Py-BOBYQA Documentation

Release 1.5

Lindon Roberts

09 September 2024

CONTENTS:

Release: 1.5

Date: 09 September 2024

Author: Lindon Roberts

Py-BOBYQA is a flexible package for finding local solutions to nonlinear, nonconvex minimization problems (with optional bound and other convex constraints), without requiring any derivatives of the objective. Py-BOBYQA is a Python implementation of the BOBYQA solver by Powell (documentation here). It is particularly useful when evaluations of the objective function are expensive and/or noisy.

That is, Py-BOBYQA solves

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad & f(x) \\ \text{s.t.} \quad & a \leq x \leq b \\ & x \in C := C_1 \cap \cdots \cap C_n, \quad \text{all } C_i \text{ convex} \end{aligned}$$

If provided, the constraints the variables are non-relaxable (i.e. Py-BOBYQA will never ask to evaluate a point outside the bounds), although the general $x \in C$ constraint may be slightly violated from rounding errors.

Full details of the Py-BOBYQA algorithm are given in our papers:

1. Coralia Cartis, Jan Fiala, Benjamin Marteau and Lindon Roberts, Improving the Flexibility and Robustness of Model-Based Derivative-Free Optimization Solvers, *ACM Transactions on Mathematical Software*, 45:3 (2019), pp. 32:1-32:41 [preprint]
2. Coralia Cartis, Lindon Roberts and Oliver Sheridan-Methven, Escaping local minima with derivative-free methods: a numerical investigation, *Optimization*, 71:8 (2022), pp. 2343-2373. [arXiv preprint: 1812.11343]
3. Lindon Roberts, Model Construction for Convex-Constrained Derivative-Free Optimization, *arXiv preprint arXiv:2403.14960* (2024).

Please cite [1] when using Py-BOBYQA for local optimization, [1,2] when using Py-BOBYQA's global optimization heuristic functionality, and [1,3] if using the general convex constraints $x \in C$ functionality.

If you are interested in solving least-squares minimization problems, you may wish to try DFO-LS, which has the same features as Py-BOBYQA (plus some more), and exploits the least-squares problem structure, so performs better on such problems.

Since v1.1, Py-BOBYQA has a heuristic for global optimization (see *Using Py-BOBYQA* for details). As this is a heuristic, there are no guarantees it will find a global minimum, but it is more likely to escape local minima if there are better values nearby.

CHAPTER

ONE

INSTALLING PY-BOBYQA

1.1 Requirements

Py-BOBYQA requires the following software to be installed:

- Python 3.8 or higher (http://www.python.org/)

Additionally, the following python packages should be installed (these will be installed automatically if using pip, see *Installation using pip*):

- NumPy (http://www.numpy.org/)
- SciPy (http://www.scipy.org/)
- Pandas (http://pandas.pydata.org/)

Optional package: Py-BOBYQA versions 1.2 and higher also support the trustregion package for fast trust-region subproblem solutions. To install this, make sure you have a Fortran compiler (e.g. gfortran) and NumPy installed, then run `pip install trustregion`. You do not have to have trustregion installed for Py-BOBYQA to work, and it is not installed by default.

1.2 Installation using pip

For easy installation, use pip:

```
$ pip install Py-BOBYQA
```

Note that if an older install of Py-BOBYQA is present on your system you can use:

```
$ pip install --upgrade Py-BOBYQA
```

to upgrade Py-BOBYQA to the latest version.

1.3 Manual installation

The source code for Py-BOBYQA is available on Github:

```
$ git clone https://github.com/numericalalgorithmsgroup/pybobyqa
$ cd pybobyqa
```

Py-BOBYQA is written in pure Python and requires no compilation. It can be installed using:

```
$ pip install .
```

To upgrade Py-BOBYQA to the latest version, navigate to the top-level directory (i.e. the one containing `setup.py`) and rerun the installation using `pip`, as above:

```
$ git pull
$ pip install .
```

1.4 Testing

If you installed Py-BOBYQA manually, you can test your installation using the pytest package:

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

1.5 Uninstallation

If Py-BOBYQA was installed using pip you can uninstall as follows:

```
$ pip uninstall Py-BOBYQA
```

If Py-BOBYQA was installed manually you have to remove the installed files by hand (located in your python site-packages directory).

CHAPTER

TWO

OVERVIEW

2.1 When to use Py-BOBYQA

Py-BOBYQA is designed to solve the nonlinear least-squares minimization problem (with optional bound and general convex constraints)

$$\begin{aligned} \min_{x\in\mathbb{R}^n} &\quad f(x) \\ \text{s.t.} &\quad a \leq x \leq b \\ &\quad x \in C := C_1 \cap \cdots \cap C_n, \quad \text{all } C_i \text{ convex} \end{aligned}$$

We call $f(x)$ the objective function.

Py-BOBYQA is a *derivative-free* optimization algorithm, which means it does not require the user to provide the derivatives of $f(x)$, nor does it attempt to estimate them internally (by using finite differencing, for instance).

There are two main situations when using a derivative-free algorithm (such as Py-BOBYQA) is preferable to a derivative-based algorithm (which is the vast majority of least-squares solvers).

