how such a filter behaves in the example of the *Optimization of an X-Gate for a Transmon Qubit*.

Modifying the optimized pulses "manually" through a `modify_params_after_iter` function means that we lose all guarantees of monotonic convergence. If the optimization with a spectral filter does not converge, you should increase the value of λ_a in the *pulse_options* that are passed to *optimize_pulses()*. A larger value of λ_a results in smaller updates in each iteration. This should also translate into the filter pulses being closer to the unfiltered pulses, increasing the probability that the changes due to the filter do not undo the monotonic convergence. You may also find that the optimization fails if the control problem physically cannot be solved with controls in the desired spectral range. Without a good physical intuition, trial and error may be required.

10.12 How to limit the amplitude of the controls

Amplitude constraints on the control can be realized indirectly through parametrization [59]. For example, consider the physical Hamiltonian $\hat{H} = \hat{H}_0 + \epsilon(t)\hat{H}_1$.

There are several possible parametrizations of $\epsilon(t)$ in terms of an unconstrained function $u(t)$:

- For $\epsilon(t) \geq 0$:

$$\epsilon(t) = u^2(t)$$

- For $0 \leq \epsilon(t) < \epsilon_{\max}$:

$$\epsilon(t) = \epsilon_{\max} \tanh^2(u(t))$$

- For $\epsilon_{\min} < \epsilon(t) < \epsilon_{\max}$:

$$\epsilon(t) = \frac{\epsilon_{\max} - \epsilon_{\min}}{2} \tanh(u(t)) + \frac{\epsilon_{\max} + \epsilon_{\min}}{2}$$

Krotov's method can now calculate the update $\Delta u(t)$ in each iteration, and then $\Delta\epsilon(t)$ via the above equations.

There is a caveat: In the update equation (7.11), we now have the term

$$\left(\frac{\partial \hat{H}}{\partial u}\bigg|_{\substack{\phi^{(i+1)}(t) \\ u^{(i+1)}(t)}} \right) = \left(\frac{\partial \epsilon}{\partial u} \frac{\partial \hat{H}}{\partial \epsilon}\bigg|_{\substack{\phi^{(i+1)}(t) \\ u^{(i+1)}(t)}} \right)$$

on the right hand side. As the dependendence of $\epsilon(t)$ on $u(t)$ is non-linear, we are left with a dependency on the unknown updated parametrization $u^{(i+1)}(t)$. We resolve this by approximating $u^{(i+1)}(t) \approx u^{(i)}(t)$, or equivalently $\Delta u(t) \ll u(t)$, which can be enforced by choosing a sufficiently large value of λ_a in the *pulse_options* that are passed to *optimize_pulses()*.

Currently, the `krotov` package does not yet support parametrizations in the above form, although this is a planned feature. In the meantime, you could modify the control to fit within the desired amplitude constaints in the same way as applying spectral constaints, see *How to apply spectral constraints*.

10.13 How to parallelize the optimization

Krotov's method is inherently parallel across different objectives. See *krotov.parallelization*, and the *Optimization of an X-Gate for a Transmon Qubit* for an example.

10.14 How to prevent losing an optimization result

Optimizations usually take several hundred to several thousand iterations to fully converge. Thus, the *optimize_pulses()* routine may require significant runtime (often multiple days for large problems). Once an optimization has completed, you are strongly encouraged to store the result to disk, using *Result.dump()*. You may also consider using *dump_result()* during the *check_convergence* step to dump the current state of the optimization to disk at regular intervals. This protects you from losing work if the optimization is interrupted in any way, like an unexpected crash.

In order to continue after such a crash, you can restore a *Result* object containing the recent state of the optimization using *Result.load()* (with the original *objectives* and `finalize=True` if the dump file originates from *dump_result()*). You may then call *optimize_pulses()* and pass the loaded *Result* object as *continue_from*. The new optimization will start from the most recent optimized controls as a guess, and continue to count iterations from the previous result. See *How to continue from a previous optimization* for further details.

10.15 How to continue from a previous optimization

See *How to prevent losing an optimization result* for how to continue from an optimization that ended (crashed) prematurely. Even when an optimization has completed normally, you may still want to continue with further iterations – either because you find that the original *iter_stop* was insufficient to reach full convergence, or because you would like to modify some parameters, like the λ_a values for each control. In this case, you can again call *optimize_pulses()* and pass the *Result* object from the previous optimization as *continue_from*. Note that while you are free to change the *pulse_options* between the two optimization, the *objectives* must remain the same. The functional (*chi_constructor*) and the

info_hook should also remain the same (otherwise, you may and up with inconsistencies in your *Result*). The *Result* object returned by the second optimization will include all the data from the first optimization.

10.16 How to maximize numerical efficiency

For systems of non-trivial size, the main numerical effort should be in the simulation of the system dynamics. Every iteration of Krotov's method requires a full backward propagation and a full forward propagation of the states associated with each objective, see *krotov.propagators*. Therefore, the best numerical efficiency can be achieved by optimizing the performance of the *propagator* that is passed to *optimize_pulses()*.

One possibility is to implement problem-specific propagators, such as *krotov.propagators.DensityMatrixODEPropagator*. Going further, you might consider implementing the propagator with the help of lower-level instructions, e.g., by using Cython.

10.17 How to deal with the optimization running out of memory

Krotov's method requires the storage of at least one set of propagated state over the entire time grid, for each objective. For the second-order update equation, up to three sets of stored states per objective may be required. In particular for larger systems and dynamics in Liouville space, the memory required for storing these states may be prohibitively expensive.

The *optimize_pulses()* accepts a *storage* parameter to which a constructor for an array-like container can be passed wherein the propagated states will be stored. It is possible to pass custom out-of-memory storage objects, such as Dask arrays. This may carry a significant penalty in runtime, however, as states will have to be read from disk, or across the network.

10.18 How to avoid the overhead of QuTiP objects

If you know what you are doing, it is possible to set up an *Objective* without any `qutip.Qobj` instances, using arbitrary low-level objects instead. See the *Optimization with numpy Arrays* for an example.

11 Other Optimization Methods

In the following, we compare Krotov's method to other numerical optimization methods that have been used widely in quantum control, with an emphasis on methods that have been implemented as open source software.

11.1 Iterative schemes from variational calculus

Gradient-based optimal control methods derive the condition for the optimal control field from the application of the variational principle to the optimization functional in Eq. (7.1). Since the functional depends both on the states and the control field, it is necessary to include the equation of motion (Schrödinger or Liouville-von-Neumann) as a constraint. That is, the states $\{|\phi_k\rangle\}$ must be compatible with the equation of motion under the control fields $\{\epsilon_l(t)\}$. In order to convert the constrained optimization problem into an unconstrained one, the equation of motion is included in the functional with the co-states $|\chi_k(t)\rangle$ as Lagrange multipliers [60][61][62][63].

The necessary condition for an extremum becomes $\delta J = 0$ for this extended functional. Evaluation of the extremum condition results in [63]

$$\Delta\epsilon_l(t) \propto \frac{\delta J}{\delta \epsilon_l} \propto \mathrm{Im}\left\langle \chi_k(t) \middle| \hat{\mu} \middle| \phi_k(t) \right\rangle, \tag{11.1}$$

where $\hat{\mu} = \partial \hat{H}/\partial \epsilon_l(t)$ is the operator coupling to the field $\epsilon_l(t)$. Equation (11.1) is both continuous in time and implicit in $\epsilon_l(t)$ since the states $|\phi_k(t)\rangle$, $|\chi_k(t)\rangle$ also depend on $\epsilon_l(t)$. Numerical solution of Eq. (11.1) thus requires an iterative scheme and a choice of time discretization.

The most intuitive time-discretization yields a *concurrent* update scheme [63][5][43],

$$\Delta\epsilon_l^{(i)}(t) \propto \mathrm{Im}\left\langle \chi_k^{(i-1)}(t) \middle| \hat{\mu} \middle| \phi_k^{(i-1)}(t) \right\rangle. \tag{11.2}$$

Here, at iterative step (i), the backward-propagated co-states $\{|\chi_k(t)\rangle\}$ and the forward-propagated states $\{|\phi_k(t)\rangle\}$ both evolve under the 'guess' controls $\epsilon_l^{(i-1)}(t)$ of that iteration. Thus, the update is determined entirely by information from the previous iteration and can be evaluated at each point t independently. However, this scheme does not guarantee monotonic convergence, and requires a line search to determine the appropriate magnitude of the pulse update [63].

A further ad-hoc modification of the functional [64] allows to formulate a family of update schemes that do guarantee monotonic convergence [65][66]. These schemes introduce separate fields $\{\epsilon_l(t)\}$ and $\{\tilde{\epsilon}_l(t)\}$ for the forward and backward propagation, respectively, and use the update scheme [67]

$$\begin{aligned}\epsilon_l^{(i)}(t) &= (1-\delta)\tilde{\epsilon}_l^{(i-1)}(t) - \frac{\delta}{\alpha}\,\mathrm{Im}\left\langle \chi_k^{(i-1)}(t)\middle|\hat{\mu}\middle|\phi_k^{(i)}(t)\right\rangle \\ \tilde{\epsilon}_l^{(i)}(t) &= (1-\eta)\epsilon_l^{(i-1)}(t) - \frac{\eta}{\alpha}\,\mathrm{Im}\left\langle \chi_k^{(i)}(t)\middle|\hat{\mu}\middle|\phi_k^{(i)}(t)\right\rangle,\end{aligned} \tag{11.3}$$

with $\delta, \eta \in [0, 2]$ and an arbitrary step width α. For the control of wavepacket dynamics, an implementation of this generalized class of algorithms is available in the WavePacket Matlab package [68].

11.2 Krotov's method

The method developed by Krotov [39][40][41][42] and later translated to the language of quantum control by Tannor and coworkers [5][43][44][45][22] takes a somewhat unintuitive approach to disentangle the interdependence of field and states by adding a zero to the functional. This allows to *construct* an updated control field that is guaranteed to lower the value of the functional, resulting in monotonic convergence. The full method is described in *Krotov's Method*, but its essence can be boiled down to the update in each iteration (i), Eq. (7.2), taking the form

$$\Delta\epsilon_l^{(i)}(t) \propto \mathrm{Im}\left\langle \chi_k^{(i-1)}(t)\middle|\hat{\mu}\middle|\phi_k^{(i)}(t)\right\rangle, \tag{11.4}$$

with co-states $|\chi_k(t)^{(i-1)}\rangle$ backward-propagated under the *guess* controls $\{\epsilon_l^{(i-1)}(t)\}$ and the states $|\phi_k^{(i)}(t)\rangle$ forward-propagated under the *optimized* controls $\{\epsilon_l^{(i)}(t)\}$. Compared to the *concurrent* form of Eq. (11.2), the Krotov update scheme is *sequential*: The update at time t depends on the states forward-propagated using the updated controls at all previous times, see *Time discretization* for details.

It is worth noting that the sequential update can be recovered as a limiting case of the monotonically convergent class of algorithms in Eq. (11.3), for $\delta = 1$, $\eta = 0$. This may explain why parts of the quantum control community consider *any* sequential update scheme as "Krotov's method" [69][70]. However, following Krotov's construction [39][40][41][42] requires no ad-hoc modification of the functional and can thus be applied more generally. In particular, as discussed in *Second order update*, a second-order construction can address non-convex functionals.

In all its variants [5][43][44][45][22], Krotov's method is a first-order gradient with respect to the control fields (even in the second-order construction which is second order only with respect to the states). As the optimization approaches the optimum, this gradient can become very small, resulting in slow convergence. It is possible to extend Krotov's method to take into account information from the quasi-Hessian [71]. However, this "K-BFGS" variant of Krotov's method is a substantial extension to the procedure as described in *Krotov's Method*, and is currently not supported by the `krotov` package.

The update Eq. (11.4) is specific to the running cost in Eq. (7.3). In most of the *Iterative schemes from variational calculus*, a constraint on the *pulse fluence* is used instead. Formally, this is also compatible with Krotov's method, by choosing $\epsilon_{l,\mathrm{ref}}^{(i)}(t) \equiv 0$ in Eq. (7.3) [72]. It turns the *update* equations (11.4), (11.2) into *replacement* equations, with $\epsilon_l^{(i)}(t)$ on the left-hand side instead of $\Delta\epsilon_l^{(i)}(t)$, cf. Eq. (11.3) for $\delta = 1$, $\eta = 0$. In our experience, this leads to numerical instability and should be avoided. A mixture of *update* and *replacement* is possible when a penalty of the pulse fluence is necessary [73].

11.3 GRadient Ascent Pulse Engineering (GRAPE)

While the monotonically convergent methods based on variational calculus must "guess" the appropriate time discretization, and Krotov's method finds the sequential time discretization by a clever construction, the GRAPE method sidesteps the problem by discretizing the functional *first*, before applying the variational calculus.

Specifically, we consider the piecewise-constant discretization of the dynamics onto a time grid, where the final time states $\{|\phi_k^{(i-1)}(T)\rangle\}$ resulting from the time evolution of the initial states $\{|\phi_k\rangle\}$ under the guess controls $\epsilon_n^{(i-1)}$ in iteration (i) of the optimization are obtained as

$$|\phi_k^{(i-1)}(T)\rangle = \hat{U}_{nt-1}^{(i-1)} \ldots \hat{U}_n^{(i-1)} \ldots \hat{U}_1^{(i-1)} \big|\phi_k\big\rangle\,, \tag{11.5}$$

where $\hat{U}_n^{(i-1)}$ is the time evolution operator on the time interval n in Hilbert space,

$$\hat{U}_n^{(i-1)} = \exp\left[-\frac{\mathrm{i}}{\hbar}\hat{H}\big(\underbrace{\epsilon^{(i-1)}(\tilde{t}_{n-1})}_{=\epsilon_n^{(i-1)}}\big)\,\mathrm{d}t\right]; \qquad \tilde{t}_n \equiv t_n + \mathrm{d}t/2\,.$$

The independent control parameters are now the scalar values ϵ_n, respectively $\epsilon_{l,n}$ if there are multiple control fields indexed by l.

The GRAPE method looks at the direct gradient $\partial J/\partial\epsilon_n$ and updates each control parameter in the direction of that gradient [21]. The step width must be determined by a line search.

Typically, only the final time functional J_T has a nontrivial gradient. For simplicity, we assume that J_T can be expressed in terms of the complex overlaps $\{\tau_k\}$ between the target states $\{|\phi_k^{\mathrm{tgt}}\rangle\}$ and the propagated states $\{|\phi_k(T)\rangle\}$, as e.g. in Eqs. (7.7), (7.9). Using Eq. (11.5) leads to

$$\begin{aligned}\frac{\partial\tau_k}{\partial\epsilon_n} &= \frac{\partial}{\partial\epsilon_n}\big\langle\phi_k^{\mathrm{tgt}}\big|\hat{U}_{nt-1}^{(i-1)} \ldots \hat{U}_n^{(i-1)} \ldots \hat{U}_1^{(i-1)}\big|\phi_k\big\rangle \\ &= \underbrace{\big\langle\phi_k^{\mathrm{tgt}}\big|\hat{U}_{nt-1}^{(i-1)} \ldots \hat{U}_{n+1}^{(i-1)}}_{\langle\chi_k^{(i-1)}(t_{n+1})|} \frac{\partial\hat{U}_n^{(i-1)}}{\partial\epsilon_n} \underbrace{\hat{U}_{n-1}^{(i-1)} \ldots \hat{U}_1^{(i-1)}\big|\phi_k\big\rangle}_{|\phi_k^{(i-1)}(t_n)\rangle}\end{aligned} \tag{11.6}$$

as the gradient of these overlaps. The gradient for J_T, respectively J if there are additional running costs then follows from the chain rule. The numerical evaluation of Eq. (11.6) involves the backward-propagated states $|\chi_k(t_{n+1})\rangle$ and the forward-propagated states $|\phi_k(t_n)\rangle$. As only states from iteration $(i-1)$ enter in the gradient, GRAPE is a *concurrent* scheme.

The comparison of the sequential update equation (11.4) of Krotov's method and the concurrent update equation (11.2) has inspired a sequential evaluation of the "gradient", modifying the right-hand side of Eq. (11.6) to $\langle\chi_k^{(i-1)}(t_{n+1})|\partial_\epsilon U_n^{(i-1)}|\phi_k^{(i)}(t_n)\rangle$. That is, the states $\{|\phi_k(t)\rangle\}$ are forward-propagated under the optimized field [74]. This can be generalized to "hybrid" schemes that interleave concurrent and sequential calculation of the gradient [70]. An implementation of the concurrent/sequential/hybrid gradient is available in the DYNAMO Matlab package [70]. The sequential gradient scheme is sometimes referred to as "Krotov-type" [70][75]. To avoid confusion with the specific method defined in *Krotov's Method*, we prefer the name "sequential GRAPE".

GRAPE does not give a guarantee of monotonic convergence. As the optimization approaches the minimum of the functional, the first order gradient is generally insufficient to drive the optimization further [71]. To remedy this, a numerical estimate of the Hessian $\partial^2 J_T/\partial\epsilon_j\partial\epsilon_{j'}$ should also be included in the calculation of the update. The L-BFGS-B quasi-Newton method [76][77] is most commonly used for this purpose, resulting in the "Second-order

GRAPE" [78] or "GRAPE-LBFGS" method. L-BFGS-B is implemented as a Fortran library [77] and widely available, e.g. wrapped in optimization toolboxes like SciPy [79]. This means that it can be easily added as a "black box" to an existing gradient optimization. As a result, augmenting GRAPE with a quasi-Hessian is essentially "for free". Thus, we always mean GRAPE to refer to GRAPE-LBFGS. Empirically, GRAPE-LBFGS *usually* converges monotonically.

Thus, for (discretized) time-continuous controls, both GRAPE and Krotov's method can generally be used interchangeably. Historically, Krotov's method has been used primarily in the control of molecular dynamics, while GRAPE has been popular in the NMR community. Some potential benefits of Krotov's method compared to GRAPE are [71]:

- Krotov's method mathematically guarantees monotonic convergence in the continuous-time limit. There is no line-search required for the step width $1/\lambda_{a,l}$.
- The sequential nature of Krotov's update scheme, with information from earlier times entering the update at later times within the same iteration, results in faster convergence than the concurrent update in GRAPE [70][80]. This advantage disappears as the optimization approaches the optimum [71].
- The choice of functional J_T in Krotov's method only enters in the boundary condition for the backward-propagated states, Eq. (7.14), while the update equation stays the same otherwise. In contrast, for functionals J_T that do not depend trivially on the overlaps [81][82][83][84][85], the evaluation of the gradient in GRAPE may deviate significantly from its usual form, requiring a problem-specific implementation from scratch. This may be mitigated by the use of automatic differentiation in future implementations [86][87].

GRAPE has a significant advantage if the controls are not time-continuous, but are *physically* piecewise constant ("bang-bang control"). The calculation of the GRAPE-gradient is unaffected by this, whereas Krotov's method can break down when the controls are not approximately continuous. QuTiP contains an implementation of GRAPE limited to this use case.

