sckit-quant

Release 0.8.1

Dec 10, 2020

Contents

Scikit-quant is a collection of optimizers tuned for usage on Noisy Inter-mediate-Scale Quantum (NISQ) devices. Results for several VQE and Hubbard model case studies are presented in this arxiv paper (final paper was presented at IEEE's QCE'20). This is the manual for the software used.

CHAPTER 1

Changelog

1.1 master: 0.8.0

- Added SQNomand
- Require rpy2 to be installed separately as ORBIT use is uncommon

1.2 2020-05-22: 0.7.0

- Added Qiskit interoperability interface
- Added SciPy interoperability interface
- Fixed a couple of logic flow bugs in SnobFit
- Fix number of requests if nreq=1 (SnobFit; Jan Rittig)
- Fix indexing error and double delete in NaN handling (SnobFit; Jan Rittig)

1.3 2020-04-28: 0.6.0

- Remove use of numpy.matrix in SQImFil
- Bug fixes in SQCommon and SQSnobFit
- Start of documentation

CHAPTER 2

License and copyright

scikit-quant includes several optimizers with permission of their respective authors and all files are individually marked as such. This permission extends to redistribution for academic and personal use. For commercial use of any of the optimizers, please read the relevant license and check with original authors where needed. The translations from MATLAB to Python are derived works and are packaged independently on PyPI to allow their inclusion/exclusion as needed.

2.1 Main scikit-quant code

The following license applies to the common scikit-quant code and any of our changes/additions.

You are under no obligation whatsoever to provide any bug fixes, patches, or upgrades to the features, functionality or performance of the source code ("Enhancements") to anyone; however, if you choose to make your Enhancements available either publicly, or directly to Lawrence Berkeley National Laboratory, without imposing a separate written license agreement for such Enhancements, then you hereby grant the following license: a non-exclusive, royalty-free perpetual license to install, use, modify, prepare derivative works, incorporate into other computer software, distribute, and sublicense such Enhancements or derivative works thereof, in binary and source code form.

2.2 Additional copyright holders

In addition to LBNL/UC Berkeley, this package contains files copyrighted by one or more of the following people and organizations, and licensed under the same conditions (except for some compatible licenses as retained in the source code):

- Jan Rittig
- Matthias Degroote

CHAPTER 3

Installation

To install with `pip` through PyPI, it is recommend to use virtualenv (or module venv for modern pythons). The use of virtualenv prevents pollution of any system directories and allows you to wipe out the full installation simply by removing the virtualenv created directory ("WORK" in this example):

```
$ virtualenv WORK
$ source WORK/bin/activate
(WORK) $ python -m pip install scikit-quant
(WORK) $
```

If you use `pip` directly on the command line, instead of through `python`, make sure that the `PATH` envar points to the bin directory that will contain the installed entry points during the installation, as the build process needs them. You may also need to install `wheel` first if you have an older version of `pip` and/or do not use virtualenv (which installs wheel by default). Older versions of `pip` may also require the `--user` option, if you do not have write access to the installation directory. Example:

```
$ python -m pip install wheel --user
$ PATH=$HOME/.local/bin:$PATH python -m pip install scikit-quant --user
```

Alternatively, you can use `pip` from a `conda` environment:

```
$ conda create -n WORK
$ conda activate WORK
(WORK) $ python -m pip install scikit-quant
(WORK) $
```

CHAPTER 4

Trying it out

This is a basic guide to using the optimizers mainly intended to test whether your installation works. If you are already familiar to using optimizers within a quantum programming framework, you may be better served using the interoperability interfaces, such as the ones to *Qiskit* and *SciPy*.

First, you need to have some objective function to optimize. All the optimizers are *minimizers* and expect to do simple "less than" comparisons on the result. Thus, if instead you need to maximize the result, simply add a minus sign. The objective function is expected to accept an evaluation point in the form of a `numpy` array of floating point values, or a list of such evaluation points to allow evaluation in parallel.

Example of an objective function:

```
import numpy as np

# some interesting objective function to minimize
def objective_function(x):
    fv = np.inner(x, x)
    fv *= 1 + 0.1*np.sin(10*(x[0]+x[1]))
    return np.random.normal(fv, 0.01)
```

All optimizers provided require bounds. This is not true for optimizers in general, but is of such great benefit when dealing with noisy objective functions that it is pretty much a requirement. In most cases, the better the bounds, the faster the optimizer will run and the higher the quality of the result. For difficult problems, it may be necessary to refine bounds while switching optimizers to solve. Not all optimizers are equally sensitive to bounds.

```
# create a numpy array of bounds, one (low, high) for each parameter
bounds = np.array([[-np.pi, np.pi], [-np.pi, np.pi]], dtype=float)
```

Likewise, consider whether a good initial estimate can be provided, and if yes, it is often worthwhile to spend some (classical) computational resources to obtain a high quality initial estimate. Not every optimizer benefits equally of a good initial estimate, but most do, especially when combined with tight bounds. If no initial estimate is provided, a random point is used within the given bounds.

```
# initial values for all parameters
x0 = np.array([0.5, 0.5])
```