If **the residuals are noisy**, then calculating or even estimating their derivatives may be impossible (or at least very inaccurate). By noisy, we mean that if we evaluate $f(x)$ multiple times at the same value of x, we get different results. This may happen when a Monte Carlo simulation is used, for instance, or $f(x)$ involves performing a physical experiment.

If **the residuals are expensive to evaluate**, then estimating derivatives (which requires n evaluations of $f(x)$ for every point of interest x) may be prohibitively expensive. Derivative-free methods are designed to solve the problem with the fewest number of evaluations of the objective as possible.

However, if you have provide (or a solver can estimate) derivatives of $f(x)$, then it is probably a good idea to use one of the many derivative-based solvers (such as one from the SciPy library).

2.2 Details of the Py-BOBYQA Algorithm

Py-BOBYQA is a type of *trust-region* method, a common category of optimization algorithms for nonconvex problems. Given a current estimate of the solution x_k, we compute a model which approximates the objective $m_k(s) \approx f(x_k+s)$ (for small steps s), and maintain a value $\Delta_k > 0$ (called the *trust region radius*) which measures the size of s for which the approximation is good.

At each step, we compute a trial step s_k designed to make our approximation $m_k(s)$ small (this task is called the *trust region subproblem*). We evaluate the objective at this new point, and if this provided a good decrease in the objective, we take the step ($x_{k+1} = x_k + s_k$), otherwise we stay put ($x_{k+1} = x_k$). Based on this information, we choose a new value Δ_{k+1}, and repeat the process.

In Py-BOBYQA, we construct our approximation $m_k(s)$ by interpolating a linear or quadratic approximation for $f(x)$ at several points close to x_k. To make sure our interpolated model is accurate, we need to regularly check that the points are well-spaced, and move them if they aren't (i.e. improve the geometry of our interpolation points).

Py-BOBYQA is a Python implementation of the BOBYQA solver by Powell [Powell2009]. More details about Py-BOBYQA algorithm are given in our paper [CFMR2018].

2.3 References

CHAPTER

THREE

USING PY-BOBYQA

This section describes the main interface to Py-BOBYQA and how to use it.

3.1 Nonlinear Minimization

Py-BOBYQA is designed to solve the local optimization problem

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad & f(x) \\ \text{s.t.} \quad & a \leq x \leq b \\ & x \in C := C_1 \cap \cdots \cap C_n, \quad \text{all } C_i \text{ convex} \end{aligned}$$

where the bound constraints $a \leq x \leq b$ and general convex constraints $x \in C$ are optional. The upper and lower bounds on the variables are non-relaxable (i.e. Py-BOBYQA will never ask to evaluate a point outside the bounds). The general convex constraints are also non-relaxable, but they may be slightly violated at some points from rounding errors. The objective function $f(x)$ is usually nonlinear and nonquadratic. If you know your objective is linear or quadratic, you should consider a solver designed for such functions (see here for details).

Py-BOBYQA iteratively constructs an interpolation-based model for the objective, and determines a step using a trust-region framework. For an in-depth technical description of the algorithm see the paper [CFMR2018], and for the global optimization heuristic, see [CRO2022]. For details about how Py-BOBYQA handles general convex constraints, see [R2024].

3.2 How to use Py-BOBYQA

The main interface to Py-BOBYQA is via the function `solve`

```
soln = pybobyqa.solve(objfun, x0)
```

The input `objfun` is a Python function which takes an input $x \in \mathbb{R}^n$ and returns the objective value $f(x) \in \mathbb{R}$. The input of `objfun` must be one-dimensional NumPy arrays (i.e. with `x.shape == (n,)`) and the output must be a single Float.

The input `x0` is the starting point for the solver, and (where possible) should be set to be the best available estimate of the true solution $x_{min} \in \mathbb{R}^n$. It should be specified as a one-dimensional NumPy array (i.e. with `x0.shape == (n,)`). As Py-BOBYQA is a local solver, providing different values for `x0` may cause it to return different solutions, with possibly different objective values.

The output of `pybobyqa.solve` is an object containing:

- `soln.x` - an estimate of the solution, $x_{min} \in \mathbb{R}^n$, a one-dimensional NumPy array.

- `soln.f` - the objective value at the calculated solution, $f(x_{min})$, a Float.
- `soln.gradient` - an estimate of the gradient vector of first derivatives of the objective, $g_i \approx \partial f(x_{min})/\partial x_i$, a NumPy array of length n.
- `soln.hessian` - an estimate of the Hessian matrix of second derivatives of the objective, $H_{i,j} \approx \partial^2 f(x_{min})/\partial x_i \partial x_j$, a NumPy array of size $n \times n$.
- `soln.nf` - the number of evaluations of `objfun` that the algorithm needed, an Integer.
- `soln.nx` - the number of points x at which `objfun` was evaluated, an Integer. This may be different to `soln.nf` if sample averaging is used.
- `soln.nruns` - the number of runs performed by Py-BOBYQA (more than 1 if using multiple restarts), an Integer.
- `soln.flag` - an exit flag, which can take one of several values (listed below), an Integer.
- `soln.msg` - a description of why the algorithm finished, a String.
- `soln.diagnostic_info` - a table of diagnostic information showing the progress of the solver, a Pandas DataFrame.