Variants of gradient-ascent can be used to address *pulse parametrizations*. That is, the control parameters may be arbitrary parameters of the control field (e.g., spectral coefficients) instead of the field amplitude ϵ_n in a particular time interval. This is often relevant to design control fields that meet experimental constraints. One possible realization is to calculate the gradients for the control parameters from the gradients of the time-discrete control amplitudes via the chain rule [88][89]. This approach has recently been named "GRadient Optimization Using Parametrization" (GROUP) [90]. An alternative for a moderate number of control parameters is "gradient-optimization of analytic controls" (GOAT) [91]. GOAT evaluates the relevant gradient with forward-mode differentiation; that is, $\partial\tau_k/\partial\epsilon_n$ is directly evaluated alongside τ_k. For $N = |\{\epsilon_m\}|$ control parameters, this implies N forward propagations of the state-gradient pair per iteration. Alternatively, the N propagations can be concatenated into a single propagation in a Hilbert space enlarged by a factor N (the original state paired with N gradients).

A benefit of GOAT over the more general GROUP is that it does not piggy-back on the piecewise-constant discretization of the control field, and thus may avoid the associated numerical error. This allows to optimize to extremely high fidelities as required for some error correction protocols [91].

11.4 GRAPE in QuTiP

An implementation of GRAPE is included in QuTiP, see the section on Quantum Optimal Control in the QuTiP docs. It is used via the `qutip.control.pulseoptim.optimize_pulse()`

function. However, some of the design choices in QuTiP's GRAPE effectively limit the routine to applications with physically piecewise-constant pulses (where GRAPE has an advantage over Krotov's method, as discussed in the previous section).

For discretized time-continuous pulses, the implementation of Krotov's method in *optimize_pulses()* has the following advantages over qutip.control.pulseoptim.optimize_pulse():

- Krotov's method can optimize for more than one control field at the same time (hence the name of the routine *optimize_pulses()* compared to optimize_pulse()).
- Krotov's method optimizes a list of *Objective* instances simultaneously. The optimization for multiple simultaneous objectives in QuTiP's GRAPE implementation is limited to optimizing a quantum gate. Other uses of simultaneous objectives, such as optimizing for robustness, are not available.
- Krotov's method can start from an arbitrary set of guess controls. In the GRAPE implementation, guess pulses can only be chosen from a specific set of options (including "random"). Again, this makes sense for a control field that is piecewise constant with relatively few switching points, but is very disadvantageous for time-continuous controls.
- Krotov's method has complete flexibility in which propagation method is used (via the *propagator* argument to *optimize_pulses()*), while QuTiP's GRAPE only allows to choose between fixed number of methods for time-propagation. Supplying a problem-specific propagator is not possible.

Thus, QuTiP's GRAPE implementation and the implementation of Krotov's method in this package complement each other, but will not compare directly.

11.5 Gradient-free optimization

In situations where the problem can be reduced to a relatively small number of control parameters (typically less than ≈ 20, although this number may be pushed to ≈ 50 by sequential increase of the number of parameters and re-parametrization [92][93]), gradient-free optimization becomes feasible. The most straightforward use case are controls with an analytic shape (e.g. due to the constraints of an experimental setup), with just a few free parameters. As an example, consider control pulses that are restricted to a Gaussian shape, so that the only free parameters are peak amplitude, pulse width and delay. The control parameters are not required to be parameters of a time-dependent control, but may also be static parameters in the Hamiltonian, e.g. the polarization of the laser beams utilized in an experiment [94].

A special case of gradient-free optimization is the Chopped RAndom Basis (CRAB) method [95][96]. The essence of CRAB is in the specific choice of the parametrization in terms of a low-dimensional *random* basis, as the name implies. Thus, it can be used when the parametrization is not pre-defined as in the case of direct free parameters in the pulse shape discussed above. The optimization itself is normally performed by Nelder-Mead simplex based on this parametrization, although any other gradient-free method could be used as well. An implementation of CRAB is available in QuTiP, see QuTiP's documentation of CRAB, and uses the same qutip.control.pulseoptim.optimize_pulse() interface as the GRAPE method discussed above (*GRAPE in QuTiP*) with the same limitations. CRAB is prone to getting stuck in local minima of the optimization landscape. To remedy this, a variant of CRAB, "dressed CRAB" (DCRAB) has been developed [92] that re-parametrizes the controls when this happens.

Gradient-free optimization does not require backward propagation, only forward propagation of the initial states and evaluation of the optimization functional J. The functional is not

required to be analytic. It may be of a form that does not allow calculation of the gradients $\partial J_T / \partial \langle \phi_k |$ (Krotov's method) or $\partial J / \partial \epsilon_j$ (GRAPE). The optimization also does not require any storage of states. However, the number of iterations can grow extremely large, especially with an increasing number of control parameters. Thus, an optimization with a gradient-free method is not necessarily more efficient overall compared to a gradient-based optimization with much faster convergence. For only a few parameters, however, it can be highly efficient. This makes gradient-free optimization useful for "pre-optimization", that is, for finding guess controls that are then further optimized with a gradient-based method [31].

Generally, gradient-free optimization can be easily realized directly in QuTiP or any other software package for the simulation of quantum dynamics:

- Write a function that takes an array of optimization parameters as input and returns a figure of merit. This function would, e.g., construct a numerical control pulse from the control parameters, simulate the dynamics using qutip.mesolve.mesolve, and evaluate a figure of merit (like the overlap with a target state).
- Pass the function to scipy.optimize.minimize for gradient-free optimization.

The implementation in `scipy.optimize.minimize()` allows to choose between different optimization methods, with Nelder-Mead simplex being the default. There exist also more advanced optimization methods available in packages like NLopt [97] or Nevergrad [98] that may be worth exploring for improvements in numerical efficiency and additional functionality such as support for non-linear constraints.

11.6 Choosing an optimization method

In the following, we discuss some of the concerns in the choice of optimization methods. The discussion is limited to iterative open-loop methods, where the optimization is based on a numerical simulation of the dynamics. It excludes analytical control methods such as geometric control, closed-loop methods, or coherent feedback control; see Ref. [99] for an overview.

Whether to use a gradient-free optimization method, GRAPE, or Krotov's method depends on the size of the problem, the requirements on the control fields, and the mathematical properties of the optimization functional. Gradient-free methods should be used if the number of independent control parameters is smaller than ≈ 20, or the functional is of a form that does not allow to calculate gradients easily. It is always a good idea to use a gradient-free method to obtain improved guess pulses for use with a gradient-based method [31].

GRAPE or its variants should be used if the control parameters are discrete, such as on a coarse-grained time grid, and the derivative of J with respect to each control parameter is easily computable. Note that the implementation provided in QuTiP is limited to state-to-state transitions and quantum gates, even though the method is generally applicable to a wider range of objectives.

When the control parameters are general analytic coefficients instead of time-discrete amplitudes, the GROUP [88][89][90] or GOAT [91] variant of gradient-ascent may be a suitable choice. GOAT in particular can avoid the numerical error associated with time discretization. However, as the method scales linearly in memory and/or CPU with the number of control parameters, this is best used when then number of parameters is below 100.

Krotov's method should be used if the control is close to time-continuous, and if the derivative of J_T with respect to the states, Eq. (7.14), can be calculated. When these conditions are met, Krotov's method gives excellent convergence. The general family of monotonically convergent iteration schemes [65] may also be used.

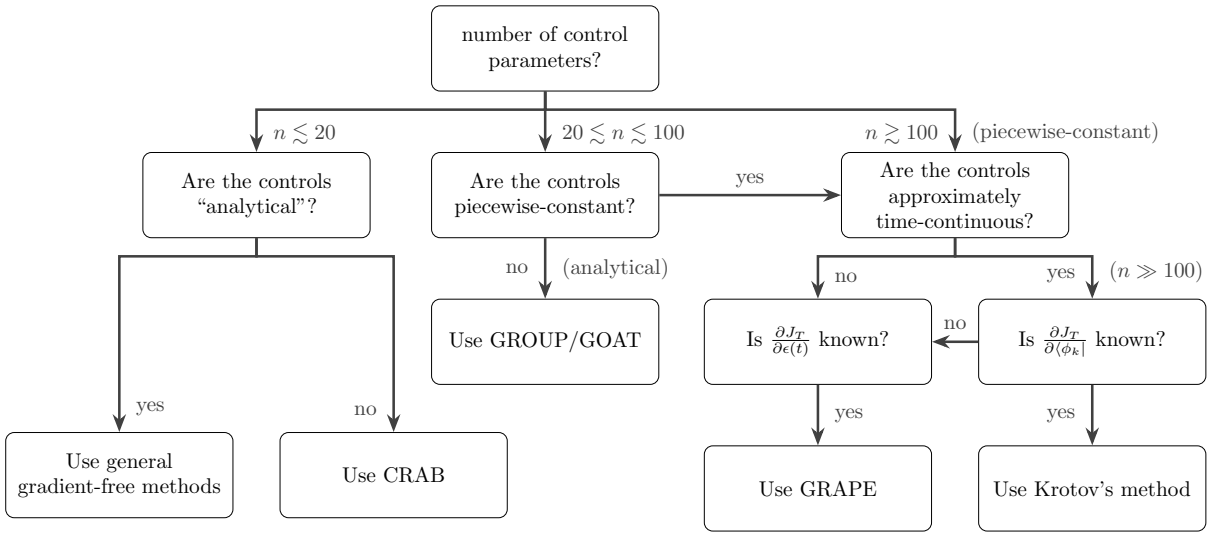


Fig. 11.1: Decision tree for the choice of a numerical open-loop optimization method. For "piecewise-constant controls", the control parameters are the values of the control field in each time interval. For "analytical" controls, we assume that the control fields are described by a fixed analytical formula parametrized by the control parameters. The "non-analytical" controls for CRAB refer to the *random* choice of a fixed number of spectral components, where the control parameters are the coefficients for those spectral components. Each method in the diagram is meant to include all its variants, a multitude of gradient-free methods and e.g. DCRAB for CRAB, GRAPE-LBFGS and sequential/hybrid gradient-descent for GRAPE, and K-BFGS for Krotov, see text for detail.

The decision tree in Fig. 11.1 can guide the choice of an optimization method. The key deciding factors are the number of control parameters and whether the controls are time-discrete. Of course, the parametrization of the controls is itself a choice. Sometimes, experimental constraints only allow controls that depend on a small number of tunable parameters. However, this necessarily limits the exploration of the full physical optimization landscape. At the other end of the spectrum, arbitrary time-continuous controls such as those assumed in Krotov's method have no inherent constraints and are especially useful for more fundamental tasks, such as mapping the design landscape of a particular system [100] or determining the quantum speed limit, i.e., the minimum time in which the system can reach a given target [101][102][15].

12 API

12.1 krotov package

The main function exposed here is *optimize_pulses()*.

This acts on a list of *Objective* instances, which may either be constructed manually, or through the helper functions *gate_objectives()* and *ensemble_objectives()*.

The submodules contain various auxiliary functions for constructing arguments for *optimize_pulses()*, for common use cases. This includes functions for constructing control fields (guess pulses), optimization functionals, convergence checks, analysis tools, and propagators, as well as more technical routines for parallelization, low-level data conversion, and estimators for second-order updates.

Submodules:

12.1.1 krotov.convergence module

Routines for *check_convergence* in *krotov.optimize.optimize_pulses()*

A *check_convergence* function may be used to determine whether an optimization is converged, and thus can be stopped before the maximum number of iterations (*iter_stop*) is reached. A function suitable for *check_convergence* must receive a *Result* object, and return a value that evaluates as True or False in a Boolean context, indicating whether the optimization has converged or not.

The *Result* object that the *check_convergence* function receives as an argument will be up-to-date for the current iteration. That is, it will already contain the current values from *optimize_pulses()*'s *info_hook* in *Result.info_vals*, the current *tau_vals*, etc. The *Result.optimized_controls* attribute will contain the current optimized pulses (defined on the intervals of *tlist*).

The *check_convergence* function must not modify the *Result* object it receives in any way. The proper place for custom modifications after each iteration in *optimize_pulses()* is through the *modify_params_after_iter* routine (e.g., dynamically adjusting λ_a if convergence is too slow or pulse updates are too large).

It is recommended that a *check_convergence* function returns None (which is False in a Boolean context) if the optimization has not yet converged. If the optimization has converged,

check_convergence should return a message string (which is True in a Boolean context). The returned string will be included in the final *Result.message*.

A typical usage for *check_convergence* is ending the optimization when the error falls below a specified limit. Such a *check_convergence* function can be generated by *value_below()*. Often, this "error" is the value of the functional J_T. However, it is up to the user to ensure that the explicit value of J_T can be calculated; J_T in Krotov's method is completely implicit, and enters the optimization only indirectly via the *chi_constructor* passed to *optimize_pulses()*. A specific *chi_constructor* implies the minimization of the functional J_T from which *chi_constructor* was derived. A convergence check based on the *explicit* value of J_T can be realized by passing an *info_hook* that returns the value of J_T. This value is then stored in *Result.info_vals*, which is where *value_below()* looks for it.

An *info_hook* could also calculate and return an arbitrary measure of *success*, not related to J_T (e.g. a fidelity, or a concurrence). Since we expect the optimization (the minimization of J_T) to maximize a fidelity, a convergence check might want to look at whether the calculated value is *above* some threshold. This can be done via *value_above()*.

In addition to looking at the *value* of some figure of merit, one might want stop the optimization when there is an insufficient improvement between iterations. The *delta_below()* function generates a *check_convergence* function for this purpose. Multiple convergence conditions ("stop optimization when J_T reaches 10^{-5}, or if $\Delta J_T < 10^{-6}$") can be defined via *Or()*.

While Krotov's method is guaranteed to monotonically converge in the continuous limit, this no longer strictly holds when time is discretized (in particular if λ_a is too small). You can use *check_monotonic_error()* or *check_monotonic_fidelity()* as a *check_convergence* function that stops the optimization when monotonic convergence is lost.

The *check_convergence* routine may also be used to store the current state of the optimization to disk, as a side effect. This is achieved by the routine *dump_result()*, which can be chained with other convergence checks with *Or()*. Dumping the current state of the optimization at regular intervals protects against losing the results of a long running optimization in the event of a crash.

Summary

Functions:

Or	Chain multiple *check_convergence* functions together in a logical Or.
check_monotonic_error	Check for monotonic convergence with respect to the error
check_monotonic_fidelity	Check for monotonic convergence with respect to the fidelity
delta_below	Constructor for a routine that checks if $\lvert v_1 - v_0 \rvert < \varepsilon$
dump_result	Return a function for dumping the result every so many iterations
value_above	Constructor for routine that checks if a value is above *limit*
value_below	Constructor for routine that checks if a value is below *limit*

__all__: *Or*, *check_monotonic_error*, *check_monotonic_fidelity*, *delta_below*,

dump_result, *value_above*, *value_below*

Reference

krotov.convergence.**Or**(**funcs*)

Chain multiple *check_convergence* functions together in a logical Or.

Each parameter must be a function suitable to pass to *optimize_pulses()* as *check_convergence*. It must receive a *Result* object and should return None or a string message.

Returns A function `check_convergence(result)` that returns the result of the first "non-passing" function in **funcs*. A "non-passing" result is one that evaluates to True in a Boolean context (should be a string message)

Return type callable

krotov.convergence.**value_below**(*limit*, *spec=('info_vals', T[-1])*, *name=None*, ***kwargs*)

Constructor for routine that checks if a value is below *limit*

Parameters

- **limit** (*float or str*) – A float value (or str-representation of a float) against which to compare the value extracted from *Result*
- **spec** – A specification of the *Result* attribute from which to extract the value to compare against *limit*. Defaults to a specification extracting the last value in *Result.info_vals* (returned by the *info_hook* passed to *optimize_pulses()*). This should be some kind of error measure, e.g., the value of the functional J_T that is being minimized.
- **name** (*str or None*) – A name identifying the checked value, used for the message returned by the *check_convergence* routine. Defaults to `str(spec)`.
- ****kwargs** – Keyword arguments to pass to `glom()` (see Note)

Returns A function `check_convergence(result)` that extracts the value specified by *spec* from the *Result* object, and checks it against *limit*. If the value is below the *limit*, it returns an appropriate message string. Otherwise, it returns None.

Return type callable

Note: The *spec* can be a callable that receives *Result* and returns the value to check against the limit. You should also pass a *name* like 'J_T', or 'error' as a label for the value. For more advanced use cases, *spec* can be a `glom()`-specification that extracts the value to check from the *Result* object as `glom.glom(result, spec, **kwargs)`.

Example

```
>>> check_convergence = value_below(
...     limit='1e-4',
...     spec=lambda r: r.info_vals[-1],  # same as the default spec
...     name='J_T'
```

(continues on next page)

(continued from previous page)

```
... )
>>> r = krotov.result.Result()
>>> r.info_vals.append(1e-4)
>>> check_convergence(r)  # returns None
>>> r.info_vals.append(9e-5)
>>> check_convergence(r)
'J_T < 1e-4'
```

krotov.convergence.**value_above**(*limit*, *spec=('info_vals', T[-1])*, *name=None*, ***kwargs*)

Constructor for routine that checks if a value is above *limit*

Like *value_below()*, but for checking whether an extracted value is *above*, not below a value. By default, it looks at the last value in *Result.info_vals*, under the assumption that the *info_hook* passed to *optimize_pulses()* returns some figure of merit we expect to be maximized, like a fidelity. Note that an *info_hook* is free to return an arbitrary value, not necessarily the value of the functional J_T that the optimization is minimizing (specified implicitly via the *chi_constructor* argument to *optimize_pulses()*).

Example

```
>>> check_convergence = value_above(
...     limit='0.999',
...     spec=lambda r: r.info_vals[-1],
...     name='Fidelity'
... )
>>> r = krotov.result.Result()
>>> r.info_vals.append(0.9)
>>> check_convergence(r)  # returns None
>>> r.info_vals.append(1 - 1e-6)
>>> check_convergence(r)
'Fidelity > 0.999'
```

krotov.convergence.**delta_below**(*limit*, *spec1=('info_vals', T[-1])*, *spec0=('info_vals', T[-2])*, *absolute_value=True*, *name=None*, ***kwargs*)

Constructor for a routine that checks if $|v_1 - v_0| < \varepsilon$

Parameters

- **limit** (*float or str*) – A float value (or str-representation of a float) for ε
- **spec1** – A `glom()` specification of the *Result* attribute from which to extract v_1. Defaults to a spec extracting the last value in *Result.info_vals*.
- **spec0** – A `glom()` specification of the *Result* attribute from which to extract v_0. Defaults to a spec extracting the last-but-one value in *Result.info_vals*.
- **absolute_value** (*bool*) – If False, check for $v_1 - v_0 < \varepsilon$, instead of the absolute value.
- **name** (*str or None*) – A name identifying the delta, used for the message returned by the *check_convergence* routine. Defaults to `"Δ({spec1}, {spec0}"`.

- ****kwargs** – Keyword arguments to pass to `glom()`

Note: You can use *delta_below()* to implement a check for strict monotonic convergence, e.g. when *info_hook* returns the optimization error, by flipping *spec0* and *spec1*, setting *limit* to zero, and setting *absolute_value* to False. See *check_monotonic_error()*.