The objective function is considered expensive to calculate (running a circuit many times on the QPU). It is therefore important to consider a *budget* (number of allowed evaluations), rather than to rely solely on convergence criteria, especially since tight tolerances can not alway be met in the case of large noise. The budget is always an upper limit: if convergence happens earlier, the minimizer will stop.

```
# budget (number of calls, assuming 1 count per call)
budget = 100
```

Finally, import and run the minimizer. The result object will contain the optimal parameters (`result.optpar`) and optimal value (`result.optval`). The history object contains the full call history.

```
from skquant.opt import minimize

# method can be ImFil, SnobFit, NOMAD, Orbit, or Bobyqa (case insensitive)
result, history = \
    minimize(objective_function, x0, bounds, budget, method='imfil')
```

CHAPTER 5

Bugs and feedback

Please report bugs, ask questions, request improvements, and post general comments on the issue tracker.

CHAPTER 6

ImFil

Implicit Filtering (ImFil) is an algorithm designed for problems with local minima caused by high-frequency, low-amplitude noise and with an underlying large scale structure that is easily optimized. ImFil uses difference gradients during the search and can be considered as an extension of coordinate search. In ImFil, the optimization is controlled by evaluating the objective function at a cluster (or stencil) of points within the given bounds. The minimum of those evaluations then drives the next cluster of points, using first-order interpolation to estimate the derivative, and aided by user-provided exploration directions, if any. Convergence is reached if the "budget" for objective function evaluations is spent, if the smallest cluster size has been reached, or if incremental improvement drops below a preset threshold.

The initial clusters of points are almost completely determined by the problem boundaries, making ImFil relatively insensitive to the initial solution and allows it to easily escape from local minima. Conversely, this means that if the initial point is known to be of high quality, ImFil must be provided with tight bounds around this point, or it will unnecessarily evaluate points in regions that do not contain the global minimum.

As a practical matter, for the noisy objective functions we studied, we find that the total number of evaluations is driven almost completely by the requested step sizes between successive clusters, rather than finding convergence explicitly.

We have rewritten the original ImFil MATLAB implementation in Python

Reference: C.T. Kelley, "Implicit Filtering", 2011, ISBN: 978-1-61197-189-7

Original software available at http://ctk.math.ncsu.edu/imfil.html

CHAPTER 7

SnobFit

Stable Noisy Optimization by Branch and FIT (SnobFit) is an optimizer developed specifically for optimization problems with noisy and expensive to compute objective functions. SnobFit iteratively selects a set of new evaluation points such that a balance between global and local search is achieved, and thus the algorithm can escape from local optima. Each call to SnobFit requires the input of a set of evaluation points and their corresponding function values and SnobFit returns a new set of points to be evaluated, which is used as input for the next call of SnobFit. Therefore, in a single optimization, SnobFit is called several times. The initial set of points is provided by the user and should contain as many expertly chosen points as possible (if too few are given, the choice is a uniformly random set of points, and thus providing good bounds becomes important). In addition to these points, the user can also specify the uncertainties associated with each function value. We have not exploited this feature in our test cases, because although we know the actual noise values from the simulation, properly estimating whole-circuit systematic errors from real hardware is an open problem.

As the name implies, SnobFit uses a branching algorithm that recursively subdivides the search space into smaller subregions from which evaluation points are chosen. In order to search locally, SnobFit builds a local quadratic model around the current best point and minimizes it to select one new evaluation point. Other local search points are chosen as approximate minimizers within a trust region defined by safeguarded nearest neighbors. Finally, SnobFit also generates points in unexplored regions of the parameter space and this represents the more global search aspect.

We have rewritten the original SnobFit MATLAB implementation in Python

Reference: W. Huyer and A. Neumaier, “Snobfit - Stable Noisy Optimization by Branch and Fit”, ACM Trans. Math. Software 35 (2008), Article 9.

Original software available at http://www.mat.univie.ac.at/~neum/software/snobfit

CHAPTER 8

NOMAD

NOMAD, or "Nonlinear Optimization by Mesh Adaptive Direct Search (MADS)" is a C++ implementation of the MADS algorithm. MADS searches the parameter space by iteratively generating a new sample point from a mesh that is adaptively adjusted based on the progress of the search. If the newly selected sample point does not improve the current best point, the mesh is refined. NOMAD uses two steps ({em search} and {em poll}) alternately until some preset stopping criterion (such as minimum mesh size, maximum number of failed consecutive trials, or maximum number of steps) is met. The search step can return any point on the current mesh, and therefore offers no convergence guarantees. % if the objective function results are noisy. If the search step fails to find an improved solution, the poll step is used to explore the neighborhood of the current best solution. The poll step is central to the convergence analysis of NOMAD, and therefore any hyperparameter optimization or other tuning to make progress should focus on the poll step. Options include: poll direction type (local model, random, uniform angles, etc.), poll size, and number of polling points.