The possible values of `soln.flag` are defined by the following variables:

- `soln.EXIT_SUCCESS` - Py-BOBYQA terminated successfully (the objective value or trust region radius are sufficiently small).
- `soln.EXIT_MAXFUN_WARNING` - maximum allowed objective evaluations reached. This is the most likely return value when using multiple restarts.
- `soln.EXIT_SLOW_WARNING` - maximum number of slow iterations reached.
- `soln.EXIT_FALSE_SUCCESS_WARNING` - Py-BOBYQA reached the maximum number of restarts which decreased the objective, but to a worse value than was found in a previous run.
- `soln.EXIT_INPUT_ERROR` - error in the inputs.
- `soln.EXIT_TR_INCREASE_ERROR` - error occurred when solving the trust region subproblem.
- `soln.EXIT_LINALG_ERROR` - linear algebra error, e.g. the interpolation points produced a singular linear system.

These variables are defined in the `soln` object, so can be accessed with, for example

```
if soln.flag == soln.EXIT_SUCCESS:
    print("Success!")
```

3.3 Optional Arguments

The `solve` function has several optional arguments which the user may provide:

```
pybobyqa.solve(objfun, x0, args=(), bounds=None, projections=None,
               npt=None, rhobeg=None, rhoend=1e-8, maxfun=None,
               nsamples=None, user_params=None, objfun_has_noise=False,
               seek_global_minimum=False,
               scaling_within_bounds=False,
               do_logging=True, print_progress=False)
```

These arguments are:

- `args` - a tuple of extra arguments passed to the objective function.

- `bounds` - a tuple `(lower, upper)` with the vectors a and b of lower and upper bounds on x (default is $a_i = -10^{20}$ and $b_i = 10^{20}$). To set bounds for either `lower` or `upper`, but not both, pass a tuple `(lower, None)` or `(None, upper)`.
- `projections` - a list of functions defining the Euclidean projections for each general convex constraint C_i. Each element of the list `projections` is a function that takes an input vector x and returns the closest point to x that is in C_i. An example of using this is given below.
- `npt` - the number of interpolation points to use (default is $2n+1$ for a problem with `len(x0)=n` if `objfun_has_noise=False`, otherwise it is set to $(n+1)(n+2)/2$). Py-BOBYQA requires $n+1 \leq npt \leq (n+1)(n+2)/2$. Larger values are particularly useful for noisy problems.
- `rhobeg` - the initial value of the trust region radius (default is 0.1 if `scaling_within_bounds=True`, otherwise $0.1\max(\|x_0\|_\infty, 1)$).
- `rhoend` - minimum allowed value of trust region radius, which determines when a successful termination occurs (default is 10^{-8}).
- `maxfun` - the maximum number of objective evaluations the algorithm may request (default is $\min(100(n+1), 1000)$).
- `nsamples` - a Python function `nsamples(delta, rho, iter, nrestarts)` which returns the number of times to evaluate `objfun` at a given point. This is only applicable for objectives with stochastic noise, when averaging multiple evaluations at the same point produces a more accurate value. The input parameters are the trust region radius (`delta`), the lower bound on the trust region radius (`rho`), how many iterations the algorithm has been running for (`iter`), and how many restarts have been performed (`nrestarts`). Default is no averaging (i.e. `nsamples(delta, rho, iter, nrestarts)=1`).
- `user_params` - a Python dictionary `{'param1':  val1, 'param2':val2, ...}` of optional parameters. A full list of available options is given in the next section *Advanced Usage*.
- `objfun_has_noise` - a flag to indicate whether or not `objfun` has stochastic noise; i.e. will calling `objfun(x)` multiple times at the same value of `x` give different results? This is used to set some sensible default parameters (including using multiple restarts), all of which can be overridden by the values provided in `user_params`.
- `seek_global_minimum` - a flag to indicate whether to search for a global minimum, rather than a local minimum. This is used to set some sensible default parameters, all of which can be overridden by the values provided in `user_params`. If `True`, both upper and lower bounds must be set. Note that Py-BOBYQA only implements a heuristic method, so there are no guarantees it will find a global minimum. However, by using this flag, it is more likely to escape local minima if there are better values nearby. The method used is a multiple restart mechanism, where we repeatedly re-initialize Py-BOBYQA from the best point found so far, but where we use a larger trust reigon radius each time (note: this is different to more common multi-start approach to global optimization).
- `scaling_within_bounds` - a flag to indicate whether the algorithm should internally shift and scale the entries of `x` so that the bounds become $0 \leq x \leq 1$. This is useful is you are setting `bounds` and the bounds have different orders of magnitude. If `scaling_within_bounds=True`, the values of `rhobeg` and `rhoend` apply to the *shifted* variables.
- `do_logging` - a flag to indicate whether logging output should be produced. This is not automatically visible unless you use the Python logging module (see below for simple usage).
- `print_progress` - a flag to indicate whether to print a per-iteration progress log to terminal.

In general when using optimization software, it is good practice to scale your variables so that moving each by a given amount has approximately the same impact on the objective function. The `scaling_within_bounds` flag is designed to provide an easy way to achieve this, if you have set the bounds `lower` and `upper`.