Example

```
>>> check_convergence = delta_below(limit='1e-4', name='ΔJ_T')
>>> r = krotov.result.Result()
>>> r.info_vals.append(9e-1)
>>> check_convergence(r)  # None
>>> r.info_vals.append(1e-1)
>>> check_convergence(r)  # None
>>> r.info_vals.append(4e-4)
>>> check_convergence(r)  # None
>>> r.info_vals.append(2e-4)
>>> check_convergence(r)  # None
>>> r.info_vals.append(1e-6)
>>> check_convergence(r)  # None
>>> r.info_vals.append(1e-7)
>>> check_convergence(r)
'ΔJ_T < 1e-4'
```

krotov.convergence.**check_monotonic_error**(*result*)

Check for monotonic convergence with respect to the error

Check that the last value in *Result.info_vals* is smaller than the last-but-one value. If yes, return None. If no, return an appropriate error message.

This assumes that the *info_hook* passed to *optimize_pulses()* returns the value of the functional J_T (or another quantity that we expect to be minimized), which is then available in *Result.info_vals*.

Example

```
>>> r = krotov.result.Result()
>>> r.info_vals.append(9e-1)
>>> check_monotonic_error(r)  # None
>>> r.info_vals.append(1e-1)
>>> check_monotonic_error(r)  # None
>>> r.info_vals.append(2e-1)
>>> check_monotonic_error(r)
'Loss of monotonic convergence; error decrease < 0'
```

See also:

Use *check_monotonic_fidelity()* for when *info_hook* returns a "fidelity", that is, a measure that should *increase* in each iteration.

krotov.convergence.**check_monotonic_fidelity**(*result*)

Check for monotonic convergence with respect to the fidelity

This is like *check_monotonic_error()*, but looking for a monotonic *increase* in the values in *Result.info_vals*. Thus, it is assumed that the *info_hook* returns a fidelity (to be maximized), not an error (like J_T, to be minimized).

Example

```
>>> r = krotov.result.Result()
>>> r.info_vals.append(0.0)
>>> check_monotonic_fidelity(r)  # None
>>> r.info_vals.append(0.2)
>>> check_monotonic_fidelity(r)  # None
>>> r.info_vals.append(0.15)
>>> check_monotonic_fidelity(r)
'Loss of monotonic convergence; fidelity increase < 0'
```

krotov.convergence.**dump_result**(*filename*, *every=10*)

Return a function for dumping the result every so many iterations

For long-running optimizations, it can be useful to dump the current state of the optimization every once in a while, so that the result is not lost in the event of a crash or unexpected shutdown. This function returns a routine that can be passed as a *check_convergence* routine that does nothing except to dump the current *Result* object to a file (cf. *Result.dump()*). Failure to write the dump file stops the optimization.

Parameters

- **filename** (*str*) – Name of file to dump to. This may include a field `{iter}` which will be formatted with the most recent iteration number, via `str.format()`. Existing files will be overwritten.
- **every** (*int*) – dump the *Result* every so many iterations.

Note: Choose *every* so that dumping does not happen more than once every few minutes, at most. Dumping after every single iteration may slow down the optimization due to I/O overhead.

Examples

- dump every 10 iterations to the same file *oct_result.dump*:

```
>>> check_convergence = dump_result('oct_result.dump')
```

- dump every 100 iterations to files `oct_result_000100.dump`, `oct_result_000200.dump`, etc.:

```
>>> check_convergence = dump_result(
...     'oct_result_{iter:06d}.dump', every=100)
```

12.1.2 krotov.functionals module

Functionals and *chi_constructor* routines.

Any *chi_constructor* routine passed to *optimize_pulses()* must take the following keyword-arguments:

- *fw_states_T* (list of Qobj): The list of states resulting from the forward-propagation of each *Objective.initial_state* under the guess pulses of the current iteration (the optimized pulses of the previous iteration)
- *objectives* (list of *Objective*): A list of the optimization objectives.
- *tau_vals* (list of complex or None): The overlaps of the *Objective.target* and the corresponding *fw_states_T*, assuming *Objective.target* contains a quantum state. If the objective defines no target state, a list of Nones

Krotov's method does not have an explicit dependence on the optimization functional. It only enters through the *chi_constructor* which calculates the boundary condition for the backward propagation, that is, the states

$$\left|\chi_k^{(i)}(T)\right\rangle = -\left.\frac{\partial J_T}{\partial \langle\phi_k|}\right|_{\phi^{(i)}(T)}$$

for functionals defined in Hilbert space, or

$$\hat{\chi}_k^{(i)}(T) = -\left.\frac{\partial J_T}{\partial \langle\langle\hat{\rho}_k|}\right|_{\rho^{(i)}(T)}$$

in Liouville space, using the abstract Hilbert-Schmidt notation $\langle\langle a|b\rangle\rangle \equiv \mathrm{tr}[a^\dagger b]$. Passing a specific *chi_constructor* results in the minimization of the final time functional from which that *chi_constructor* was derived.

The functions in this module that evaluate functionals are intended for use inside a function that is passed as an *info_hook* to *optimize_pulses()*. Thus, they calculate J_T from the same keyword arguments as the *info_hook*. The values for J_T may be used in a convergence analysis, see *krotov.convergence*.

Summary

Functions:

F_avg	Average gate fidelity
F_re	Real-part fidelity
F_sm	Square-modulus fidelity
F_ss	State-to-state phase-insensitive fidelity
J_T_hs	Hilbert-Schmidt distance measure functional $J_{T,\mathrm{hs}}$
J_T_re	Real-part functional $J_{T,\mathrm{re}}$
J_T_sm	Square-modulus functional $J_{T,\mathrm{sm}}$
J_T_ss	State-to-state phase-insensitive functional $J_{T,\mathrm{ss}}$
chis_hs	States $\hat{\chi}_k$ for functional $J_{T,\mathrm{hs}}$
chis_re	States $\|\chi_k\rangle$ for functional $J_{T,\mathrm{re}}$
chis_sm	States $\|\chi_k\rangle$ for functional $J_{T,\mathrm{sm}}$
chis_ss	States $\|\chi_k\rangle$ for functional $J_{T,\mathrm{ss}}$
f_tau	Average complex overlaps of the target states with the *fw_states_T*.
gate	Gate that maps *basis_states* to *fw_states_T*
mapped_basis	Result of applying the gate *O* to *basis_states*

__all__: *F_avg*, *F_re*, *F_sm*, *F_ss*, *J_T_hs*, *J_T_re*, *J_T_sm*, *J_T_ss*, *chis_hs*, *chis_re*, *chis_sm*, *chis_ss*, *f_tau*, *gate*, *mapped_basis*

Reference

krotov.functionals.**f_tau**(*fw_states_T*, *objectives*, *tau_vals=None*, ***kwargs*)

Average complex overlaps of the target states with the *fw_states_T*.

That is,

$$f_\tau = \frac{1}{N} \sum_{k=1}^{N} w_k \tau_k$$

where τ_k are the elements of *tau_vals*, assumed to be

$$\tau_k = \left\langle \Psi_k^{\text{tgt}} \mid \Psi_k(T) \right\rangle,$$

in Hilbert space, or

$$\tau_k = \mathrm{tr}\left[\hat{\rho}_k^{\text{tgt}\,\dagger} \hat{\rho}_k(T) \right]$$

in Liouville space, where $|\Psi_k\rangle$ or $\hat{\rho}_k$ are the elements of *fw_states_T*, and $|\Psi_k^{\text{tgt}}\rangle$ or $\hat{\rho}^{\text{tgt}}$ are the target states from the *target* attribute of the objectives. If *tau_vals* are None, they will be calculated internally.

N is the number of objectives, and w_k is an optional weight for each objective. For any objective that has a (custom) *weight* attribute, the w_k is taken from that attribute; otherwise, $w_k = 1$. The weights, if present, are not automatically normalized, they are assumed to have values such that the resulting f_τ lies in the unit circle of the complex plane. Usually, this means that the weights should sum to N. The exception would be for mixed target states, where the weights should compensate for the non-unit purity. The problem may be circumvented by using *J_T_hs()* for mixed target states.

The *kwargs* are ignored, allowing the function to be used in an *info_hook*.

krotov.functionals.**F_ss**(*fw_states_T*, *objectives*, *tau_vals=None*, ***kwargs*)

State-to-state phase-insensitive fidelity

$$F_{\text{ss}} = \frac{1}{N} \sum_{k=1}^{N} w_k \left|\tau_k\right|^2 \quad \in [0, 1]$$

with N, w_k and τ_k as in *f_tau()*.

The *kwargs* are ignored, allowing the function to be used in an *info_hook*.

krotov.functionals.**J_T_ss**(*fw_states_T*, *objectives*, *tau_vals=None*, ***kwargs*)

State-to-state phase-insensitive functional $J_{T,\text{ss}}$

$$J_{T,\text{ss}} = 1 - F_{\text{ss}} \quad \in [0, 1].$$

All arguments are passed to *F_ss()*.

krotov.functionals.**chis_ss**(*fw_states_T*, *objectives*, *tau_vals*)

States $|\chi_k\rangle$ for functional $J_{T,\text{ss}}$

$$|\chi_k\rangle = -\frac{\partial J_{T,\text{ss}}}{\partial \left\langle \Psi_k(T) \right|} = \frac{1}{N} w_k \tau_k \left| \Psi_k^{\text{tgt}} \right\rangle$$

with τ_k and w_k as defined in *f_tau()*.

krotov.functionals.**F_sm**(*fw_states_T*, *objectives*, *tau_vals=None*, ***kwargs*)
Square-modulus fidelity

$$F_{\mathrm{sm}} = |f_\tau|^2 \quad \in [0, 1].$$

All arguments are passed to *f_tau()* to evaluate f_τ.

krotov.functionals.**J_T_sm**(*fw_states_T*, *objectives*, *tau_vals=None*, ***kwargs*)
Square-modulus functional $J_{T,\mathrm{sm}}$

$$J_{T,\mathrm{sm}} = 1 - F_{\mathrm{sm}} \quad \in [0, 1]$$

All arguments are passed to *f_tau()* while evaluating F_{sm} in *F_sm()*.

krotov.functionals.**chis_sm**(*fw_states_T*, *objectives*, *tau_vals*)
States $|\chi_k\rangle$ for functional $J_{T,\mathrm{sm}}$

$$|\chi_k\rangle = -\frac{\partial J_{T,\mathrm{sm}}}{\partial \langle\Psi_k(T)|} = \frac{1}{N^2} w_k \sum_j^N w_j \tau_j \left|\Psi_k^{\mathrm{tgt}}\right\rangle$$

with optional weights w_k, cf. *f_tau()* (default: $w_k = 1$). If given, the weights should generally sum to N.

krotov.functionals.**F_re**(*fw_states_T*, *objectives*, *tau_vals=None*, ***kwargs*)
Real-part fidelity

$$F_{\mathrm{re}} = \mathrm{Re}[f_\tau] \quad \in \begin{cases} [-1, 1] & \text{in Hilbert space} \\ [0, 1] & \text{in Liouville space.} \end{cases}$$

All arguments are passed to *f_tau()* to evaluate f_τ.

krotov.functionals.**J_T_re**(*fw_states_T*, *objectives*, *tau_vals=None*, ***kwargs*)
Real-part functional $J_{T,\mathrm{re}}$

$$J_{T,\mathrm{re}} = 1 - F_{\mathrm{re}} \quad \in \begin{cases} [0, 2] & \text{in Hilbert space} \\ [0, 1] & \text{in Liouville space.} \end{cases}$$

All arguments are passed to *f_tau()* while evaluating F_{re} in *F_re()*.

Note: If the target states are mixed, $J_{T,\mathrm{re}}$ may take negative values (for *fw_states_T* that are "in the right direction", but more pure than the target states). In this case, you may consider using *J_T_hs()*.

---

krotov.functionals.**chis_re**(*fw_states_T*, *objectives*, *tau_vals*)
States $|\chi_k\rangle$ for functional $J_{T,\mathrm{re}}$

$$|\chi_k\rangle = -\frac{\partial J_{T,\mathrm{re}}}{\partial \langle\Psi_k(T)|} = \frac{1}{2N} w_k \left|\Psi_k^{\mathrm{tgt}}\right\rangle$$

with optional weights w_k, cf. *f_tau()* (default: $w_k = 1$). If given, the weights should generally sum to N.

Note: *tau_vals* are ignored, but are present to satisfy the requirments of the *chi_constructor* interface.

krotov.functionals.**J_T_hs**(*fw_states_T*, *objectives*, *tau_vals=None*, ***kwargs*)

Hilbert-Schmidt distance measure functional $J_{T,\mathrm{hs}}$

$$J_{T,\mathrm{hs}} = \frac{1}{2N} \sum_{k=1}^{N} w_k \left\| \hat{\rho}_k(T) - \hat{\rho}_k^{\mathrm{tgt}} \right\|_{\mathrm{hs}}^2 \quad \in \begin{cases} [0,2] & \text{in Hilbert space} \\ [0,1] & \text{in Liouville space} \end{cases}$$

in Liouville space (using the Hilbert-Schmidt norm), or equivalently with $|\Psi_k(T)\rangle$ and $|\Psi_k^{tgt}\rangle$ in Hilbert space. The functional is evaluated as

$$J_{T,\mathrm{hs}} = \frac{1}{2N} \sum_{k=1}^{N} w_k \left(\|\hat{\rho}_k(T)\|_{\mathrm{hs}}^2 + \left\|\hat{\rho}^{\mathrm{tgt}}\right\|_{\mathrm{hs}}^2 - 2\,\mathrm{Re}[\tau_k] \right)$$

where the $\hat{\rho}_k$ are the elements of *fw_states_T*, the $\hat{\rho}_k^{\mathrm{tgt}}$ are the target states from the *target* attribute of the objectives, and the τ_k are the elements of *tau_vals* (which will be calculated internally if passed as None).

The w_k are optional weights, cf. *f_tau()*. If given, the weights should generally sum to N.

The *kwargs* are ignored, allowing the function to be used in an *info_hook*.

Note: For pure states (or Hilbert space states), $J_{T,\mathrm{hs}}$ is equivalent to $J_{T,\mathrm{re}}$, cf. *J_T_re()*. However, the backward-propagated states χ_k obtained from the two functionals (*chis_re()* and *chis_hs()*) are *not* equivalent. This may result in a vastly different optimization landscape that requires a significantly different value of the λ_a value that regulates the overall magnitude of the pulse updates (given in *pulse_options* in *optimize_pulses()*).

---

krotov.functionals.**chis_hs**(*fw_states_T*, *objectives*, *tau_vals*)

States $\hat{\chi}_k$ for functional $J_{T,\mathrm{hs}}$

$$\hat{\chi}_k = -\frac{\partial J_{T,\mathrm{sm}}}{\partial \langle\!\langle \hat{\rho}_k(T) |} = \frac{1}{2N} w_k \left(\hat{\rho}_k^{\mathrm{tgt}} - \hat{\rho}_k(T) \right)$$

with optional weights w_k, cf. *f_tau()* (default: $w_k = 1$).

This is derived from $J_{T,\mathrm{hs}}$ rewritten in the abstract Hilbert-Schmidt notation $\langle\!\langle a | b \rangle\!\rangle \equiv \mathrm{tr}[a^\dagger b]$:

$$J_{T,\mathrm{hs}} = \frac{-1}{2N} \sum_{k=1}^{N} w_k \Big(\underbrace{\langle\!\langle \hat{\rho}_k(T) | \hat{\rho}_k^{\mathrm{tgt}} \rangle\!\rangle + \langle\!\langle \hat{\rho}_k^{\mathrm{tgt}} | \hat{\rho}_k(T) \rangle\!\rangle}_{=2\,\mathrm{Re}[\tau_k]} - \underbrace{\langle\!\langle \hat{\rho}_k(T) | \hat{\rho}_k(T) \rangle\!\rangle}_{=\|\hat{\rho}_k(T)\|_{\mathrm{hs}}^2} - \underbrace{\langle\!\langle \hat{\rho}_k^{\mathrm{tgt}} | \hat{\rho}_k^{\mathrm{tgt}} \rangle\!\rangle}_{=\|\hat{\rho}^{\mathrm{tgt}}\|_{\mathrm{hs}}^2} \Big).$$

Note: *tau_vals* are ignored, but are present to satisfy the requirments of the *chi_constructor* interface.

krotov.functionals.**F_avg**(*fw_states_T*, *basis_states*, *gate*, *mapped_basis_states=None*, *prec=1e-05*)

Average gate fidelity

$$F_{\mathrm{avg}} = \int \left\langle \Psi \middle| \hat{O}^\dagger \mathcal{E}[|\Psi\rangle\langle\Psi|] \hat{O} \middle| \Psi \right\rangle \mathrm{d}\Psi$$

where $\hat{O}$ is the target *gate*, and $\mathcal{E}$ represents the dynamical map from time zero to T.

In Liouville space, this is numerically evaluated as

$$F_{\text{avg}} = \frac{1}{N(N+1)} \sum_{i,j=1}^{N} \left(\langle \phi_i | \hat{O}^\dagger \hat{\rho}_{ij} \hat{O} | \phi_j \rangle + \langle \phi_i | \hat{O}^\dagger \hat{\rho}_{jj} \hat{O} | \phi_i \rangle \right),$$

where $|\phi_i\rangle$ is the i'th element of *basis_states*, and $\hat{\rho}_{ij}$ is the $(i-1)N + j$'th element of *fw_states_T*, that is, $\hat{\rho}_{ij} = \mathcal{E}[|\phi_i\rangle\langle\phi_j|]$, with N the dimension of the Hilbert space.

In Hilbert space (unitary dynamics), this simplifies to

$$F_{\text{avg}} = \frac{1}{N(N+1)} \left(\left| \text{tr}\left[\hat{O}^\dagger \hat{U}\right] \right|^2 + \text{tr}\left[\hat{O}^\dagger \hat{U} \hat{U}^\dagger \hat{O}\right] \right),$$

where $\hat{U}$ the gate that maps *basis_states* to the result of a forward propagation of those basis states, stored in *fw_states_T*.

Parameters

- **fw_states_T** (*list[qutip.Qobj]*) – The forward propagated states. For dissipative dynamics, this must be the forward propagation of the full basis of Liouville space, that is, all N^2 dyadic combinations of the Hilbert space logical basis states. For unitary dynamics, the N forward-propagated *basis_states*.
- **basis_states** (*list[qutip.Qobj]*) – The N Hilbert space logical basis states
- **gate** (*qutip.Qobj*) – The $N \times N$ quantum gate in the logical subspace, e.g. `qutip.qip.gates.cnot()`.
- **mapped_basis_states** (*None or list[qutip.Qobj]*) – If given, the result of applying gate to *basis_states*. If not given, this will be calculated internally via *mapped_basis()*. It is recommended to pass pre-calculated *mapped_basis_states* when evaluating F_{avg} repeatedly for the same target.
- **prec** (*float*) – assert that the fidelity is correct at least up to the given precision. Mathematically, F_{avg} is a real value. However, errors in the *fw_states_T* can lead to a small non-zero imaginary part. We assert that this imaginary part is below *prec*.

krotov.functionals.**gate**(*basis_states*, *fw_states_T*)
Gate that maps *basis_states* to *fw_states_T*

Example

```
>>> from qutip import ket
>>> basis = [ket(nums) for nums in [(0, 0), (0, 1), (1, 0), (1, 1)]]
>>> fw_states_T = mapped_basis(qutip.gates.cnot(), basis)
>>> U = gate(basis, fw_states_T)
>>> assert (U - qutip.gates.cnot()).norm() < 1e-15
```

krotov.functionals.**mapped_basis**(*O*, *basis_states*)
Result of applying the gate *O* to *basis_states*

Example

```
>>> from qutip import ket
>>> basis = [ket(nums) for nums in [(0, 0), (0, 1), (1, 0), (1, 1)]]
>>> states = mapped_basis(qutip.gates.cnot(), basis)
>>> assert (states[0] - ket((0, 0))).norm() < 1e-15
>>> assert (states[1] - ket((0, 1))).norm() < 1e-15
>>> assert (states[2] - ket((1, 1))).norm() < 1e-15  # swap (1, 1) ...
>>> assert (states[3] - ket((1, 0))).norm() < 1e-15  # ... and (1, 0)
```

12.1.3 krotov.info_hooks module

Routines that can be passed as *info_hook* to *optimize_pulses()*

Summary

Functions:

chain	Chain multiple *info_hook* or *modify_params_after_iter* callables together.
print_debug_information	Print full debug information about the current Krotov iteration
print_table	Print a tabular overview of the functional values in the iteration

__all__: *chain*, *print_debug_information*, *print_table*

Reference

krotov.info_hooks.**chain**(**hooks*)

Chain multiple *info_hook* or *modify_params_after_iter* callables together.