The use of meshes means that the number of evaluations needed scales at least geometrically with the number of parameters to be optimized. It is therefore important to restrict the search space as much as possible using bounds and, if the science of the problem so indicates, give preference to polling directions of the more important parameters.

We incorporate the published open-source NOMAD code through a modified Python interface.

Reference: S. Le Digabel. "NOMAD: Nonlinear Optimization with the MADS algorithm." ACM Trans. on Mathematical Software, 37(4):44:1–44:15, 2011.

Software available at: https://www.gerad.ca/nomad/

CHAPTER 9

PyBobyqa

BOBYQA (Bound Optimization BY Quadratic Approximation) has been designed to minimize bound constrained black-box optimization problems. BOBYQA employs a trust region method and builds a quadratic approximation in each iteration that is based on a set of automatically chosen and adjusted interpolation points. New sample points are iteratively created by either a "trust region" or an "alternative iterations" step. In both methods, a vector (step) is chosen and added to the current iterate to obtain the new point. In the trust region step, the vector is determined such that it minimizes the quadratic model around the current iterate and lies within the trust region. It is also ensured that the new point (the sum of the vector and the current iterate) lies within the parameter upper and lower bounds. BOBYQA uses the alternative iteration step whenever the norm of the vector is too small, and would therefore reduce the accuracy of the quadratic model. In that case, the vector is chosen such that good linear independence of the interpolation points is obtained. The current best point is updated with the new point if the new function value is better than the current best function value. Note that there are some restrictions for the choice of the initial point due to the requirements for constructing the quadratic model. BOBYQA may thus adjust the initial automatically if needed.

Although it is not intuitively obvious that BOBYQA would work well on noisy problems, we find that it performs well in practice if the initial parameters are quite close to optimal and the minimum and maximum sizes of the trust region are properly set. This is rather straightforward to do for the specific case of VQE, where a good initial guess can be obtained relatively cheaply from classical simulation. For Hubbard model problems, which have many (shallow) local minima, BOBYQA does not perform nearly as well.

We use the existing PyBobyqa implementation directly from PyPI.

Reference: Coralia Cartis, et. al., "Improving the Flexibility and Robustness of Model-Based Derivative-Free Optimization Solvers", technical report, University of Oxford, (2018).

Software available at http://github.com/numericalalgorithmsgroup/pybobyqa/

CHAPTER 10

Qiskit

A set of interoperable components for Qiskit Aqua's optimizer package are available in `skquant.interop.qiskit`. These classes derive from Qiskit's `Optimizer` class and implement the same interface, such that the skquant optimmizers can be used as drop-in replacements in Qiskit-based codes.

Caution: The optimizer classes in Qiskit's `Optimizer` package do not follow proper conventions themselves. In writing the interop component classes, an attempt was made to stick to the most prevalent conventions present as of `Aqua` version 0.7.1.

Example usage:

```
from skquant.interop.qiskit import SnobFit

x0 = np.array([0.5, 0.5])
bounds = np.array([[-1, 1], [-1, 1]], dtype=float)

optimizer = SnobFit(maxfun=40, maxmp=len(x0)+6)

ret = optimizer.optimize(num_vars=len(x0),
                         objective_function=your_objective,
                         variable_bounds=bounds,
                         initial_point=x0)
```

Available component classes are `ImFil`, `SnobFit`, `Nomad`, and `PyBobyqa`.

CHAPTER 11

SciPy

A set of interoperable methods for SciPy's optimizer package are available in `skquant.interop.scipy`. These methods follow the SciPy convention, allowing them to be passed to its `minimize` function, such that the skquant optimmizers can be used as drop-in replacements in SciPy-based codes

Example usage:

```
from skquant.interop.scipy import imfil
from scipy.optimize import minimize

x0 = np.array([0.5, 0.5])
bounds = np.array([[-1, 1], [-1, 1]], dtype=float)
budget = 40

result = minimize(your_objective, x0, method=imfil,
                  bounds=bounds, options={'budget' : budget})
```

The returned `result` is a `scipy.optimize.OptimizeResult` object and follows the same conventions for all return parameters that make sense. Available component classes are `imfil`, `snobfit`, and `pybobyqa`.

CHAPTER 12

Repositories

The developers repository is on github.

CHAPTER 13

Test suite

All packages have a `test` subdirectory that contains tests runnable by `pytest`. In addition, the top-level `test` has a `test_all.sh` bash script to walk the directories and run all tests.

To install `pytest`:

```
$ python -m pip install pytest
```

and to run any test, simply enter the `test` subdirectory and run:

```
$ pytest
```

Some commonly used pytest parameters:

```
-h : print help

-x : stop on the first failing test
-v : verbose
-s : show captured output

<file name> : run only tests from <file name>
-k <expr>   : run only tests containing <expr> in their name
```

CHAPTER 14

Bugs and feedback

Please report bugs or requests for improvement on the issue tracker.