3.4 A Simple Example

Suppose we wish to minimize the Rosenbrock test function:

$$\min_{(x_1,x_2)\in\mathbb{R}^2} \quad 100(x_2 - x_1^2)^2 + (1-x_1)^2$$

This function has exactly one local minimum $f(x_{min}) = 0$ at $x_{min} = (1,1)$. A commonly-used starting point for testing purposes is $x_0 = (-1.2, 1)$. The following script shows how to solve this problem using Py-BOBYQA:

```
# Py-BOBYQA example: minimize the Rosenbrock function
from __future__ import print_function
import numpy as np
import pybobyqa

# Define the objective function
def rosenbrock(x):
    return 100.0 * (x[1] - x[0] ** 2) ** 2 + (1.0 - x[0]) ** 2

# Define the starting point
x0 = np.array([-1.2, 1.0])

# Call Py-BOBYQA
soln = pybobyqa.solve(rosenbrock, x0)

# Display output
print(soln)
```

Note that Py-BOBYQA is a randomized algorithm: in its first phase, it builds an internal approximation to the objective function by sampling it along random directions. In the code above, we set NumPy's random seed for reproducibility over multiple runs, but this is not required. The output of this script, showing that Py-BOBYQA finds the correct solution, is

```
****** Py-BOBYQA Results ******
Solution xmin = [1. 1.]
Objective value f(xmin) = 1.013856052e-20
Needed 151 objective evaluations (at 151 points)
Approximate gradient = [ 2.35772499e-08 -1.07598803e-08]
Approximate Hessian = [[ 802.00799968 -400.04089119]
 [-400.04089119  199.99228723]]
Exit flag = 0
Success: rho has reached rhoend
******************************
```

This and all following problems can be found in the examples directory on the Py-BOBYQA Github page.

3.5 Adding Bounds and More Output

We can extend the above script to add constraints. To do this, we can add the lines

```
# Define bound constraints (lower <= x <= upper)
lower = np.array([-10.0, -10.0])
upper = np.array([0.9, 0.85])

# Call Py-BOBYQA (with bounds)
soln = pybobyqa.solve(rosenbrock, x0, bounds=(lower,upper))
```

Py-BOBYQA correctly finds the solution to the constrained problem:

```
****** Py-BOBYQA Results ******
Solution xmin = [0.9  0.81]
Objective value f(xmin) = 0.01
Needed 146 objective evaluations (at 146 points)
Approximate gradient = [-2.00000006e-01 -4.74578563e-09]
Approximate Hessian = [[ 649.66398033 -361.03094781]
 [-361.03094781  199.94223213]]
Exit flag = 0
Success: rho has reached rhoend
******************************
```

However, we also get a warning that our starting point was outside of the bounds:

```
RuntimeWarning: x0 above upper bound, adjusting
```

Py-BOBYQA automatically fixes this, and moves x_0 to a point within the bounds, in this case $x_0 = (-1.2, 0.85)$.

We can also get Py-BOBYQA to print out more detailed information about its progress using the logging module. To do this, we need to add the following lines:

```
import logging
logging.basicConfig(level=logging.INFO, format='%(message)s')

# ... (call pybobyqa.solve)
```

And we can now see each evaluation of `objfun`:

```
Function eval 1 at point 1 has f = 39.65 at x = [-1.2   0.85]
Initialising (coordinate directions)
Function eval 2 at point 2 has f = 14.337296 at x = [-1.08  0.85]
Function eval 3 at point 3 has f = 55.25 at x = [-1.2   0.73]
...
Function eval 145 at point 145 has f = 0.0100000013172792 at x = [0.89999999 0.
↪80999999]
Function eval 146 at point 146 has f = 0.00999999999999993 at x = [0.9  0.81]
Did a total of 1 run(s)
```

If we wanted to save this output to a file, we could replace the above call to `logging.basicConfig()` with

```
logging.basicConfig(filename="myfile.log", level=logging.INFO,
                    format='%(message)s', filemode='w')
```

If you have logging for some parts of your code and you want to deactivate all Py-BOBYQA logging, you can use the optional argument `do_logging=False` in `pybobyqa.solve()`.

An alternative option available is to get Py-BOBYQA to print to terminal progress information every iteration, by setting the optional argument `print_progress=True` in `pybobyqa.solve()`. If we do this for the above example, we get

```
 Run  Iter     Obj       Grad     Delta      rho     Evals
  1     1    1.43e+01  1.74e+02  1.20e-01  1.20e-01    5
  1     2    5.57e+00  1.20e+02  3.66e-01  1.20e-01    6
  1     3    5.57e+00  1.20e+02  6.00e-02  1.20e-02    6
...
  1    132   1.00e-02  2.00e-01  1.50e-08  1.00e-08   144
  1    133   1.00e-02  2.00e-01  1.50e-08  1.00e-08   145
```

3.6 Adding General Convex Constraints

We can also add more general convex constraints $x \in C := C_1 \cap \cdots \cap C_n$ to our problem, where each C_i is a convex set. To do this, we need to know the Euclidean projection operator for each C_i:

$$\mathrm{proj}_{C_i}(x) := \mathrm{argmin}_{y \in C_i} \|y - x\|_2^2.$$

i.e. given a point x, return the closest point to x in the set C_i. There are many examples of simple convex sets C_i for which this function has a known, simple form, such as:

- Bound constraints (but since Py-BOBYQA supports this directly, it is better to give these explicitly via the `bounds` input, as above)
- Euclidean ball constraints: $\|x - c\|_2 \leq r$
- Unit simplex: $x_i \geq 0$ and $\sum_{i=1}^n x_i \leq 1$
- Linear inequalities: $a^T x \geq b$

In Py-BOBYQA, set the input `projections` to be a list of projection functions, one per C_i. Internally, Py-BOBYQA computes the projection onto the intersection of these sets and the bound constraints using Dykstra's projection algorithm.