Example

```
>>> def print_fidelity(**kwargs):
...     F_re = np.average(np.array(kwargs['tau_vals']).real)
...     print("    F = %f" % F_re)
>>> info_hook = chain(print_debug_information, print_fidelity)
```

Note: Functions that are connected via *chain()* may use the *shared_data* share the same *shared_data* argument, which they can use to communicate down the chain.

krotov.info_hooks.**print_debug_information**(*, *objectives*, *adjoint_objectives*, *backward_states*, *forward_states*, *forward_states0*, *guess_pulses*, *optimized_pulses*, *g_a_integrals*, *lambda_vals*, *shape_arrays*, *fw_states_T*, *tlist*, *tau_vals*, *start_time*, *stop_time*, *iteration*, *info_vals*, *shared_data*, *out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Print full debug information about the current Krotov iteration

This routine is intended to be passed to *optimize_pulses()* as *info_hook*, and it exemplifies the full signature of a routine suitable for this purpose.

Keyword Arguments

- **objectives** (*list[Objective]*) – list of the objectives
- **adjoint_objectives** (*list[Objective]*) – list of the adjoint objectives
- **backward_states** (*list*) – If available, for each objective, an array-like object containing the states of the Krotov backward propagation.
- **forward_states** – If available (second order only), for each objective, an array-like object containing the forward-propagated states under the optimized pulses. None otherwise.
- **forward_states0** – If available (second order only), for each objective, an array-like object containing the forward-propagated states under the guess pulses. None otherwise.
- **guess_pulses** (*list[numpy.ndarray]*) – list of guess pulses
- **optimized_pulses** (*list[numpy.ndarray]*) – list of optimized pulses
- **g_a_integrals** (*numpy.ndarray*) – array of values $\int_0^T g_a(t)\,\mathrm{d}t = \int_0^T \frac{\lambda_a}{S(t)} \left\vert\Delta\epsilon(t)\right\vert^2 \mathrm{d}t$, for each pulse $\epsilon(t)$. The pulse updates $\Delta\epsilon(t)$ are the differences of the *optimized_pulses* and the *guess_pulses* (zero in the zeroth iteration that only performs a forward-propagation of the guess pulses). The quantity $\int g_a(t)\,\mathrm{d}t$ is a very useful measure of how much the pulse amplitudes changes in each iteration. This tells us whether we've chosen good values for λ_a. Values that are too small cause "pulse explosions" which immediately show up in $\int_0^T g_a(t)\,\mathrm{d}t$. Also, whether $\int g_a(t)\,\mathrm{d}t$ is increasing or decreasing between iterations gives an indication whether the optimization is "speeding up" or "slowing down", and thus whether convergence is reached (negligible pulse updates).
- **lambda_vals** (*numpy.ndarray*) – for each pulse, the value of the λ_a parameter
- **shape_arrays** (*list[numpy.ndarray]*) – for each pulse, the array of update-shape values $S(t)$
- **fw_states_T** (*list*) – for each objective, the forward-propagated state
- **tlist** (*numpy.ndarray*) – array of time grid values on which the states are defined

- **tau_vals** (*numpy.ndarray*) – for each objective, the complex overlap for the target state with the forward-propagated state, or None if no target state is defined.
- **start_time** (*float*) – The time at which the iteration started, in epoch seconds
- **stop_time** (*float*) – The time at which the iteration started, in epoch seconds
- **iteration** (*int*) – The current iteration number. For the initial propagation of the guess controls, zero.
- **shared_data** (*dict*) – Dict of data shared between any *modify_params_after_iter* and any *info_hook* functions chained together via *chain()*.
- **info_vals** (*list*) – List of the return values of the info_hook from previous iterations
- **out** – An open file handle where to write the information. This parameter is not part of the *info_hook* interface, and defaults to stdout. Use `functools.partial()` to pass a different value.

Note: This routine implements the full signature of an *info_hook* in *optimize_pulses()*, excluding *out*. However, since the *info_hook* only allows for keyword arguments, it is usually much simpler to use Python's variable keyword arguments syntax (**kwargs). For example, consider the following *info_hook* that prints (and stores) the value of the real-part gate fidelity:

```
def print_fidelity(**kwargs):
    F_re = np.average(np.array(kwargs['tau_vals']).real)
    print("    F = %f" % F_re)
    return F_re
```

---

krotov.info_hooks.**print_table**(*J_T*, *show_g_a_int_per_pulse=False*, *J_T_prev=None*, *unicode=True*, *out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Print a tabular overview of the functional values in the iteration

An example output is:

```
iter.        J_T    ∫gₐ(t)dt          J       ΔJ_T         ΔJ  secs
   0   1.00e+00    0.00e+00   1.00e+00        n/a        n/a     0
   1   7.65e-01    2.33e-02   7.88e-01  -2.35e-01  -2.12e-01     1
   2   5.56e-01    2.07e-02   5.77e-01  -2.09e-01  -1.88e-01     1
```

The table has the following columns:

- iteration number
- value of the final-time functional J_T
- If *show_g_a_int_per_pulse* is True and there is more than one control pulse: one column for each pulse containing the value of $\int_0^T \frac{\lambda_{a,i}}{S_i(t)} |\Delta\epsilon_i(t)|^2 \, \mathrm{d}t$

- The value of $\sum_i \int_0^T g_a(\epsilon_i(t))\,\mathrm{d}t = \sum_i \int_0^T \frac{\lambda_{a,i}}{S_i(t)} |\Delta\epsilon_i(t)|^2\,\mathrm{d}t$, or just $\int_0^T \frac{\lambda_a}{S(t)} |\Delta\epsilon(t)|^2\,\mathrm{d}t$ if there is only a single control pulse (as in the above example output). This value (respectively the individual values with *show_g_a_int_per_pulse*) should always be at least three orders of magnitude smaller than the pulse fluence $\sum_i \int_0^T |\epsilon_i(t)|^2\,\mathrm{d}t$. Larger changes in the pulse amplitude may be a sign of a "pulse explosion" due to values for $\lambda_{a,i}$ that are too small. Changes in $\sum_i \int_0^T \frac{\lambda_{a,i}}{S_i(t)}$ are often a better indicator of whether the optimization is "speeding up"/"slowing down"/reaching convergence than the values of J_T.
- The value of the total functional $J = J_T + \sum_i \int_0^T g_a(\epsilon_i(t))\,\mathrm{d}t$
- The change ΔJ_T in the final time functional compared to the previous iteration. This should be a negative value, indicating monotonic convergence in a minimization of J_T.
- The change ΔJ in the total functional compared to the previous iteration. This is evaluated as $\Delta J = \Delta J_T + \sum_i \int_0^T g_a(\epsilon_i(t))\,\mathrm{d}t$. Somewhat counter-intuitively, ΔJ does not contain a contribution from the $g_a(t)$ of the previous iteration. This is because the $\Delta\epsilon_i(T)$ on which $g_a(t)$ depends must be evaluated with respect to the same reference field (the guess pulse of the *current* iteration), to that $\Delta\epsilon_i(T) = 0$ when evaluated with the optimized pulse of the previous iteration (i.e., the same guess pulse of the current iteration).
- The number of seconds in wallclock time spent on the iteration
- An indicator * or ** if there is a loss of monotonic convergence in ΔJ_T and/or ΔJ. Krotov's method mathematically guarantees a negative ΔJ in the continuous limit. Assuming there are no errors in the time propagation, or in the *chi_constructor* passed to *optimize_pulses()*, a loss of monotonic convergence is due to the λ_a associated with the pulses (via *pulse_options* in *optimize_pulses()*) being too small. In practice, we usually don't care too much about a loss of monotonic convergence in ΔJ, but a loss of convergence in ΔJ_T is a serious sign of trouble. It is often associated with sharp discontinuous spikes in the optimized pulses, or a dramatic increase in the pulse amplitude.

  Parameters

  - **J_T** (*callable*) – A function that extract the value of the final time functional from the keyword-arguments passed to the *info_hook*.
  - **show_g_a_int_per_pulse** (*callable*) – If True, print a column with the value of $\int_0^T g_a(\epsilon_i(t))\,\mathrm{d}t = \int_0^T \frac{\lambda_{a,i}}{S_i(t)} |\Delta\epsilon_i(t)|^2\,\mathrm{d}t$ for every pulse $\epsilon_i(t)$. Otherwise, only print the sum over those integrals for all pulses.
  - **J_T_prev** (*None or callable*) – A function that extract the value of the final time functional *from the previous iteration*. If None, use the last values from the *info_vals* passed to the *info_hook*.
  - **unicode** (*bool*) – Whether to use unicode symbols for the column headers. Some systems have broken monospace fonts in the Jupyter notebook that cause the headers not to line up as intended.
  - **out** – An open file handle where to write the table. Defaults to stdout.

12.1.4 krotov.mu module

Routines for *mu* in *krotov.optimize.optimize_pulses()*

The first-order Krotov update equation is usually written as

$$\Delta\epsilon(t) \propto \mathrm{Im}\left\langle \chi_k^{(i)}(t) \middle| \left(\left.\frac{\partial \hat{H}}{\partial \epsilon}\right|_{\substack{\phi^{(i+1)}(t) \\ \epsilon^{(i+1)}(t)}} \right) \middle| \phi_k^{(i+1)}(t) \right\rangle ,$$

where $|\chi_k\rangle$ are states backward-propagated from a boundary condition determined by the functional, $|\phi_k\rangle$ are forward-propagated from the initial states, and $\frac{\partial \hat{H}}{\partial \epsilon}$ is the derivative of the Hamiltonian with respect to the field. However, this is true only for Hilbert-space states evolving under a Schrödinger equation.

More generally (e.g. when the states χ_k and ϕ_k are density matrices and the equation of motion is the master equation in Lindblad form), the correct formulation is

$$\frac{\partial \hat{H}}{\partial \epsilon} \rightarrow \mu = \frac{\partial H}{\partial \epsilon} ,$$

where H is now the abstract operator appearing in the equation of motion of the abstract state

$$\dot{\phi}_k(t) = -iH\phi_k(t)$$

For density matrices, we have

$$\frac{\partial}{\partial t}\hat{\rho}_k(t) = \mathcal{L}\hat{\rho}_k(t)$$

and thus $H = i\mathcal{L}$.

To allow for arbitrary equations of motion, a routine *mu* may be passed to *optimize_pulses()* that returns the abstract operator μ as a Qobj, or alternatively as a callable that takes ϕ_k as its argument and evaluates $\mu\phi_k$. The default *mu* is *derivative_wrt_pulse()*, which covers the most common equation of motions:

- standard Schrödinger equation
- master equation, where either the *H* attribute of the objective contains a Hamiltonian and there are Lindblad operators in *c_ops*, or the *H* attribute contains a super-operator $\mathcal{L}$ directly (the case discussed above).

Alternative implementations of *mu* must have the same signature as *derivative_wrt_pulse()*, but should only be required in rare circumstances, such as when the derivative still depends on the control values or on the states. (Or, if you can provide a more efficient problem-specific implementation).

Summary

Functions:

derivative_wrt_pulse	Calculate ∂H/∂ϵ for the standard equations of motion.

__all__: *derivative_wrt_pulse*

Reference

krotov.mu.**derivative_wrt_pulse**(*objectives*, *i_objective*, *pulses*, *pulses_mapping*, *i_pulse*, *time_index*)

Calculate ∂H/∂ϵ for the standard equations of motion.

Parameters

- **objectives** (*list*) – List of *Objective* instances
- **i_objective** (*int*) – The index of the objective in *objectives* whose equation of motion the derivative should be calculated.
- **pulses** (*list*) – The list of pulses occuring in *objectives*
- **pulses_mapping** (*list*) – The mapping of elements of *pulses* to the components of *objectives*, as returned by *extract_controls_mapping()*
- **i_pulse** (*int*) – The index of the pulse in *pulses* for which to calculate the derivative
- **time_index** (*int*) – The index of the value in `pulses[i_pulse]` that should be plugged in to ∂H/∂ϵ. Not used, as this routine only considers equations of motion that are linear in the controls.

Returns The quantum operator or super-operator that represents ∂H/∂ϵ. In general, the return type can be any callable *mu* so that `mu(state)` calculates the result of applying ∂H/∂ϵ to *state*. In most cases, a `Qobj` will be returned, which is just the most convenient example of an appropriate callable.

Return type callable

This function covers the following cases:

- the *H* attribute of the objective contains a Hamiltonian, there are no *c_ops* (Schrödinger equation: the abstract H in ∂H/∂ϵ is the Hamiltonian directly)
- the *H* attribute of the objective contains a Hamiltonian $\hat{H}$, and there are Lindblad operators $\hat{L}_i$ in *c_ops* (master equation in Lindblad form). The abstract H is $i\mathcal{L}$ for the Liouvillian defined as

$$\mathcal{L}[\hat{\rho}] = -i[\hat{H}, \hat{\rho}] + \sum_i \left(\hat{L}_i \hat{\rho} \hat{L}_i^\dagger - \frac{1}{2} \left\{ \hat{L}_i^\dagger \hat{L}_i, \hat{\rho} \right\} \right)$$

- the *H* attribute of the objective contains a super-operator $\mathcal{L}$, there are no *c_ops* (general master equation). The abstract H is again $i\mathcal{L}$.

12.1.5 krotov.objectives module

Routines for formulating objectives.

Objectives, represented as an *Objective* instance, describe the *physical* objective of an optimization, e.g. a state-to-state transformation, or a quantum gate. This is distinct from the *mathematical* formulation of an optimization functional (*krotov.functionals*). For the same physical objective, there are usually several different functionals whose minimization achieve that objective.

Summary

Classes:

Objective	A single objective for optimization with Krotov's method.

Functions:

ensemble_objectives	Extend *objectives* for an "ensemble optimization"
gate_objectives	Construct a list of objectives for optimizing towards a quantum gate
liouvillian	Convert Hamiltonian and Lindblad operators into a Liouvillian.

`__all__`: *Objective*, *ensemble_objectives*, *gate_objectives*, *liouvillian*

Reference

`krotov.objectives.`**`FIX_QUTIP_932 = True`**

Workaround for QuTiP issue 932.

If True, in *Objective.mesolve()*, replace any array controls with an equivalent function. This results in a signficant slowdown of the propagation, as it circumvents the use of Cython. Defaults to False on Linux, and True on any non-Linux system.

class `krotov.objectives.`**`Objective`**(*, *initial_state*, *H*, *target*, *c_ops=None*)

Bases: `object`

A single objective for optimization with Krotov's method.

Parameters

- **initial_state** (*qutip.Qobj*) – value for *initial_state*
- **H** (*qutip.Qobj or list*) – value for *H*
- **target** (*qutip.Qobj or None*) – value for *target*
- **c_ops** (*list or None*) – value for *c_ops*

Example

```
>>> H0 = - 0.5 * qutip.operators.sigmaz()
>>> H1 = qutip.operators.sigmax()
>>> eps = lambda t, args: ampl0
>>> H = [H0, [H1, eps]]
>>> krotov.Objective(
...     initial_state=qutip.ket('0'), target=qutip.ket('1'), H=H
... )
Objective[|Ψ₀(2)⟩ to |Ψ₁(2)⟩ via [H₀[2,2], [H₁[2,2], u₁(t)]]]
```

Raises `ValueError` – If any arguments have an invalid type or structure. This can be surpressed by setting the *type_checking* class attribute to False.

Note: Giving collapse operators via *c_ops* only makes sense if the *propagator* passed to *optimize_pulses()* takes them into account explicitly. It is strongly recommended to set *H* as a Lindblad operator instead, see *liouvillian()*.

H
The (time-dependent) Hamiltonian or Liouvillian in nested-list format, cf. `qutip.mesolve.mesolve()`. This includes the control fields.

Type qutip.Qobj or list

initial_state
The initial state, as a Hilbert space state, or a density matrix.

Type qutip.Qobj

target
An object describing the "target" of the optimization, for the dynamics starting from *initial_state*. Usually, this will be the target state (the state into which *initial_state* should evolve). More generally, it can be an arbitrary object meeting the conventions of a specific *chi_constructor* function that will be passed to *optimize_pulses()*.

c_ops
List of collapse operators, cf. `mesolve()`, in lieu of *H* being a Liouvillian.

Type list or None

str_use_unicode = True
Whether the string representation of an *Objective* may use unicode symbols, cf. *summarize()* (class attribute).

type_checking = True
By default, instantiating *Objective* with invalid types raises a `ValueError`. Setting this to False disables type checks in the initializer, allowing certain advanced use cases such as using plain numpy objects instead of QuTiP objects (class attribute).

adjoint()
The *Objective* containing the adjoint of all components.

This does not affect the controls in *H*: these are assumed to be real-valued. Also, *Objective.target* will be left unchanged if its adjoint cannot be calculated (if it is not a target state).

mesolve(*tlist*, *rho0=None*, *H=None*, *c_ops=None*, *e_ops=None*, ***kwargs*)
Run `qutip.mesolve.mesolve()` on the system of the objective.

Solve the dynamics for the *H* and *c_ops* of the objective, starting from the objective's *initial_state*, by delegating to `qutip.mesolve.mesolve()`. Both the initial state and the dynamical generator for the propagation can be overridden by passing *rho0* and *H*/*c_ops*.

Parameters

- **tlist** (*numpy.ndarray*) – array of time grid points on which the states are defined

- **rho0** (*qutip.Qobj or None*) – The initial state for the propagation. If None, the *initial_state* attribute is used.
- **H** (*qutip.Qobj or None*) – The dynamical generator (Hamiltonian or Liouvillian) for the propagation. If None, the *H* attribute is used.
- **c_ops** (*list or None*) – List of collapse (Lindblad) operators. If None, the *c_ops* attribute is used.
- **e_ops** (*list or None*) – A list of operators whose expectation values to calculate, for every point in *tlist*. See `qutip.mesolve.mesolve()`.
- ****kwargs** – All further arguments will be passed to `qutip.mesolve.mesolve()`.

Returns Result of the propagation, see `qutip.mesolve.mesolve()` for details.

Return type qutip.solver.Result

propagate(*tlist*, *, *propagator*, *rho0=None*, *H=None*, *c_ops=None*, *e_ops=None*, *expect=<function expect>*)

Propagate the system of the objective over the entire time grid.

Solve the dynamics for the *H* and *c_ops* of the objective. If *rho0* is not given, the *initial_state* will be propagated. This is similar to the *mesolve()* method, but instead of using `qutip.mesolve.mesolve()`, the *propagate* function is used to go between points on the time grid. This function is the same as what is passed to *optimize_pulses()*. The crucial difference between this and *mesolve()* is in the time discretization convention. While *mesolve()* uses piecewise-constant controls centered around the values in *tlist* (the control field switches in the middle between two points in *tlist*), *propagate()* uses piecewise-constant controls on the intervals of *tlist* (the control field switches on the points in *tlist*). The function *expect* is used to calculate expecation values; it receives two parameters, an operator from *e_ops* and a state, and must returnd the expectation value of the operator.

Comparing the result of *mesolve()* and *propagate()* allows to estimate the "time discretization error". If the error is significant, a shorter time step shoud be used.

Returns Result of the propagation, using the same structure as *mesolve()*.

Return type qutip.solver.Result

classmethod reset_symbol_counters()

Reset the internal symbol counters used for printing objectives.

See *summarize()*.

summarize(*use_unicode=True*, *reset_symbol_counters=False*)

Return a one-line summary of the objective as a string.

Parameters

- **use_unicode** (*bool*) – If False, only use ascii symbols in the output
- **reset_symbol_counters** (*bool*) – If True, reset the internal object counters (see *reset_symbol_counters()*) before calculating the result

The *summarize()* method (which is also used for the `repr()` and `__str__` of an *Objective*) keeps per-process internal counters for the various categories of objects that may occur as attributes of an *Objective* (kets, bras, Hermitian operators, non-Hermitian Operators, density matrices, Liouvillians, Lindblad operators, numpy

arrays, control functions). This allows to keep track of objects across multiple objectives. The counters can be reset with *reset_symbol_counters()*.

The ouput uses various unicode symbols (or ascii-equivalents, if *use_unicode* is False):

- 'Ψ' ('Psi') for qutip.Qobj quantum states (kets or bras)
- 'ρ' ('rho') for qutip.Qobj operators that occur as initial or target states (density matrices)
- 'L' for Lindblad operators (elements of *c_ops*)
- 'H' for Hermitian qutip.Qobj operators (Hamiltonians)
- 'A' for non-Hermitian qutip.Qobj operators in *H*
- '' ('Lv') for qutip.Qobj super-operators (Liouvillians)
- 'a' for numpy arrays (of any dimension)
- 'u' for (callable) control functions.

Example

```
>>> from qutip import ket, tensor, sigmaz, sigmax, sigmap, identity
>>> u1 = lambda t, args: 1.0
>>> u2 = lambda t, args: 1.0
>>> a1 = np.random.random(100) + 1j*np.random.random(100)
>>> a2 = np.random.random(100) + 1j*np.random.random(100)
>>> H = [
...     tensor(sigmaz(), identity(2)) +
...     tensor(identity(2), sigmaz()),
...     [tensor(sigmax(), identity(2)), u1],
...     [tensor(identity(2), sigmax()), u2]]
>>> C1 = [[tensor(identity(2), sigmap()), a1]]
>>> C2 = [[tensor(sigmap(), identity(2)), a2]]
>>> ket00 = ket((0,0))
>>> ket11 = ket((1,1))
>>> obj = Objective(
...     initial_state=ket00,
...     target=ket11,
...     H=H
... )
>>> obj.reset_symbol_counters()
>>> obj.summarize()
'|Ψ₀(2⊗2)⟩ to |Ψ₁(2⊗2)⟩ via [H₀[2⊗2,2⊗2], [H₁[2⊗2,2⊗2], u₁(t)], [H₂[2⊗2,2⊗2],␣
↪u₂(t)]]'
>>> obj = Objective(
...     initial_state=ket00,
...     target=ket11,
...     H=H,
...     c_ops=[C1, C2]
... )
>>> obj.summarize()
'|Ψ₀(2⊗2)⟩ to |Ψ₁(2⊗2)⟩ via {H:[H₀[2⊗2,2⊗2], [H₁[2⊗2,2⊗2], u₁(t)], [H₂[2⊗2,
↪2⊗2], u₂(t)]], c_ops:([[L₀[2⊗2,2⊗2], a₀[100]]],[[L₁[2⊗2,2⊗2], a₁[100]]])}'
>>> obj.summarize(use_unicode=False)
'|Psi0(2*2)> to |Psi1(2*2)> via {H:[H0[2*2,2*2], [H1[2*2,2*2], u1(t)],␣
↪[H2[2*2,2*2], u2(t)]], c_ops:([[L0[2*2,2*2], a0[100]]],[[L1[2*2,2*2],␣
↪a1[100]]])}'
```

(continues on next page)

(continued from previous page)

```
>>> copy.deepcopy(obj).summarize()  # different objects!
'|Ψ₂(2⊗2)⟩ to |Ψ₃(2⊗2)⟩ via {H:[H₃[2⊗2,2⊗2], [H₄[2⊗2,2⊗2], u₁(t)], [H₅[2⊗2,
↪2⊗2], u₂(t)]], c_ops:([[L₂[2⊗2,2⊗2], a₂[100]]],[[L₃[2⊗2,2⊗2], a₃[100]]])}'
>>> copy.deepcopy(obj).summarize(reset_symbol_counters=True)
'|Ψ₀(2⊗2)⟩ to |Ψ₁(2⊗2)⟩ via {H:[H₀[2⊗2,2⊗2], [H₁[2⊗2,2⊗2], u₁(t)], [H₂[2⊗2,
↪2⊗2], u₂(t)]], c_ops:([[L₀[2⊗2,2⊗2], a₀[100]]],[[L₁[2⊗2,2⊗2], a₁[100]]])}'
```

krotov.objectives.**gate_objectives**(*basis_states*, *gate*, *H*, *c_ops=None*, *local_invariants=False*, *liouville_states_set=None*, *weights=None*, *normalize_weights=True*)

Construct a list of objectives for optimizing towards a quantum gate

Parameters

- **basis_states** (*list[qutip.Qobj]*) – A list of n canonical basis states
- **gate** – The gate to optimize for, as a $n \times n$ matrix-like object (must have a *shape* attribute, and be indexable by two indices). Alternatively, *gate* may be the string 'perfect_entangler' or 'PE', to indicate the optimization for an arbitrary two-qubit perfect entangler.
- **H** (*list or qutip.Qobj*) – The Hamiltonian (or Liouvillian) for the time evolution, in nested-list format.
- **c_ops** (*list or None*) – A list of collapse (Lindblad) operators, or None for unitary dynamics or if *H* is a Liouvillian (preferred!)
- **local_invariants** (*bool*) – If True, initialize the objectives for an optimization towards a two-qubit gate that is "locally equivalent" to *gate*. That is, the result of the optimization should implement *gate* up to single-qubit operations.
- **liouville_states_set** (*None or str*) – If not None, one of "full", "3states", "d+1". This sets the objectives for a gate optimization in Liouville space, using the states defined in Goerz et al. New J. Phys. 16, 055012 (2014). See Examples for details.
- **weights** (*None or list*) – If given as a list, weights for the different objectives. These will be added as a custom attribute to the respective *Objective*, and may be used by a particular functional (*chi_constructor*). The intended use case is for the *liouville_states_set* values '3states', and 'd+1', where the different objectives have clear physical interpretations that might be given differing importance. A weight of 0 will completely drop the corresponding objective.
- **normalize_weights** (*bool*) – If True, and if *weights* is given as a list of values, normalize the weights so that they sum to N, the number of objectives. IF False, the weights will be used unchanged.

Returns

The objectives that define the optimization towards the gate. For a "normal" gate with a basis in Hilbert space, the objectives will have the *basis_states* as each *initial_state* and the result of applying *gate* to the *basis_states* as each *target*.

For an optimization towards a perfect-entangler, or for the *local_invariants* of the given *gate*, each *initial_state* will be the Bell basis state described in "Theorem 1" in Y. Makhlin, Quantum Inf. Process. 1, 243 (2002), derived

from the canonical *basis_states*. The *target* will be the string 'PE' for a perfect-entanglers optimization, and *gate* for the local-invariants optimization.

Return type list[*Objective*]

Raises **ValueError** – If *gate*, *basis_states*, and *local_invariants* are incompatible, or *gate* is invalid (not a recognized string)

Note: The dimension of the *basis_states* is not required to be the dimension of the *gate*; the *basis_states* may define a logical subspace in a larger Hilbert space.

Examples

- A single-qubit gate:

```
>>> from qutip import ket, bra, tensor
>>> from qutip import sigmaz, sigmax, sigmay, sigmam, identity
>>> basis = [ket([0]), ket([1])]
>>> gate = sigmay()  # = -i|0⟩⟨1| + i|1⟩⟨0|
>>> H = [sigmaz(),[sigmax(), lambda t, args: 1.0]]
>>> objectives = gate_objectives(basis, gate, H)
>>> assert objectives == [
...     Objective(
...         initial_state=basis[0],
...         target=(1j * basis[1]),
...         H=H
...     ),
...     Objective(
...         initial_state=basis[1],
...         target=(-1j * basis[0]),
...         H=H
...     )
... ]
```

- An arbitrary two-qubit perfect entangler:

```
>>> basis = [ket(n) for n in [(0, 0), (0, 1), (1, 0), (1, 1)]]
>>> H = [
...     tensor(sigmaz(), identity(2)) +
...     tensor(identity(2), sigmaz()),
...     [tensor(sigmax(), identity(2)), lambda t, args: 1.0],
...     [tensor(identity(2), sigmax()), lambda t, args: 1.0]]
>>> objectives = gate_objectives(basis, 'PE', H)
>>> from weylchamber import bell_basis
>>> for i in range(4):
...     assert objectives[i] == Objective(
...        initial_state=bell_basis(basis)[i],
...        target='PE',
...        H=H
...     )
```

- A two-qubit gate, up to single-qubit operation ("local invariants"):

```
>>> objectives = gate_objectives(
...     basis, qutip.gates.cnot(), H, local_invariants=True
... )
>>> for i in range(4):
...     assert objectives[i] == Objective(
...        initial_state=bell_basis(basis)[i],
...        target=qutip.gates.cnot(),
...        H=H
...     )
```

- A two-qubit gate in a dissipative system tracked by 3 density matrices:

```
>>> L = krotov.objectives.liouvillian(H, c_ops=[
...     tensor(sigmam(), identity(2)),
...     tensor(identity(2), sigmam())])
>>> objectives = gate_objectives(
...     basis, qutip.gates.cnot(), L,
...     liouville_states_set='3states',
...     weights=[20, 1, 1]
... )
```

The three states, for a system with a logical subspace of dimension d with a basis $\{|i\rangle\}$, $i \in [1, d]$ are:

$$\hat{\rho}_1 = \sum_{i=1}^{d} \frac{2(d-i+1)}{d(d+1)} |i\rangle\langle i|$$

$$\hat{\rho}_2 = \sum_{i,j=1}^{d} \frac{1}{d} |i\rangle\langle j|$$

$$\hat{\rho}_3 = \sum_{i=1}^{d} \frac{1}{d} |i\rangle\langle i|$$

The explicit form of the three states in this example is:

```
>>> assert np.allclose(objectives[0].initial_state.full(),
...     np.diag([0.4, 0.3, 0.2, 0.1]))

>>> assert np.allclose(objectives[1].initial_state.full(),
...     np.full((4, 4), 1/4))

>>> assert np.allclose(objectives[2].initial_state.full(),
...     np.diag([1/4, 1/4, 1/4, 1/4]))
```

The objectives in this example are weighted (20/1/1):

```
>>> "%.5f" % objectives[0].weight
'2.72727'
>>> "%.5f" % objectives[1].weight
'0.13636'
>>> "%.5f" % objectives[2].weight
'0.13636'
>>> sum_of_weights = sum([obj.weight for obj in objectives])
>>> "%.1f" % sum_of_weights
'3.0'
```

- A two-qubit gate in a dissipative system tracked by $d + 1 = 5$ pure-state density matrices:

```
>>> objectives = gate_objectives(
...     basis, qutip.gates.cnot(), L,
...     liouville_states_set='d+1'
... )
```

The first four *initial_states* are the pure states corresponding to the Hilbert space basis

```
>>> assert objectives[0].initial_state == qutip.ket2dm(ket('00'))
>>> assert objectives[1].initial_state == qutip.ket2dm(ket('01'))
>>> assert objectives[2].initial_state == qutip.ket2dm(ket('10'))
>>> assert objectives[3].initial_state == qutip.ket2dm(ket('11'))
```

The fifth state is $\hat{\rho}_2$ from '3states':

```
>>> assert np.allclose(objectives[4].initial_state.full(),
...     np.full((4, 4), 1/4))
```

- A two-qubit gate in a dissipative system tracked by the full Liouville space basis:

```
>>> objectives = gate_objectives(
...     basis, qutip.gates.cnot(), L,
...     liouville_states_set='full'
... )
```

The Liouville space basis states are all the possible dyadic products of the Hilbert space basis:

```
>>> assert objectives[0].initial_state == ket('00') * bra('00')
>>> assert objectives[1].initial_state == ket('00') * bra('01')
>>> assert objectives[2].initial_state == ket('00') * bra('10')
>>> assert objectives[3].initial_state == ket('00') * bra('11')
>>> assert objectives[4].initial_state == ket('01') * bra('00')
>>> assert objectives[5].initial_state == ket('01') * bra('01')
>>> assert objectives[6].initial_state == ket('01') * bra('10')
>>> assert objectives[7].initial_state == ket('01') * bra('11')
>>> assert objectives[8].initial_state == ket('10') * bra('00')
>>> assert objectives[9].initial_state == ket('10') * bra('01')
>>> assert objectives[10].initial_state == ket('10') * bra('10')
>>> assert objectives[11].initial_state == ket('10') * bra('11')
>>> assert objectives[12].initial_state == ket('11') * bra('00')
>>> assert objectives[13].initial_state == ket('11') * bra('01')
>>> assert objectives[14].initial_state == ket('11') * bra('10')
>>> assert objectives[15].initial_state == ket('11') * bra('11')
```

krotov.objectives.**ensemble_objectives**(*objectives*, *Hs*)

Extend *objectives* for an "ensemble optimization"

This creates a list of objectives for an optimization for robustness with respect to variations in some parameter of the Hamiltonian. The trick is to simply optimize over the average of multiple copies of the system (the *Hs*) sampling that variation. See Goerz, Halperin, Aytac, Koch, Whaley. Phys. Rev. A 90, 032329 (2014) for details.

Parameters

- **objectives** (*list[Objective]*) – The n original objectives
- **Hs** (*list*) – List of m variations of the original Hamiltonian/Liouvillian

Returns List of $n(m+1)$ new objectives that consists of the original objectives, plus one copy of the original objectives per element of *Hs* where the *H* attribute of each objectives is replaced by that element.

Return type list[*Objective*]

krotov.objectives.**liouvillian**(*H*, *c_ops*)

Convert Hamiltonian and Lindblad operators into a Liouvillian.

This is like `qutip.superoperator.liouvillian()`, but *H* may be a time-dependent Hamiltonian in nested-list format. *H* is assumed to contain a drift Hamiltonian, and the Lindblad operators in *c_ops* cannot be time-dependent.

12.1.6 krotov.optimize module

Summary

Functions:

optimize_pulses	Use Krotov's method to optimize towards the given *objectives*.

`__all__`: *optimize_pulses*

Reference

krotov.optimize.**optimize_pulses**(*objectives*, *pulse_options*, *tlist*, ***, *propagator*, *chi_constructor*, *mu=None*, *sigma=None*, *iter_start=0*, *iter_stop=5000*, *check_convergence=None*, *info_hook=None*, *modify_params_after_iter=None*, *storage='array'*, *parallel_map=None*, *store_all_pulses=False*, *continue_from=None*, *skip_initial_forward_propagation=False*, *norm=None*, *overlap=None*)

Use Krotov's method to optimize towards the given *objectives*.

Optimize all time-dependent controls found in the Hamiltonians or Liouvillians of the given *objectives*.

Parameters

- **objectives** (*list[Objective]*) – List of objectives
- **pulse_options** (*dict*) – Mapping of time-dependent controls found in the Hamiltonians of the objectives to a dictionary of options for that control. There must be options given for *every* control. As numpy arrays are unhashable and thus cannot be used as dict keys, the options for a control that is an array must be set using the key `id(control)` (see the example below). The options of any particular control must contain the following keys:
  - `'lambda_a'`: the Krotov step size (float value). This governs the overall magnitude of the pulse update. Large values result in small updates. Small values may lead to sharp spikes and numerical instability.

- `'update_shape'` : Function S(t) in the range [0, 1] that scales the pulse update for the pulse value at t. This can be used to ensure boundary conditions (S(0) = S(T) = 0), and enforce smooth switch-on and switch-off. This can be a callable that takes a single argument *t*; or the values 1 or 0 for a constant update-shape. The value 0 disables the optimization of that particular control.

For example, for *objectives* that contain a Hamiltonian of the form `[H0, [H1, u], [H2, g]]`, where `H0`, `H1`, and `H2` are `Qobj` instances, `u` is a numpy array of control values, and `g` is a control function (a callable), a possible value for *pulse_options* would look like this:

```
>>> from krotov.shapes import flattop
>>> from functools import partial
>>> u = numpy.zeros(1000)
>>> g = lambda t, args: 0.0
>>> pulse_options = {
...     id(u): {'lambda_a': 1.0, 'update_shape': 1},
...     g: dict(
...         lambda_a=1.0,
...         shape=partial(
...             flattop, t_start=0, t_stop=10, t_rise=1.5
...         )
...     )
... }
```

- **tlist** (*numpy.ndarray*) – Array of time grid values, cf. `mesolve()`
- **propagator** (*callable or list[callable]*) – Function that propagates the state backward or forwards in time by a single time step, between two points in *tlist*. Alternatively, a list of functions, one for each objective. If the propagator is stateful, it should be an instance of *krotov.propagators.Propagator*. See *krotov.propagators* for details.
- **chi_constructor** (*callable*) – Function that calculates the boundary condition for the backward propagation. This is where the final-time functional (indirectly) enters the optimization. See *krotov.functionals* for details.
- **mu** (*None or callable*) – Function that calculates the derivative $\frac{\partial H}{\partial \epsilon}$ for an equation of motion $\dot{\phi}(t) = -iH[\phi(t)]$ of an abstract operator H and an abstract state ϕ. If None, defaults to *krotov.mu.derivative_wrt_pulse()*, which covers the standard Schrödinger and master equations. See *krotov.mu* for a full explanation of the role of *mu* in the optimization, and the required function signature.
- **sigma** (*None or krotov.second_order.Sigma*) – Function (instance of a *Sigma* subclass) that calculates the second-order contribution. If None, the first-order Krotov method is used.
- **iter_start** (*int*) – The formal iteration number at which to start the optimization
- **iter_stop** (*int*) – The iteration number after which to end the optimization, whether or not convergence has been reached
- **check_convergence** (*None or callable*) – Function that determines whether the optimization has converged. If None, the optimization will only end when *iter_stop* is reached. See *krotov.convergence* for details.