For the explicit expressions for the above projections, and more examples, see for example this online database or Section 6.4.6 of the textbook [B2017].

As an example, let's minimize the above Rosenbrock function with different bounds, and with a Euclidean ball constraint, namely $(x_1 - 0.5)^2 + (x_2 - 1)^2 \leq 0.25^2$.

To do this, we can run

```python
import numpy as np
import pybobyqa

# Define the objective function
def rosenbrock(x):
    return 100.0 * (x[1] - x[0] ** 2) ** 2 + (1.0 - x[0]) ** 2

# Define the starting point
x0 = np.array([-1.2, 1.0])
```

(continues on next page)

(continued from previous page)

```
# Define bound constraints (lower <= x <= upper)
lower = np.array([0.7, -2.0])
upper = np.array([1.0, 2.0])

# Define the ball constraint ||x-center|| <= radius, and its projection␣
↪operator
center = np.array([0.5, 1.0])
radius = 0.25
ball_proj = lambda x: center + (radius/max(np.linalg.norm(x-center), radius))␣
↪* (x-center)

# Call Py-BOBYQA (with bounds and projection operator)
# Note: it is better to provide bounds explicitly, instead of using the␣
↪corresponding
#       projection function
# Note: input 'projections' must be a list of projection functions
soln = pybobyqa.solve(rosenbrock, x0, bounds=(lower,upper), projections=[ball_
↪proj])

print(soln)
```

Py-BOBYQA correctly finds the solution to the constrained problem:

```
****** Py-BOBYQA Results ******
Solution xmin = [0.70424386 0.85583188]
Objective value f(xmin) = 13.03829114
Needed 25 objective evaluations (at 25 points)
Approximate gradient = [-101.9667031    71.97449424]
Approximate Hessian = [[ 253.11858775 -279.39193327]
 [-279.39193327  201.49725358]]
Exit flag = 0
Success: rho has reached rhoend
******************************
```

Just like for bound constraints, Py-BOBYQA will automatically ensure the starting point is feasible with respect to all constraints (bounds and general convex constraints).

3.7 Example: Noisy Objective Evaluation

As described in *Overview*, derivative-free algorithms such as Py-BOBYQA are particularly useful when `objfun` has noise. Let's modify the previous example to include random noise in our objective evaluation, and compare it to a derivative-based solver:

```
# Py-BOBYQA example: minimize the noisy Rosenbrock function
from __future__ import print_function
import numpy as np
import pybobyqa

# Define the objective function
def rosenbrock(x):
    return 100.0 * (x[1] - x[0] ** 2) ** 2 + (1.0 - x[0]) ** 2
```

(continues on next page)

(continued from previous page)

```
# Modified objective function: add 1% Gaussian noise
def rosenbrock_noisy(x):
    return rosenbrock(x) * (1.0 + 1e-2 * np.random.normal(size=(1,))[0])

# Define the starting point
x0 = np.array([-1.2, 1.0])

# Set random seed (for reproducibility)
np.random.seed(0)

print("Demonstrate noise in function evaluation:")
for i in range(5):
    print("objfun(x0) = %g" % rosenbrock_noisy(x0))
print("")

# Call Py-BOBYQA
soln = pybobyqa.solve(rosenbrock_noisy, x0)

# Display output
print(soln)

# Compare with a derivative-based solver
import scipy.optimize as opt
soln = opt.minimize(rosenbrock_noisy, x0)

print("")
print("** SciPy results **")
print("Solution xmin = %s" % str(soln.x))
print("Objective value f(xmin) = %.10g" % (soln.fun))
print("Needed %g objective evaluations" % soln.nfev)
print("Exit flag = %g" % soln.status)
print(soln.message)
```

The output of this is:

```
Demonstrate noise in function evaluation:
objfun(x0) = 24.6269
objfun(x0) = 24.2968
objfun(x0) = 24.4369
objfun(x0) = 24.7423
objfun(x0) = 24.6519

****** Py-BOBYQA Results ******
Solution xmin = [-1.04327395  1.09935385]
Objective value f(xmin) = 4.080030471
Needed 42 objective evaluations (at 42 points)
Approximate gradient = [-3786376.5065785   5876675.51763198]
Approximate Hessian = [[ 1.32881117e+14 -2.68241358e+14]
 [-2.68241358e+14  6.09785319e+14]]
Exit flag = 0
Success: rho has reached rhoend
```

(continues on next page)

(continued from previous page)

```
******************************


** SciPy results **
Solution xmin = [-1.20013817  0.99992915]
Objective value f(xmin) = 23.86371763
Needed 80 objective evaluations
Exit flag = 2
Desired error not necessarily achieved due to precision loss.
```

Although Py-BOBYQA does not find the true solution (and it cannot produce a good estimate of the objective gradient and Hessian), it still gives a reasonable decrease in the objective. However SciPy's derivative-based solver, which has no trouble solving the noise-free problem, is unable to make any progress.