- **info_hook** (*None or callable*) – Function that is called after each iteration of the optimization, for the purpose of analysis. Any value returned by *info_hook* (e.g. an evaluated functional J_T) will be stored, for each iteration, in the *info_vals* attribute of the returned *Result*. The *info_hook* must have the same signature as *krotov.info_hooks.print_debug_information()*. It should not modify its arguments in any way, except for *shared_data*.
- **modify_params_after_iter** (*None or callable*) – Function that is called after each iteration, which may modify its arguments for certain advanced use cases, such as dynamically adjusting *lambda_vals*, or applying spectral filters to the *optimized_pulses*. It has the same interface as *info_hook* but should not return anything. The *modify_params_after_iter* function is called immediately before *info_hook*, and can transfer arbitrary data to any subsequent *info_hook* via the *shared_data* argument.
- **storage** (*callable*) – Storage constructor for the storage of propagated states. Must accept an integer parameter N and return an empty array-like container of length N. The default value 'array' is equivalent to `functools.partial(numpy.empty, dtype=object)`.
- **parallel_map** (*callable or tuple or None*) – Parallel function evaluator. If given as a callable, the argument must have the same specification as `qutip.parallel.serial_map()`. A value of None is the same as passing `qutip.parallel.serial_map()`. If given as a tuple, that tuple must contain three callables, each of which has the same specification as `qutip.parallel.serial_map()`. These three callables are used to parallelize (1) the initial forward-propagation, (2) the backward-propagation under the guess pulses, and (3) the forward-propagation by a single time step under the optimized pulses. See *krotov.parallelization* for details.
- **store_all_pulses** (*bool*) – Whether or not to store the optimized pulses from *all* iterations in *Result*.
- **continue_from** (*None or Result*) – If given, continue an optimization from a previous *Result*. The result must have identical *objectives*.
- **skip_initial_forward_propagation** (*bool*) – If given as *True* together with *continue_from*, skip the initial forward propagation ("zeroth iteration"), and take the forward-propagated states from *Result.states* instead.
- **norm** (*callable or None*) – A single-argument function to calculate the norm of states. If None, delegate to the `norm()` method of the states.
- **overlap** (*callable or None*) – A two-argument function to calculate the complex overlap of two states. If None, delegate to `qutip.Qobj.overlap()` for Hilbert space states and to the Hilbert-Schmidt norm $\mathrm{tr}[\rho_1^\dagger \rho 2]$ for density matrices or operators.

Returns The result of the optimization.

Return type *Result*

Raises **ValueError** – If any controls are not real-valued, or if any update shape is not a real-valued function in the range [0, 1]; if using *continue_from* with

a *Result* with differing *objectives*; if there are any required keys missing in *pulse_options*.

12.1.7 krotov.parallelization module

Support routines for running the optimization in parallel across the objectives

The time-propagation that is the main numerical effort in an optimization with Krotov's method can naturally be performed in parallel for the different objectives. There are three time-propagations that happen inside *optimize_pulses()*:

1. A forward propagation of the *initial_state* of each objective under the initial guess pulse.
2. A backward propagation of the states $|\chi_k\rangle$ constructed by the *chi_constructor* routine that is passed to *optimize_pulses()*, where the number of states is the same as the number of objectives.
3. A forward propagation of the *initial_state* of each objective under the optimized pulse in each iteration. This can only be parallelized *per time step*, as the propagated states from each time step collectively determine the pulse update for the next time step, which is then used for the next propagation step. (In this sense Krotov's method is "sequential")

The *optimize_pulses()* routine has a parameter *parallel_map* that can receive a tuple of three "map" functions to enable parallelization, corresponding to the three propagation listed above. If not given, `qutip.parallel.serial_map()` is used for all three propations, running in serial. Any alternative "map" must have the same interface as `qutip.parallel.serial_map()`.

It would be natural to assume that `qutip.parallel.parallel_map()` would be a good choice for parallel execution, using multiple CPUs on the same machine. However, this function is only a good choice for the propagation (1) and (2): these run in parallel over the entire time grid without any communication, and thus minimal overhead. However, this is not true for the propagation (3), which must synchronize after each time step. In that case, the "naive" use of `qutip.parallel.parallel_map()` results in a communication overhead that completely dominates the propagation, and actually makes the optimization slower (potentially by more than an order of magnitude).

The function *parallel_map_fw_prop_step()* provided in this module is an appropriate alternative implementation that uses long-running processes, internal caching, and minimal inter-process communication to eliminate the communication overhead as much as possible. However, the internal caching is valid only under the assumption that the *propagate* function does not have side effects.

In general,

```
parallel_map=(
    qutip.parallel_map,
    qutip.parallel_map,
    krotov.parallelization.parallel_map_fw_prop_step,
)
```

is a decent choice for enabling parallelization for a typical multi-objective optimization.

You may implement your own "map" functions to exploit parallelization paradigms other than Python's built-in `multiprocessing`, provided here. This includes distributed propagation, e.g. through ipyparallel clusters. To write your own *parallel_map* functions, review the source code of *optimize_pulses()* in detail.

In most cases, it will be difficult to obtain a linear speedup from parallelization: even with carefully tuned manual interprocess communication, the communication overhead can be substantial. For best results, it would be necessary to use *parallel_map* functions implemented in Cython, where the GIL can be released and the entire propagation (and storage of propagated states) can be done in shared-memory with no overhead.

Summary

Classes:

Consumer	A process-based task consumer
FwPropStepTask	A task that performs a single forward-propagation step

Functions:

parallel_map_fw_prop_step	*parallel_map* function for the forward-propagation by one time step

`__all__`: *Consumer*, *FwPropStepTask*, *parallel_map_fw_prop_step*

Reference

class krotov.parallelization.**Consumer**(*task_queue*, *result_queue*, *data*)
Bases: `multiprocessing.context.Process`

A process-based task consumer

Parameters

- **task_queue** (*multiprocessing.JoinableQueue*) – A queue from which to read tasks.
- **result_queue** (*multiprocessing.Queue*) – A queue where to put the results of a task
- **data** – cached (in-process) data that will be passed to each task

run()
Execute all tasks on the *task_queue*.

Each task must be a callable that takes *data* as its only argument. The return value of the task will be put on the *result_queue*. A None value on the *task_queue* acts as a "poison pill", causing the *Consumer* process to shut down.

class krotov.parallelization.**FwPropStepTask**(*i_state*, *pulse_vals*, *time_index*)
Bases: `object`

A task that performs a single forward-propagation step

The task object is a callable, receiving the single tuple of the same form as *task_args* in *parallel_map_fw_prop_step()* as input. This *data* is internally cached by the *Consumer* that will execute the task.

Parameters

- **i_state** (*int*) – The index of the state to propagation. That is, the index of the objective from whose *initial_state* the propagation started
- **pulse_vals** (*list[float]*) – the values of the pulses at *time_index* to use.
- **time_index** (*int*) – the index of the interval on the time grid covered by the propagation step

The passed arguments update the internal state (*data*) of the *Consumer* executing the task; they are the minimal information that must be passed via inter-process communication to enable the forward propagation (assuming *propagate* in *optimize_pulses()* has no side-effects)

krotov.parallelization.**parallel_map_fw_prop_step**(*shared*, *values*, *task_args*)

parallel_map function for the forward-propagation by one time step

Parameters

- **shared** – A global object to which we can attach attributes for sharing data between different calls to *parallel_map_fw_prop_step()*, allowing us to have long-running *Consumer* processes, avoiding process-management overhead. This happens to be a callable (the original internal routine for performing a forward-propagation), but here, it is (ab-)used as a storage object only.
- **values** (*list*) – a list 0..(N-1) where N is the number of objectives
- **task_args** (*tuple*) – A tuple of 7 components:

1. A list of states to propagate, one for each objective.
2. The list of objectives
3. The list of optimized pulses (updated up to *time_index*)
4. The "pulses mapping", cf *extract_controls_mapping()*
5. The list of time grid points
6. The index of the interval on the time grid over which to propagate
7. A list of *propagate* callables, as passed to *optimize_pulses()*. The propagators must not have side-effects in order for *parallel_map_fw_prop_step()* to work correctly.

12.1.8 krotov.propagators module

Routines that can be passed as *propagator* to *optimize_pulses()*

The numerical effort involved in the optimization is almost entirely within the simulation of the system dynamics. In every iteration and for every objective, the system must be "propagated" once forwards in time and once backwards in time, see also *krotov.parallelization*.

The implementation of this time propagation must be inside the user-supplied routine *propagator* that is passed to *optimize_pulses()* and must calculate the propagation over a single time step. In particular, `qutip.mesolve.mesolve()` is not automatically used for simulating any dynamics within the optimization. The signature for any *propagator* must be the same as the "reference" *expm()* propagator:

```
>>> str(inspect.signature(krotov.propagators.expm))
'(H, state, dt, c_ops=None, backwards=False, initialize=False)'
```

The arguments are as follows (cf. *Propagator*):

- *H* is the system Hamiltonian or Liouvillian, in a nested-list format similar to that used by `qutip.mesolve.mesolve()`, e.g., for a Hamiltonian $\hat{H} = \hat{H}_0 + c\hat{H}_1$, where c is the value of a control field at a particular point in time, *propagator* would receive a list `[H0, [H1, c]]` where `H0` and `H1` are `qutip.Qobj` operators. The nested-list for *H* used here, with scalar values for the controls, is obtained internally from the format used by `mesolve()`, with time-dependent controls over the entire time grid, via *krotov.structural_conversions.plug_in_pulse_values()*.
- *state* is the `qutip.Qobj` state that should be propagated, either a Hilbert space state, or a density matrix.
- *dt* is the time step (a float). It is always positive, even for `backwards=True`.
- *c_ops* is None, or a list of collapse (Lindblad) operators, where each list element is a `qutip.Qobj` instance (or possibly a nested list, for time-dependent Lindblad operators. Note that is generally preferred for *H* to be a Liouvillian, for dissipative dynamics.
- *backwards* (`bool`): If passed as *True*, the *propagator* should propagate backwards in time. In Hilbert space, this means using *-dt* instead of *dt*. In Liouville space, there is no difference between forward and backward propagation. In the context of Krotov's method, the backward propagation uses the conjugate Hamiltonian, respectively Liouvillian. However, the *propagator* routine does not need to be aware of this fact: it will receive the appropriate *H* and *c_ops*.
- *initialize* (`bool`): A flag to indicate the beginning of a propagation over a time grid. If False in subsequent calls, the *propagator* may assume that the input *state* is the result of the previous call to *propagator*.

The routines in this module are provided with no guarantee to be either general or efficient. The *expm()* propagator is exact to machine precision, but generally extremely slow. For "production use", it is recommended to supply a problem-specific *propagator* that is highly optimized for speed. You might consider the use of Cython. This is key to minimize the runtime of the optimization.

The *initialize* flag enables "stateful" propagators that cache data between calls. This can significantly improve numerical efficiency. *DensityMatrixODEPropagator* is an example for such a propagator. In general, any stateful *propagator* should be an instance of *Propagator*.

Summary

Classes:

DensityMatrixODEPropagator	Propagator for density matrix evolution under a Lindbladian
Propagator	Abstract base class for stateful propagators

Functions:

expm	Propagate using matrix exponentiation

__all__: *DensityMatrixODEPropagator*, *Propagator*, *expm*

Reference

krotov.propagators.**expm**(*H*, *state*, *dt*, *c_ops=None*, *backwards=False*, *initialize=False*)

Propagate using matrix exponentiation

This supports *H* being a Hamiltonian (for a Hilbert space *state*) or a Liouvillian (for *state* being a density matrix) in nested-list format. Collapse operators *c_ops* are not supported. The propagator is not stateful, thus *initialize* is ignored.

class krotov.propagators.**Propagator**

Bases: abc.ABC

Abstract base class for stateful propagators

abstract __call__(*H*, *state*, *dt*, *c_ops=None*, *backwards=False*, *initialize=False*)

Evaluation of a single propagation step

Parameters

- **H** (*list*) – A Hamiltonian or Liouvillian in qutip's nested-list format, with a scalar value in the place of a time-dependency. For example, `[H0, [H1, u]]` for a drift Hamiltonian `H0`, a control Hamiltonian `H1`, and a scalar value `u` that is a time-dependent control evaluated for a particular point in time.
- **state** (*qutip.Qobj*) – The state to propagate
- **dt** (*float*) – The time step over which to propagate
- **c_ops** (*list or None*) – A list of Lindblad operators. Using explicit Lindblad operators should be avoided: it is usually more efficient to convert them into a Lindbladian, passed as *H*
- **backwards** (*bool*) – Whether the propagation is forward in time or backward in time
- **initialize** (*bool*) – Whether the propagator should (re-)initialize for a new propagation, when the propagator is used to advance on a time grid, *initialize* should be passed as True for the initial time step (0 to *dt* in a forward propagation, or T to T-dt for a backward propagation), and False otherwise.

Note: A propagator may assume the propagation to be "sequential" when *initialize* is False. That is, the state to propagate is the result of the previous call to the propagator.

---

class krotov.propagators.**DensityMatrixODEPropagator**(*method='adams'*, *order=12*, *atol=1e-08*, *rtol=1e-06*, *nsteps=1000*, *first_step=0*, *min_step=0*, *max_step=0*, *reentrant=False*)

Bases: *krotov.propagators.Propagator*

Propagator for density matrix evolution under a Lindbladian

See qutip.solver.Options for all arguments except *reentrant*. Passing True for the *reentrant* re-initializes the propagator in every time step.

Warning: By default, the propagator is not "re-entrant". That is, you cannot use more than one instance of *DensityMatrixODEPropagator* in the same process at the same time. This limitation is due to scipy.integrate.ode with the "zvode" integrator not being re-entrant. Passing reentrant=True side-steps this problem by re-initializating scipy.integrate.ode in every time step. This makes it possible to use *DensityMatrixODEPropagator* in the optimization of multiple objectives, but creates a significant overhead.

__call__(*H*, *state*, *dt*, *c_ops=None*, *backwards=False*, *initialize=False*)
Evaluation of a single propagation step

Parameters

- **H** (*list*) – A Liouvillian superoperator in qutip's nested-list format, with a scalar value in the place of a time-dependency. For example, [L0, [L1, u]] for a drift Liouvillian L0, a control Liouvillian H1, and a scalar value u that is a time-dependent control evaluated for a particular point in time. If *initialize* is False, only the control values are taken into account; any operators are assumed to be identical to the internally cached values of *H* during initialization.
- **state** (*qutip.Qobj*) – The density matrix to propagate. The passed value is ignored unless *initialize* is given as True. Otherwise, it is assumed that *state* matches the (internally stored) state that was the result from the previous propagation step.
- **dt** (*float*) – The time step over which to propagate
- **c_ops** (*list or None*) – An empty list, or None. Since this propagator assumes a full Liouvillian, it cannot be combined with Lindblad operators.
- **backwards** (*bool*) – Whether the propagation is forward in time or backward in time. Since the equation of motion for a Liouvillian and conjugate Liouvillian is the same, this parameter has no effect. Instead, for the backward propagation, the conjugate Liouvillian must be passed for *L*.
- **initialize** (*bool*) – Whether to (re-)initialize for a new propagation. This caches *H* (except for the control values) and *state* internally.

12.1.9 krotov.result module

Summary

Classes:

Result	Result object for a Krotov optimization

__all__: *Result*

Reference

class krotov.result.**Result**

Bases: object

Result object for a Krotov optimization

objectives
The control objectives

Type list[*Objective*]

tlist
The time grid values

Type numpy.ndarray

iters
Iteration numbers, starting at 0.

Type list[int]

iter_seconds
for each iteration number, the number of seconds that were spent in the optimization

Type list[int]

info_vals
For each iteration, the return value of *info_hook*, or None

Type list

tau_vals
for each iteration, a list of complex overlaps between the target state and the forward-propagated state for each objective, assuming *Objective.target* contains the target state. If there is no target state, an empty list.

Type list[list[complex]

guess_controls
List of the guess controls in array format

Type list[numpy.ndarray]

optimized_controls
List of the optimized control fields, in the order corresponding to *guess_controls*

Type list[numpy.ndarray]

controls_mapping
A nested list that indicates where in *objectives* the *guess_controls* and *optimized_controls* are used (as returned by *extract_controls_mapping()*)

Type list

all_pulses
If the optimization was performed with `store_all_pulses=True`, for each iteration, a list of the optimized pulses (in the order corresponding to *guess_controls*). These pulses are defined at midpoints of the *tlist* intervals. Empty list if `store_all_pulses=False`

Type list

states
for each objective, a list of states for each value in *tlist*, obtained from propagation under the final optimized control fields.

Type list[list[qutip.Qobj]]

start_local_time
Time stamp of when the optimization started

Type time.struct_time

end_local_time
Time stamp of when the optimization ended

Type time.struct_time

message
Description of why *optimize_pulses()* completed, E.g, "Reached 1000 iterations"

Type str

time_fmt = '%Y-%m-%d %H:%M:%S'
Format used in *start_local_time_str* and *end_local_time_str*

property start_local_time_str
The *start_local_time* attribute formatted as a string

property end_local_time_str
The *end_local_time* attribute formatted as a string

property optimized_objectives
A copy of the *objectives* with the *optimized_controls* plugged in.

Type list[*Objective*]

objectives_with_controls(*controls*)
List of objectives with the given *controls* plugged in.

Parameters **controls** (*list[numpy.ndarray]*) – A list of control fields, defined on the points of *tlist*. Must be of the same length as *guess_controls* and *optimized_controls*.

Returns A copy of *objectives*, where all control fields are replaced by the elements of the *controls*.

Return type list[*Objective*]

Raises **ValueError** – If *controls* does not have the same number controls as *guess_controls* and *optimized_controls*, or if any *controls* are not defined on the points of the time grid.

See also:

For plugging in the optimized controls, the *optimized_objectives* attribute is equivalent to `result.objectives_with_controls(result.optimized_controls)`.

classmethod load(*filename*, *objectives=None*, *finalize=False*)
Construct *Result* object from a *dump()* file

Parameters

- **filename** (*str*) – The file from which to load the *Result*. Must be in the format created by *dump()*.

- **objectives** (*None or list[Objective]*) – If given, after loading *Result* from the given *filename*, overwrite *objectives* with the given *objectives*. This is necessary because *dump()* does not preserve time-dependent controls that are Python functions.
- **finalize** (*bool*) – If given as True, make sure that the *optimized_controls* are properly finalized. This allows to load a *Result* that was dumped before *optimize_pulses()* finished, e.g. by *dump_result()*.

Returns The *Result* instance loaded from *filename*

Return type *Result*

dump(*filename*)

Dump the *Result* to a binary `pickle` file.

The original *Result* object can be restored from the resulting file using *load()*. However, time-dependent control fields that are callables/functions will not be preserved, as they are not "pickleable".

Parameters **filename** (*str*) – Name of file to which to dump the *Result*.

12.1.10 krotov.second_order module

Support functions for the second-order update equation

Summary

Classes:

Sigma	Function σ(t) for the second order update equation.

Functions:

numerical_estimate_A	Update the second-order parameter A.

`__all__`: *Sigma*, *numerical_estimate_A*

Reference

class krotov.second_order.**Sigma**

Bases: `abc.ABC`

Function σ(t) for the second order update equation.

This is an abstract bases class. For any optimization that requires the second-order update equation, an appropriate problem-specific subclass of *Sigma* must be implemented that defines

- the evaluation of σ(t) in *__call__()*

- the update of any values that σ(t) depends on parametrically (typically: any of the parameters A, B, C), in *refresh()*.

An instantiation of that subclass is then passed as *sigma* to *optimize_pulses()*.

abstract __call__(*t*)
Evaluate σ(t)

abstract refresh(*forward_states*, *forward_states0*, *chi_states*, *chi_norms*, *optimized_pulses*, *guess_pulses*, *objectives*, *result*)
Recalculate the parametric dependencies of σ(t)

This is called at the end of each control iteration, and may be used to estimate the internal parameters in σ(t)

Parameters

- **forward_states** (*list*) – For each objective, an array-like container (cf. *storage* in *optimize_pulses()*) of the initial state forward-propagated under optimized controls from the current iteration.
- **forward_states0** (*list*) – The forward-propagated states under the guess controls of the current iteration.
- **chi_states** (*list*) – The (normalized) boundary condition for the backward-propagation in the current iteration, as returned by the *chi_constructor* argument to *optimize_pulses()*.
- **chi_norms** (*list*) – The norms of the un-normalized *chi_states*.
- **optimized_pulses** (*list[numpy.ndarray]*) – from the current iteration
- **guess_pulses** (*list[numpy.ndarray]*) – current iteration
- **objectives** (*list[Objective]*) – The control objectives
- **result** (Result) – The result object, up-to-date for the current iteration

krotov.second_order.**numerical_estimate_A**(*forward_states*, *forward_states0*, *chi_states*, *chi_norms*, *Delta_J_T*)
Update the second-order parameter A.

Calculate the new value of A according to the equation

$$A^{(i+1)} = \frac{\sum_k 2\,\mathrm{Re}\,\langle \chi_k(T) \mid \Delta\phi_k(T)\rangle + \Delta J_T}{\sum_k \langle \Delta\phi_k(T) \mid \Delta\phi_k(T)\rangle},$$

where $\Delta\phi_k$ is the difference of the *forward_states* $|\phi_k^{(i)}\rangle$ propagated under the optimized pulse of iteration (i), and the *forward_states0* $|\phi_k^{(i-1)}\rangle$ propagated under the guess pulse of iteration (i) – that is, the guess pulse of iteration $(i-1)$; and ΔJ_T is the difference of the final time functional,

$$\Delta J_T = J_T(\{|\phi_k^{(i)}(T)\rangle\} - J_T(\{|\phi_k^{(i-1)}(T)\rangle\}.$$

Parameters

- **forward_states** (*list*) – For each objective, the result of a forward-propagation with the optimized pulses of the current iteration.
- **forward_states0** (*list*) – For each objective, the result of a forward-propagation with the guess pulses of the current iteration

- **chi_states** (*list*) – For each objective, the normalized boundary state $|\chi_k(T)\rangle / ||\chi_k(T)\rangle|$ for the backward-propagation with the guess pulse of the current iteration.
- **chi_norms** (*list*) – The norms of the *chi_states*
- **Delta_J_T** (*float*) – The value by which the final time functional improved in the current iteration.

12.1.11 krotov.shapes module

Functions that may be used for the *update_shape* value in the options-dict for each control (*pulse_options* parameter in *optimize_pulses()*), or for generating guess pulses

Summary

Functions:

blackman	Blackman window shape
box	Box-shape (Theta-function)
flattop	Flat shape (one) with a switch-on/switch-off from zero
one_shape	Shape function 1 for all values of *t*
qutip_callback	Convert *func* into the correct form of a QuTiP time-dependent control
zero_shape	Shape function 0 for all values of *t*

__all__: *blackman*, *box*, *flattop*, *one_shape*, *qutip_callback*, *zero_shape*

Reference

krotov.shapes.**qutip_callback**(*func*, ***kwargs*)
Convert *func* into the correct form of a QuTiP time-dependent control

QuTiP requires that "callback" functions that are used to express time-dependent controls take a parameter *t* and *args*. This function takes a function *func* that takes *t* as its first parameter and an arbitrary number of other parameters. The given *kwargs* set values for these other parameters. Parameters not contained in *kwargs* are set *at runtime* from the *args* dict.

krotov.shapes.**zero_shape**(*t*)
Shape function 0 for all values of *t*

krotov.shapes.**one_shape**(*t*)
Shape function 1 for all values of *t*

krotov.shapes.**flattop**(*t*, *t_start*, *t_stop*, *t_rise*, *t_fall=None*, *func='blackman'*)
Flat shape (one) with a switch-on/switch-off from zero

The flattop function starts at 0, and ramps to to 1 during the *t_rise* interval. For `func='blackman'`, the switch-on shape is half of a Blackman window (see *blackman()*). For `func='sinsq`, it is a sine-squared curve. The function then remains at value 1, before ramping down to 0 again during *t_fall*.

Parameters

- **t** (*float*) – Time point or time grid
- **t_start** (*float*) – Start of flattop window
- **t_stop** (*float*) – Stop of flattop window
- **t_rise** (*float*) – Duration of ramp-up, starting at *t_start*
- **t_fall** (*float*) – Duration of ramp-down, ending at *t_stop*. If not given, `t_fall=t_rise`.
- **func** (*str*) – One of 'blackman', 'sinsq'

Note: You may use `numpy.vectorize` to transform this into a shape function for arrays, `functools.partial()` to fix the function arguments other than *t*, creating a function suitable for the *update_shape* value of *pulse_options*, and *qutip_callback()* to create a function suitable as a time-dependent control in QuTiP.

---

krotov.shapes.**box**(*t*, *t_start*, *t_stop*)

Box-shape (Theta-function)

The shape is 0 before *t_start* and after *t_stop* and 1 elsewhere.

Parameters

- **t** (*float*) – Time point or time grid
- **t_start** (*float*) – First value of *t* for which the box has value 1
- **t_stop** (*float*) – Last value of *t* for which the box has value 1

Note: You may use `numpy.vectorize`, `functools.partial()`, or *qutip_callback()*, cf. *flattop()*.

---

krotov.shapes.**blackman**(*t*, *t_start*, *t_stop*, *a=0.16*)

Blackman window shape

$$B(t; t_0, t_1) = \frac{1}{2}\left(1 - a - \cos\left(2\pi\frac{t-t_0}{t_1-t_0}\right) + a\cos\left(4\pi\frac{t-t_0}{t_1-t_0}\right)\right),$$

with $a = 0.16$.

See http://en.wikipedia.org/wiki/Window_function#Blackman_windows

A Blackman shape looks nearly identical to a Gaussian with a 6-sigma interval between *t_start* and *t_stop*. Unlike the Gaussian, however, it will go exactly to zero at the edges. Thus, Blackman pulses are often preferable to Gaussians.

Parameters

- **t** (*float or numpy.ndarray*) – Time point or time grid
- **t_start** (*float*) – Starting point t_0 of Blackman shape
- **t_stop** (*float*) – End point t_1 of Blackman shape

Returns If *t* is a float, return the value of the Blackman shape at *t*. If *t* is an array, return an array of same size as *t*, containing the values for the Blackman shape (zero before *t_start* and after *t_stop*)

Return type float or numpy.ndarray

12.1.12 krotov.structural_conversions module

Routines for converting between structures good for QuTiP's mesolve and Krotov

Summary

Functions:

control_onto_interval	Convert control on time grid to control on time grid intervals
discretize	Discretize the given *control* onto the *tlist* time grid
extract_controls	Extract a list of (unique) controls from the *objectives*
extract_controls_mapping	Extract a map of where *controls* are used in *objectives*
plug_in_pulse_values	Plug pulse values into H
pulse_onto_tlist	Convert *pulse* from time-grid intervals to time-grid points
pulse_options_dict_to_list	Convert *pulse_options* into a list

__all__: *control_onto_interval*, *discretize*, *extract_controls*, *extract_controls_mapping*, *pulse_onto_tlist*, *pulse_options_dict_to_list*

Reference

krotov.structural_conversions.**discretize**(*control*, *tlist*, *args=(None,)*, *kwargs=None*)

Discretize the given *control* onto the *tlist* time grid

If *control* is a callable, return array of values for *control* evaluated at all points in *tlist*. If *control* is already discretized, check that the discretization matches *tlist*

Parameters

- **control** (*callable or numpy.ndarray*) – control to be discretized. If callable, must take time value *t* as its first argument.
- **tlist** (*numpy.ndarray*) – time grid to discretize one
- **args** (*tuple or list*) – If *control* is a callable, further positional arguments to pass to *control*. The default passes a single value None, to match the requirements for a callable control function in QuTiP.
- **kwargs** (*None or dict*) – If *control* is callable, further keyword arguments to pass to *control*. If None, no keyword arguments will be passed.

Returns

Discretized array of real *control* values, same length as *tlist*

Return type numpy.ndarray

Raises

- **TypeError** – If *control* is not a function that takes two arguments (*t*, None), or a numpy array
- **ValueError** – If *control* is numpy array of incorrect size.

krotov.structural_conversions.**extract_controls**(*objectives*)

Extract a list of (unique) controls from the *objectives*

Controls are unique if they are not the same object, cf. Python's is keyword.

Parameters **objectives** (*list*) – List of *Objective* instances

Returns list of controls in *objectives*

See *extract_controls_mapping()* for an example.

krotov.structural_conversions.**extract_controls_mapping**(*objectives*, *controls*)

Extract a map of where *controls* are used in *objectives*

The result is a nested list where the first index relates to the *objectives*, the second index relates to the Hamiltonian (0) or the *c_ops* (1…), and the third index relates to the *controls*.

Example

```
>>> import qutip
>>> import krotov
>>> X, Y, Z = qutip.Qobj(), qutip.Qobj(), qutip.Qobj() # dummy Hams
>>> u1, u2 = np.array([]), np.array([])                # dummy controls
>>> psi0, psi_tgt = qutip.Qobj(), qutip.Qobj()         # dummy states
```

```
>>> H1 = [X, [Y, u1], [Z, u1]]  # ham for first objective
>>> H2 = [X, [Y, u2]]           # ham for second objective
>>> c_ops = [[[X, u1]], [[Y, u2]]]
>>> objectives = [
...     krotov.Objective(
...         initial_state=psi0,
...         target=psi_tgt,
...         H=H1,
...         c_ops=c_ops
...     ),
...     krotov.Objective(
...         initial_state=psi0,
...         target=psi_tgt,
...         H=H2,
...         c_ops=c_ops
...     )
... ]
>>> controls = extract_controls(objectives)
>>> assert controls == [u1, u2]
```

```
>>> controls_mapping = extract_controls_mapping(objectives, controls)
>>> controls_mapping
[[[[1, 2], []], [[0], []], [[], [0]]], [[[], [1]], [[0], []], [[], [0]]]]
```

The structure should be read as follows:

- For the first objective (0), in the Hamiltonian (0), where is the first pulse (0) used? (answer: in `H1[1]` and `H1[2]`)

```
>>> controls_mapping[0][0][0]
[1, 2]
```

- For the second objective (1), in the second `c_ops` (2), where is the second pulse (1) used? (answer: in `c_ops[1][0]`)

```
>>> controls_mapping[1][2][1]
[0]
```

- For the second objective (1), in the Hamiltonian (0), where is the first pulse (0) used? (answer: nowhere)

```
>>> controls_mapping[1][0][0]
[]
```

krotov.structural_conversions.**pulse_options_dict_to_list**(*pulse_options*, *controls*)

Convert *pulse_options* into a list

Given a dict *pulse_options* that contains an options-dict for every control in *controls* (cf. *optimize_pulses()*), return a list of the options-dicts in the same order as *controls*.

Raises **ValueError** – if *pulse_options* to not contain all of the *controls*

krotov.structural_conversions.**plug_in_pulse_values**(*H*, *pulses*, *mapping*, *time_index*, *conjugate=False*)

Plug pulse values into H

Parameters

- **H** (*list*) – nested list for a QuTiP-time-dependent operator
- **pulses** (*list*) – list of pulses in array format
- **mapping** (*list*) – nested list: for each pulse, a list of indices in *H* where pulse value should be inserted
- **time_index** (*int*) – Index of the value of each pulse that should be plugged in
- **conjugate** (*bool*) – If True, use conjugate complex pulse values

Returns a list with the same structure as *H* that contains the same `Qobj` operators as *H*, but where every time dependency is replaced by the value of the appropriate pulse at *time_index*.

Return type list

Example

```
>>> X, Y, Z = 'X', 'Y', 'Z' # dummy Hams, these would normally be Qobjs
>>> u1, u2 = np.array([0, 10, 0]), np.array([0, 20, 0])
>>> H = [X, [X, u1], [Y, u1], [Z, u2]]
>>> pulses = [u1, u2]
>>> mapping = [[1, 2], [3]]  # u1 is in H[1] and H[2], u2 is in H[3]
```

(continues on next page)

(continued from previous page)

```
>>> plug_in_pulse_values(H, pulses, mapping, time_index=1)
['X', ['X', 10], ['Y', 10], ['Z', 20]]
```

Note: It is of no consequence whether *H* contains the *pulses*, as long as it has the right structure:

```
>>> H = [X, [X, None], [Y, None], [Z, None]]
>>> plug_in_pulse_values(H, pulses, mapping, time_index=1)
['X', ['X', 10], ['Y', 10], ['Z', 20]]
```

---

krotov.structural_conversions.**control_onto_interval**(*control*)

Convert control on time grid to control on time grid intervals

Parameters **control** (*numpy.ndarray*) – values of controls on time grid

Returns pulse defined on the intervals of the time grid

Return type numpy.ndarray

The value for the first and last interval will be identical to the values at `control[0]` and `control[-1]` to ensure proper boundary conditions. All other intervals are calculated such that the original values in *control* are the average of the interval-values before and after that point in time.

The *pulse_onto_tlist()* function calculates the inverse to this transformation.

Note: For a callable *control*, call *discretize()* first.

---

krotov.structural_conversions.**pulse_onto_tlist**(*pulse*)

Convert *pulse* from time-grid intervals to time-grid points

Parameters **pulse** (*numpy.ndarray*) – values defined on the interval of a time grid

Returns values of the control defined directly on the time grid points. The size of the returned array is one greater than the size of *pulse*.

Return type numpy.ndarray

Inverse of *control_onto_interval()*.

The first and last value are also the first and last value of the returned control field. For all other points, the value is the average of the value of the input values before and after the point.

12.1.13 Summary

`__all__` Classes:

Objective	A single objective for optimization with Krotov's method.

`__all__` Functions:

ensemble_objectives	Extend *objectives* for an “ensemble optimization”
gate_objectives	Construct a list of objectives for optimizing towards a quantum gate
optimize_pulses	Use Krotov’s method to optimize towards the given *objectives*.

Bibliography

[1] A. Acín, I. Bloch, H. Buhrman, T. Calarco, C. Eichler, J. Eisert, D. Esteve, N. Gisin, S. J. Glaser, F. Jelezko, S. Kuhr, M. Lewenstein, M. F. Riedel, P. O. Schmidt, R. Thew, A. Wallraff, I. Walmsley, and F. K. Wilhelm. *The quantum technologies roadmap: a European community view*. New J. Phys. **20**, 080201 (2018). doi:10.1088/1367-2630/aad1ea.

[2] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press (2000).

[3] C. L. Degen, F. Reinhard, and P. Cappellaro. *Quantum sensing*. Rev. Mod. Phys. **89**, 035002 (2017). doi:10.1103/RevModPhys.89.035002.

[4] S. J. Glaser, U. Boscain, T. Calarco, C. P. Koch, W. Köckenberger, R. Kosloff, I. Kuprov, B. Luy, S. Schirmer, T. Schulte-Herbrüggen, D. Sugny, and F. K. Wilhelm. *Training Schrödinger's cat: quantum optimal control*. Eur. Phys. J. D **69**, 279 (2015). doi:10.1140/epjd/e2015-60464-1.

[5] D.J. Tannor, V. Kazakov, and V. Orlov. *Control of photochemical branching: novel procedures for finding optimal pulses and global upper bounds*. In J. Broeckhove and L. Lathouwers, editors, *Time-dependent quantum molecular dynamics*, pages 347–360. Plenum (1992).

[6] P. Gross, D. Neuhauser, and H. Rabitz. *Optimal control of curve-crossing systems*. J. Chem. Phys. **96**, 2834 (1992). doi:10.1063/1.461980.

[7] J. B. Murdoch, A. H. Lent, and M. R. Kritzer. *Computer-optimized narrowband pulses for multislice imaging*. J. Magnet. Res. **74**, 226 (1987). doi:10.1016/0022-2364(87)90336-2.

[8] S. J. Glaser and G. P. Drobny. *The tailored TOCSY experiment: Chemical shift selective coherence transfer*. Chem. Phys. Lett. **164**, 456 (1989). doi:10.1016/0009-2614(89)85238-8.

[9] C. P. Koch. *Controlling open quantum systems: tools, achievements, and limitations*. J. Phys.: Condens. Matter **28**, 213001 (2016). doi:10.1088/0953-8984/28/21/213001.

[10] J. Cui, R. van Bijnen, T. Pohl, S. Montangero, and T. Calarco. *Optimal control of Rydberg lattice gases*. Quantum Sci. Technol. **2**, 035006 (2017). doi:10.1088/2058-9565/aa7daf.

[11] S. Patsch, D. M. Reich, J.-M. Raimond, M. Brune, S. Gleyzes, and C. P. Koch. *Fast and accurate circularization of a Rydberg atom*. Phys. Rev. A **97**, 053418 (2018). doi:10.1103/PhysRevA.97.053418.

[12] C. Lovecchio, F. Schäfer, S. Cherukattil, M. Alí Khan, I. Herrera, F. S. Cataliotti, T. Calarco, S. Montangero, and F. Caruso. *Optimal preparation of quantum states on an atom-chip device*. Phys. Rev. A **93**, 010304 (2016). doi:10.1103/PhysRevA.93.010304.

[13] S. van Frank, M. Bonneau, J. Schmiedmayer, S. Hild, C. Gross, M. Cheneau, I. Bloch, T. Pichler, A. Negretti, T. Calarco, and S. Montangero. *Optimal control of complex atomic quantum systems*. Sci. Rep. **6**, 34187 (2016). doi:10.1038/srep34187.

[14] N. Ofek, A. Petrenko, R. Heeres, P. Reinhold, Z. Leghtas, B. Vlastakis, Y. Liu, L. Frunzio, S. M. Girvin, L. Jiang, M. Mirrahimi, M. H. Devoret, and R. J. Schoelkopf. *Extending the lifetime of a quantum bit with error correction in superconducting circuits*. Nature **536**, 441 (2016). doi:10.1038/nature18949.

[15] J. J. W. H. Sørensen, M. K. Pedersen, M. Munch, P. Haikka, J. H. Jensen, T. Planke, M. G. Andreasen, M. Gajdacz, K. Mølmer, A. Lieberoth, J. F. Sherson, and Q. M. players. *Exploring the quantum speed limit with computer games*. Nature **532**, 210 (2016). doi:10.1038/nature17620.

[16] R. W. Heeres, P. Reinhold, N. Ofek, L. Frunzio, L. Jiang, and Michel H. Devoret, and R. J. Schoelkopf. *Implementing a universal gate set on a logical qubit encoded in an oscillator*. Nature Commun. **8**, 94 (2017). doi:10.1038/s41467-017-00045-1.