As noted above, Py-BOBYQA has an input parameter `objfun_has_noise` to indicate if `objfun` has noise in it, which it does in this case. Therefore we can call Py-BOBYQA with

```
soln = pybobyqa.solve(rosenbrock_noisy, x0, objfun_has_noise=True)
```

This time, we find the true solution, and better estimates of the gradient and Hessian:

```
****** Py-BOBYQA Results ******
Solution xmin = [1. 1.]
Objective value f(xmin) = 1.237351799e-19
Needed 300 objective evaluations (at 300 points)
Did a total of 5 runs
Approximate gradient = [-2.17072738e-07  9.62304351e-08]
Approximate Hessian = [[ 809.56521044 -400.33737779]
 [-400.33737779  198.36487985]]
Exit flag = 1
Warning (max evals): Objective has been called MAXFUN times
******************************
```

3.8 Example: Global Optimization

The following example shows how to use the global optimization features of Py-BOBYQA. Here, we try to minimize the Freudenstein and Roth function (problem 2 in J.J. Moré, B.S. Garbow, B.S. and K.E. Hillstrom, Testing Unconstrained Optimization Software, *ACM Trans. Math. Software* 7:1 (1981), 17-41). This function has two local minima, one of which is global.

Note that Py-BOBYQA only implements a heuristic method, so there are no guarantees it will find a global minimum. However, by using the `seek_global_minimum` flag, it is more likely to escape local minima if there are better values nearby.

```
# Py-BOBYQA example: globally minimize the Freudenstein and Roth function
from __future__ import print_function
import numpy as np
import pybobyqa

# Define the objective function
# This function has a local minimum f = 48.98
```

(continues on next page)

(continued from previous page)

```
# at x = np.array([11.41, -0.8968])
# and a global minimum f = 0 at x = np.array([5.0, 4.0])
def freudenstein_roth(x):
    r1 = -13.0 + x[0] + ((5.0 - x[1]) * x[1] - 2.0) * x[1]
    r2 = -29.0 + x[0] + ((1.0 + x[1]) * x[1] - 14.0) * x[1]
    return r1 ** 2 + r2 ** 2

# Define the starting point
x0 = np.array([5.0, -20.0])

# Define bounds (required for global optimization)
lower = np.array([-30.0, -30.0])
upper = np.array([30.0, 30.0])

# Set random seed (for reproducibility)
np.random.seed(0)

print("First run - search for local minimum only")
print("")
soln = pybobyqa.solve(freudenstein_roth, x0, maxfun=500,
                      bounds=(lower, upper))
print(soln)

print("")
print("")

print("Second run - search for global minimum")
print("")
soln = pybobyqa.solve(freudenstein_roth, x0, maxfun=500,
                      bounds=(lower, upper),
                      seek_global_minimum=True)
print(soln)
```

The output of this is:

```
First run - search for local minimum only

****** Py-BOBYQA Results ******
Solution xmin = [11.41277902 -0.89680525]
Objective value f(xmin) = 48.98425368
Needed 143 objective evaluations (at 143 points)
Approximate gradient = [-1.64941396e-07  9.69795781e-07]
Approximate Hessian = [[   7.74717421 -104.51102613]
 [-104.51102613 1135.76500421]]
Exit flag = 0
Success: rho has reached rhoend
******************************



Second run - search for global minimum

```

(continues on next page)

(continued from previous page)

```
****** Py-BOBYQA Results ******
Solution xmin = [5. 4.]
Objective value f(xmin) = 3.659891409e-17
Needed 500 objective evaluations (at 500 points)
Did a total of 5 runs
Approximate gradient = [ 8.70038835e-10 -4.64918043e-07]
Approximate Hessian = [[   4.28883646   64.16836253]
 [  64.16836253 3722.93109385]]
Exit flag = 1
Warning (max evals): Objective has been called MAXFUN times
******************************
```

As we can see, the `seek_global_minimum` flag helped Py-BOBYQA escape the local minimum from the first run, and find the global minimum. More details are given in [CRO2022].

3.9 References

CHAPTER

FOUR

ADVANCED USAGE

This section describes different optional user parameters available in Py-BOBYQA.

In the last section (*Using Py-BOBYQA*), we introduced `pybobyqa.solve()`, which has the optional input `user_params`. This is a Python dictionary of user parameters. We will now go through the settings which can be changed in this way. More details are available in the paper [CFMR2018].

The default values, used if no override is given, in some cases vary depending on whether `objfun` has stochastic noise; that is, whether evaluating `objfun(x)` several times at the same `x` gives the same result or not. Whether or not this is the case is determined by the `objfun_has_noise` input to `pybobyqa.solve()` (and not by inspecting `objfun`, for instance). Similarly, the default values depend on the input flag `seek_global_minimum`, i.e. if a global minimum is desired.