[17] R. Heck, O. Vuculescu, J. J. Sørensen, J. Zoller, M. G. Andreasen, M. G. Bason, P. Ejlertsen, O. Elíasson, P. Haikka, J. S. Laustsen, L. L. Nielsen, A. Mao, R. Müller, M. Napolitano, M. K. Pedersen, A. R. Thorsen, C. Bergenholtz, T. Calarco, S. Montangero, and J. F. Sherson. *Remote optimization of an ultracold atoms experiment by experts and citizen scientists*. Proc. Nat. Acad. Sci. **115**, E11231 (2018). doi:10.1073/pnas.1716869115.

[18] G. Feng, F. H. Cho, H. Katiyar, J. Li, D. Lu, J. Baugh, and R. Laflamme. *Gradient-based closed-loop quantum optimal control in a solid-state two-qubit system*. Phys. Rev. A **98**, 052341 (2018). doi:10.1103/PhysRevA.98.052341.

[19] A. Omran, H. Levine, A. Keesling, G. Semeghini, T. T. Wang, S. Ebadi, H. Bernien, A. S. Zibrov, H. Pichler, S. Choi, J. Cui, M. Rossignolo, P. Rembold, S. Montangero, T. Calarco, M. Endres, M. Greiner, V. Vuletić, and M. D. Lukin. *Generation and manipulation of Schrödinger cat states in Rydberg atom arrays*. Science **365**, 570 (2019). doi:10.1126/science.aax9743.

[20] A. Larrouy, S. Patsch, and others. in preparation.

[21] N. Khaneja, T. Reiss, C. Kehlet, T. Schulte-Herbrüggen, and S. J. Glaser. *Optimal control of coupled spin dynamics: design of NMR pulse sequences by gradient ascent algorithms*. J. Magnet. Res. **172**, 296 (2005). doi:10.1016/j.jmr.2004.11.004.

[22] D. M. Reich, M. Ndong, and C. P. Koch. *Monotonically convergent optimization in quantum control using Krotov's method*. J. Chem. Phys. **136**, 104103 (2012). doi:10.1063/1.3691827.

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

[24] M. M. Müller, H. Haakh, T. Calarco, C. P. Koch, and C. Henkel. *Prospects for fast Rydberg gates on an atom chip*. Quantum Inf. Process. **10**, 771 (2011). doi:10.1007/s11128-011-0296-0.

[25] M. H. Goerz, E. J. Halperin, J. M. Aytac, C. P. Koch, and K. B. Whaley. *Robustness of high-fidelity rydberg gates with single-site addressability*. Phys. Rev. A **90**, 032329 (2014). doi:10.1103/PhysRevA.90.032329.

[26] M. H. Goerz, D. M. Reich, and C. P. Koch. *Optimal control theory for a unitary operation under dissipative evolution*. New J. Phys. **16**, 055012 (2014). doi:10.1088/1367-2630/16/5/055012.

[27] P. Watts, J. Vala, M. M. Müller, T. Calarco, K. B. Whaley, D. M. Reich, M. H. Goerz, and C. P. Koch. *Optimizing for an arbitrary perfect entangler: I. Functionals*. Phys. Rev. A **91**, 062306 (2015). doi:10.1103/PhysRevA.91.062306.

[28] M. H. Goerz, G. Gualdi, D. M. Reich, C. P. Koch, F. Motzoi, K. B. Whaley, J. Vala, M. M. Müller, S. Montangero, and T. Calarco. *Optimizing for an arbitrary perfect entangler. II. Application*. Phys. Rev. A **91**, 062307 (2015). doi:10.1103/PhysRevA.91.062307.

[29] D. Basilewitsch, R. Schmidt, D. Sugny, S. Maniscalco, and C. P. Koch. *Beating the limits with initial correlations*. New J. Phys. **19**, 113042 (2017). doi:10.1088/1367-2630/aa96f8.

[30] J. Preskill. *Quantum computing in the NISQ era and beyond*. Quantum **2**, 79 (2018). doi:10.22331/q-2018-08-06-79.

[31] M. H. Goerz, K. B. Whaley, and C. P. Koch. *Hybrid optimization schemes for quantum control*. EPJ Quantum Technol. **2**, 21 (2015). doi:10.1140/epjqt/s40507-015-0034-0.

[32] M. H. Goerz, F. Motzoi, K. B. Whaley, and C. P. Koch. *Charting the circuit-QED design landscape using optimal control theory*. npj Quantum Inf. **3**, 37 (2017). doi:10.1038/s41534-017-0036-0.

[33] J. Bezanson, A. Edelman, S. Karpinski, and V. Shah. *Julia: a fresh approach to numerical computing*. SIAM Rev. **59**, 65 (2017). doi:10.1137/141000671.

[34] J. Akeret, L. Gamper, A. Amara, and A. Refregier. *HOPE: a Python just-in-time compiler for astrophysical computations*. Astron. Comput. **10**, 1 (2015). doi:10.1016/j.ascom.2014.12.001.

[35] H. Eichhorn, J. L. Cano, F. McLean, and R. Anderl. *A comparative study of programming languages for next-generation astrodynamics systems*. CEAS Space J. **10**, 115 (2018). doi:10.1007/s12567-017-0170-8.

[36] D. J. Tannor and S. A. Rice. *Control of selectivity of chemical reaction via control of wave packet evolution*. J. Chem. Phys. **83**, 5013 (1985). doi:10.1063/1.449767.

[37] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ (1957).

[38] L. S. Pontryagin, V. G. Boltyanskii, G. R. V., and E. F. Mishchenko. *The Mathematical Theory of Optimal Processes*. Interscience, New York, NY (1962).

[39] V. F. Krotov and I. N. Fel'dman. Engrg. Cybernetics **21**, 123 (1984).

[40] V. F. Krotov. Control and Cybernetics **17**, 115 (1988).

[41] V. Krotov. *Global Methods in Optimal Control Theory*. CRC Press (1995).

[42] A.I. Konnov and V. F. Krotov. *On global methods of successive improvement of controlled processes*. Autom. Rem. Contr. **60**, 1427 (1999).

[43] J. Somlói, V. A. Kazakov, and D. J. Tannor. *Controlled dissociation of I2 via optical transitions between the X and B electronic states*. Chem. Phys. **172**, 85 (1993). doi:10.1016/0301-0104(93)80108-L.

[44] A. Bartana, R. Kosloff, and D. J. Tannor. *Laser cooling of internal degrees of freedom. II*. J. Chem. Phys. **106**, 1435 (1997). doi:10.1063/1.473973.

[45] S. E. Sklarz and D. J. Tannor. *Loading a Bose-Einstein condensate onto an optical lattice: an application of optimal control theory to the nonlinear Schrödinger equation*. Phys. Rev. A **66**, 053619 (2002). doi:10.1103/PhysRevA.66.053619.

[46] J. P. Palao and R. Kosloff. *Optimal control theory for unitary transformations*. Phys. Rev. A **68**, 062308 (2003). doi:10.1103/PhysRevA.68.062308.

[47] A. Kaiser and V. May. *Optimal control theory for a target state distributed in time: optimizing the probe-pulse signal of a pump-probe-scheme*. J. Chem. Phys. **121**, 2528 (2004). doi:10.1063/1.1769370.

[48] I. Serban, J. Werschnik, and E. K. U. Gross. *Optimal control of time-dependent targets*. Phys. Rev. A **71**, 053810 (2005). doi:10.1103/PhysRevA.71.053810.

[49] J. P. Palao, R. Kosloff, and C. P. Koch. *Protecting coherence in optimal control theory: state-dependent constraint approach*. Phys. Rev. A **77**, 063412 (2008). doi:10.1103/PhysRevA.77.063412.

[50] A. Bartana, R. Kosloff, and D. J. Tannor. *Laser cooling of molecular internal degrees of freedom by a series of shaped pulses*. J. Chem. Phys. **99**, 196 (1993). doi:10.1063/1.465797.

[51] Y. Ohtsuki, W. Zhu, and H. Rabitz. *Monotonically convergent algorithm for quantum optimal control with dissipation*. J. Chem. Phys. **110**, 9825 (1999). doi:10.1063/1.478036.

[52] M. H. Goerz and K. Jacobs. *Efficient optimization of state preparation in quantum networks using quantum trajectories*. Quantum Sci. Technol. **3**, 045005 (2018). doi:10.1088/2058-9565/aace16.

[53] D. Jaksch, J. I. Cirac, P. Zoller, S. L. Rolston, R. Côté, and M. D. Lukin. *Fast quantum gates for neutral atoms*. Phys. Rev. Lett. **85**, 2208 (2000). doi:10.1103/PhysRevLett.85.2208.

[54] M. M. Müller, D. M. Reich, M. Murphy, H. Yuan, J. Vala, K. B. Whaley, T. Calarco, and C. P. Koch. *Optimizing entangling quantum gates for physical systems*. Phys. Rev. A **84**, 042315 (2011). doi:10.1103/PhysRevA.84.042315.

[55] P. Watts, M. O'Connor, and J. Vala. *Metric structure of the space of two-qubit gates, perfect entanglers and quantum control*. Entropy **15**, 1963 (2013). doi:10.3390/e15061963.

[56] M. Musz, M. Kuś, and K. Życzkowski. *Unitary quantum gates, perfect entanglers, and unistochastic maps*. Phys. Rev. A **87**, 022111 (2013). doi:10.1103/PhysRevA.87.022111.

[57] M. B. Plenio and P. L. Knight. *The quantum-jump approach to dissipative dynamics in quantum optics*. Rev. Mod. Phys. **70**, 101 (1998). doi:10.1103/RevModPhys.70.101.

[58] D. M. Reich, J. P. Palao, and C. P. Koch. *Optimal control under spectral constraints: enforcing multi-photon absorption pathways*. J. Mod. Opt. **61**, 822 (2014). doi:10.1080/09500340.2013.844866.

[59] M. M. Müller, D. M. Reich, M. Murphy, H. Yuan, J. Vala, K. B. Whaley, T. Calarco, and C. P. Koch. *Optimizing entangling quantum gates for physical systems*. Phys. Rev. A **84**, 042315 (2011). doi:10.1103/PhysRevA.84.042315.

[60] R. Kosloff, S.A. Rice, P. Gaspard, S. Tersigni, and D.J. Tannor. *Wavepacket dancing: achieving chemical selectivity by shaping light pulses*. Chem. Phys. **139**, 201 (1989). doi:10.1016/0301-0104(89)90012-8.

[61] S. Shi and H. Rabitz. *Quantum mechanical optimal control of physical observables in microsystems*. J. Chem. Phys. **92**, 364 (1990). doi:10.1063/1.458438.

[62] S. Shi and H. Rabitz. *Optimal control of bond selectivity in unimolecular reactions*. Comput. Phys. Commun. **63**, 71 (1991). doi:10.1016/0010-4655(91)90239-H.

[63] D. J. Tannor and Y. Jin. *Design of femtosecond pulse sequences to control photochemical products*. In *Mode Selective Chemistry*, pages 333–345. Springer (1991).

[64] W. Zhu, J. Botina, and H. Rabitz. *Rapidly convergent iteration methods for quantum optimal control of population*. J. Chem. Phys. **108**, 1953 (1998). doi:10.1063/1.475576.

[65] Y. Maday and G. Turinici. *New formulations of monotonically convergent quantum control algorithms*. J. Chem. Phys. **118**, 8191 (2003). doi:10.1063/1.1564043.

[66] Y. Ohtsuki, G. Turinici, and H. Rabitz. *Generalized monotonically convergent algorithms for solving quantum optimal control problems*. J. Chem. Phys. **120**, 5509 (2004). doi:10.1063/1.1650297.

[67] J. Werschnik and E. K. U. Gross. *Quantum optimal control theory*. J. Phys. B **40**, R175 (2007). doi:10.1088/0953-4075/40/18/r01.

[68] B. Schmidt and C. Hartmann. *WavePacket: a Matlab package for numerical quantum dynamics. II: open quantum systems, optimal control, and model reduction*. Comput. Phys. Commun. **228**, 229 (2018). doi:10.1016/j.cpc.2018.02.022.

[69] S. G. Schirmer and P. de Fouquieres. *Efficient algorithms for optimal control of quantum dynamics: the krotov method unencumbered*. New J. Phys. **13**, 073029 (2011). doi:10.1088/1367-2630/13/7/073029.

[70] S. Machnes, U. Sander, S. J. Glaser, P. de Fouquières, A. Gruslys, S. Schirmer, and T. Schulte-Herbrüggen. *Comparing, optimizing, and benchmarking quantum-control algorithms in a unifying programming framework*. Phys. Rev. A **84**, 022305 (2011). doi:10.1103/PhysRevA.84.022305.

[71] R. Eitan, M. Mundt, and D. J. Tannor. *Optimal control with accelerated convergence: combining the Krotov and quasi-Newton methods*. Phys. Rev. A **83**, 053426 (2011). doi:10.1103/PhysRevA.83.053426.

[72] J. P. Palao and R. Kosloff. *Quantum computing by an optimal control algorithm for unitary transformations*. Phys. Rev. Lett. **89**, 188301 (2002). doi:10.1103/PhysRevLett.89.188301.

[73] M. Goerz. *Optimizing Robust Quantum Gates in Open Quantum Systems*. PhD thesis, Universität Kassel, (2015). URL: https://kobra.bibliothek.uni-kassel.de/handle/urn:nbn:de:hebis:34-2015052748381.

[74] S. Schirmer. *Implementation of quantum gates via optimal control*. J. Mod. Opt. **56**, 831 (2009). doi:10.1080/09500340802344933.

[75] F. F. Floether, P. de Fouquières, 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.

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

[77] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal. *Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization*. ACM Trans. Math. Softw. **23**, 550 (1997). doi:10.1145/279232.279236.

[78] P. de Fouquières, S.G. Schirmer, S.J. Glaser, and I. Kuprov. *Second order gradient ascent pulse engineering*. J. Magnet. Res. **212**, 412 (2011). doi:10.1016/j.jmr.2011.07.023.

[79] E. Jones, T. Oliphant, P. Peterson, and others. *SciPy: open source scientific tools for Python*. (2001–). URL: http://www.scipy.org/.

[80] G. Jäger, D. M. Reich, M. H. Goerz, C. P. Koch, and U. Hohenester. *Optimal quantum control of Bose-Einstein condensates in magnetic microtraps: comparison of GRAPE and Krotov optimization schemes*. Phys. Rev. A **90**, 033628 (2014). doi:10.1103/PhysRevA.90.033628.

[81] J. L. Neves, B. Heitmann, N. Khaneja, and S. J. Glaser. *Heteronuclear decoupling by optimal tracking*. J. Magnet. Res. **201**, 7 (2009). doi:10.1016/j.jmr.2009.07.024.

[82] T. T. Nguyen and S. J. Glaser. *An optimal control approach to design entire relaxation dispersion experiments*. J. Magnet. Res. **282**, 142 (2017). doi:10.1016/j.jmr.2017.07.010.

[83] Q. Ansel, M. Tesch, S. J. Glaser, and D. Sugny. *Optimizing fingerprinting experiments for parameter identification: application to spin systems*. Phys. Rev. A **96**, 053419 (2017). doi:10.1103/PhysRevA.96.053419.

[84] P. E. Spindler, Y. Zhang, B. Endeward, N. Gershernzon, T. E. Skinner, S. J. Glaser, and T. F. Prisner. *Shaped optimal control pulses for increased excitation bandwidth in epr*. J. Magnet. Res. **218**, 49 (2012). doi:10.1016/j.jmr.2012.02.013.

[85] Z. Tošner, R. Sarkar, J. Becker-Baldus, C. Glaubitz, S. Wegner, F. Engelke, S. J. Glaser, and B. Reif. *Overcoming volume selectivity of dipolar recoupling in biological solid-state NMR spectroscopy*. Angew. Chem. Int. Ed. **57**, 14514 (2018). doi:10.1002/anie.201805002.

[86] N. Leung, M. Abdelhafez, J. Koch, and D. Schuster. *Speedup for quantum optimal control from automatic differentiation based on graphics processing units*. Phys. Rev. A **95**, 042318 (2017). doi:10.1103/PhysRevA.95.042318.

[87] M. Abdelhafez, D. I. Schuster, and J. Koch. *Gradient-based optimal control of open quantum systems using quantum trajectories and automatic differentiation*. Phys. Rev. A **99**, 052327 (2019). doi:10.1103/PhysRevA.99.052327.

[88] T. E. Skinner and N. I. Gershenzon. *Optimal control design of pulse shapes as analytic functions*. J. Magnet. Res. **204**, 248 (2010). doi:10.1016/j.jmr.2010.03.002.

[89] F. Motzoi, J. M. Gambetta, S. T. Merkel, and F. K. Wilhelm. *Optimal control methods for rapidly time-varying hamiltonians*. Phys. Rev. A **84**, 022307 (2011). doi:10.1103/PhysRevA.84.022307.

[90] J. J. W. H. Sørensen, M. O. Aranburu, T. Heinzel, and J. F. Sherson. *Quantum optimal control in a chopped basis: applications in control of Bose-Einstein condensates*. Phys. Rev. A **98**, 022119 (2018). doi:10.1103/PhysRevA.98.022119.

[91] S. Machnes, E. Assémat, D. Tannor, and F. K. Wilhelm. *Tunable, flexible, and efficient optimization of control pulses for practical qubits*. Phys. Rev. Lett. **120**, 150401 (2018). doi:10.1103/PhysRevLett.120.150401.

[92] 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.

[93] R. E. Goetz, A. Karamatskou, R. Santra, and C. P. Koch. *Quantum optimal control of photoelectron spectra and angular distributions*. Phys. Rev. A **93**, 013413 (2016). doi:10.1103/PhysRevA.93.013413.

[94] K. P. Horn, F. Reiter, Y. Lin, D. Leibfried, and C. P. Koch. *Quantum optimal control of the dissipative production of a maximally entangled state*. New J. Phys. **20**, 123010 (2018). doi:10.1088/1367-2630/aaf360.

[95] 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.

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

[97] S. G. Johnson. *The NLopt nonlinear-optimization package*. http://ab-initio.mit.edu/nlopt.

[98] J. Rapin and O. Teytaud. *Nevergrad - A gradient-free optimization platform*. https://GitHub.com/FacebookResearch/Nevergrad, (2018).

[99] I.R. Petersen. *Quantum control theory and applications: a survey*. IET Control Theory & Applications **4**, 2651 (2010). doi:10.1049/iet-cta.2009.0508.

[100] M. H. Goerz, F. Motzoi, K. B. Whaley, and C. P. Koch. *Charting the circuit QED design landscape using optimal control theory*. npj Quantum Information **3**, 37 (2017). doi:10.1038/s41534-017-0036-0.

[101] T. Caneva, M. Murphy, T. Calarco, R. Fazio, S. Montangero, V. Giovannetti, and G. E. Santoro. *Optimal control at the quantum speed limit*. Phys. Rev. Lett. **103**, 240501 (2009). doi:10.1103/PhysRevLett.103.240501.

[102] M. H. Goerz, T. Calarco, and C. P. Koch. *The quantum speed limit of optimal controlled phasegates for trapped neutral atoms*. J. Phys. B **44**, 154011 (2011). doi:10.1088/0953-4075/44/15/154011.

Python Module Index

k