4.1 General Algorithm Parameters

- `general.rounding_error_constant` - Internally, all interpolation points are stored with respect to a base point x_b; that is, we store $\{y_t - x_b\}$, which reduces the risk of roundoff errors. We shift x_b to x_k when $\|s_k\| \leq \text{const}\|x_k - x_b\|$, where 'const' is this parameter. Default is 0.1.
- `general.safety_step_thresh` - Threshold for when to call the safety step, $\|s_k\| \leq \gamma_S \rho_k$. Default is $\gamma_S = 0.5$.
- `general.check_objfun_for_overflow` - Whether to cap the value of $r_i(x)$ when they are large enough that an OverflowError will be encountered when trying to evaluate $f(x)$. Default is `True`.

4.2 Logging and Output

- `logging.n_to_print_whole_x_vector` - If printing all function evaluations to screen/log file, the maximum `len(x)` for which the full vector `x` should be printed also. Default is 6.
- `logging.save_diagnostic_info` - Flag so save diagnostic information at each iteration. Default is `False`.
- `logging.save_poisedness` - If saving diagnostic information, whether to include the Λ-poisedness of Y_k in the diagnostic information. This is the most computationally expensive piece of diagnostic information. Default is `True`.
- `logging.save_xk` - If saving diagnostic information, whether to include the full vector x_k. Default is `False`.

4.3 Initialization of Points

- `init.random_initial_directions` - Build the initial interpolation set using random directions (as opposed to coordinate directions). Default is `True`. Not used if general convex constraints provided.
- `init.random_directions_make_orthogonal` - If building initial interpolation set with random directions, whether or not these should be orthogonalized. Default is `True`. Not used if general convex constraints provided.
- `init.run_in_parallel` - If using random directions, whether or not to ask for all `objfun` to be evaluated at all points without any intermediate processing. Default is `False`. Not used if general convex constraints provided.

4.4 Trust Region Management

- `tr_radius.eta1` - Threshold for unsuccessful trust region iteration, η_1. Default is 0.1.
- `tr_radius.eta2` - Threshold for very successful trust region iteration, η_2. Default is 0.7.
- `tr_radius.gamma_dec` - Ratio to decrease Δ_k in unsuccessful iteration, γ_{dec}. Default is 0.5 for smooth problems or 0.98 for noisy problems (i.e. `objfun_has_noise = True`).
- `tr_radius.gamma_inc` - Ratio to increase Δ_k in very successful iterations, γ_{inc}. Default is 2.
- `tr_radius.gamma_inc_overline` - Ratio of $\|s_k\|$ to increase Δ_k by in very successful iterations, $\overline{\gamma}_{inc}$. Default is 4.
- `tr_radius.alpha1` - Ratio to decrease ρ_k by when it is reduced, α_1. Default is 0.1 for smooth problems or 0.9 for noisy problems (i.e. `objfun_has_noise = True`).
- `tr_radius.alpha2` - Ratio of ρ_k to decrease Δ_k by when ρ_k is reduced, α_2. Default is 0.5 for smooth problems or 0.95 for noisy problems (i.e. `objfun_has_noise = True`).

4.5 Termination on Small Objective Value

- `model.abs_tol` - Tolerance on $f(x_k)$; quit if $f(x_k)$ is below this value. Default is -10^{20}.

4.6 Termination on Slow Progress

- `slow.history_for_slow` - History used to determine whether the current iteration is 'slow'. Default is 5.
- `slow.thresh_for_slow` - Threshold for objective decrease used to determine whether the current iteration is 'slow'. Default is 10^{-8}.
- `slow.max_slow_iters` - Number of consecutive slow successful iterations before termination (or restart). Default is `20*len(x0)`.

4.7 Stochastic Noise Information

- noise.quit_on_noise_level - Flag to quit (or restart) if all $f(y_t)$ are within noise level of $f(x_k)$. Default is False for smooth problems or True for noisy problems.
- noise.scale_factor_for_quit - Factor of noise level to use in termination criterion. Default is 1.
- noise.multiplicative_noise_level - Multiplicative noise level in f. Can only specify one of multiplicative or additive noise levels. Default is None.
- noise.additive_noise_level - Additive noise level in f. Can only specify one of multiplicative or additive noise levels. Default is None.

4.8 Interpolation Management

- interpolation.precondition - whether or not to scale the interpolation linear system to improve conditioning. Default is True.
- interpolation.minimum_change_hessian - whether to solve the underdetermined quadratic interpolation problem by minimizing the Frobenius norm of the Hessian, or change in Hessian. Default is True.

4.9 Multiple Restarts

- restarts.use_restarts - Whether to do restarts when ρ_k reaches ρ_{end}, or (optionally) when all points are within noise level of $f(x_k)$. Default is False for smooth problems or True for noisy problems or when seeking a global minimum.
- restarts.max_unsuccessful_restarts - Maximum number of consecutive unsuccessful restarts allowed (i.e.~restarts which did not reduce the objective further). Default is 10.
- restarts.max_unsuccessful_restarts_total - Maximum number of total unsuccessful restarts allowed. Default is 20 when seeking a global minimum, otherwise it is maxfun (i.e.~not restricted).
- restarts.rhobeg_scale_after_unsuccessful_restart - Factor to increase ρ_{beg} by after unsuccessful restarts. Default is 1.1 when seeking a global minimum, otherwise it is 1.
- restarts.rhoend_scale - Factor to reduce ρ_{end} by with each restart. Default is 1.
- restarts.use_soft_restarts - Whether to use soft or hard restarts. Default is True.
- restarts.soft.num_geom_steps - For soft restarts, the number of points to move. Default is 3.
- restarts.soft.move_xk - For soft restarts, whether to preserve x_k, or move it to the best new point evaluated. Default is True.
- restarts.hard.use_old_fk - If using hard restarts, whether or not to recycle the objective value at the best iterate found when performing a restart. This saves one objective evaluation. Default is True.
- restarts.soft.max_fake_successful_steps - The maximum number of successful steps in a given run where the new (smaller) objective value is larger than the best value found in a previous run. Default is maxfun, the input to pybobyqa.solve().
- restarts.auto_detect - Whether or not to automatically determine when to restart. This is an extra condition, and restarts can still be triggered by small trust region radius, etc. Default is True.
- restarts.auto_detect.history - How many iterations of data on model changes and trust region radii to store. There are two criteria used: trust region radius decreases (no increases over the history, more decreases than

no changes), and change in model Jacobian (consistently increasing trend as measured by slope and correlation coefficient of line of best fit). Default is 30.

- `restarts.auto_detect.min_chg_model_slope` - Minimum rate of increase of $\log(\|g_k - g_{k-1}\|)$ and $\log(\|H_k - H_{k-1}\|_F)$ over the past iterations to cause a restart. Default is 0.015.
- `restarts.auto_detect.min_correl` - Minimum correlation of the data sets $(k, \log(\|g_k - g_{k-1}\|))$ and $(k, \log(\|H_k - H_{k-1}\|_F))$ required to cause a restart. Default is 0.1.

4.10 General Convex Constraints

- `projections.dykstra.d_tol` - termination tolerance for Dykstra's algorithm for computing the projection onto the intersection of all convex constraints. Default is 10^{-10}.
- `projections.dykstra.max_iters` - maximum iterations of Dykstra's algorithm for computing the projection onto the intersection of all convex constraints. Default is 100.
- `projections.feasible_tol` - tolerance for checking feasibility of initial points with respect to general convex constraints. Default is 10^{-10}.
- `projections.pgd_tol` - termination tolerance for trust-region and geometry-improving subproblems. Default is 10^{-8}.

4.11 References

CHAPTER

FIVE

DIAGNOSTIC INFORMATION

In *Using Py-BOBYQA*, we saw that the output of Py-BOBYQA returns a container which includes diagnostic information about the progress of the algorithm (`soln.diagnostic_info`). This object is a Pandas DataFrame, with one row per iteration of the algorithm. If Pandas is not available, it returns a dictionary where each key listed below has a list of values, one per iteration of the algorithm. In this section, we explain the meaning of each type of output (the columns of the DataFrame).

To save this information to a CSV file, use:

```
# Previously: define objfun and x0

# Turn on diagnostic information
user_params = {'logging.save_diagnostic_info': True}

# Call Py-BOBYQA
soln = pybobyqa.solve(objfun, x0, user_params=user_params)

# Save diagnostic info to CSV
soln.diagnostic_info.to_csv("myfile.csv")
```

Depending on exactly how Py-BOBYQA terminates, the last row of results may not be fully populated.

5.1 Current Iterate

- `xk` - Best point found so far (current iterate). This is only saved if `user_params['logging.save_xk'] = True`.
- `fk` - The value of f at the current iterate.

5.2 Trust Region

- `rho` - The lower bound on the trust region radius ρ_k.
- `delta` - The trust region radius Δ_k.
- `norm_sk` - The norm of the trust region step $\|s_k\|$.

5.3 Model Interpolation

- `npt` - The number of interpolation points.
- `interpolation_error` - The sum of squares of the interpolation errors from the interpolated model.
- `interpolation_condition_number` - The condition number of the matrix in the interpolation linear system.
- `interpolation_change_g_norm` - The norm of the change in model gradient at this iteration, $\|g_k - g_{k-1}\|$.
- `interpolation_change_H_norm` - The Frobenius norm of the change in model Hessian at this iteration, $\|H_k - H_{k-1}\|_F$.
- `poisedness` - The smallest value of Λ for which the current interpolation set Y_k is Λ-poised in the current trust region. This is the most expensive piece of information to compute, and is only computed if `user_params['logging.save_poisedness' = True`.
- `max_distance_xk` - The maximum distance from any interpolation point to the current iterate.
- `norm_gk` - The norm of the model gradient $\|g_k\|$.

5.4 Iteration Count

- `nruns` - The number of times the algorithm has been restarted.
- `nf` - The number of objective evaluations so far (see `soln.nf`)
- `nx` - The number of points at which the objective has been evaluated so far (see `soln.nx`)
- `nsamples` - The total number of objective evaluations used for all current interpolation points.
- `iter_this_run` - The number of iterations since the last restart.
- `iters_total` - The total number of iterations so far.

5.5 Algorithm Progress

- `iter_type` - A text description of what type of iteration we had (e.g. Successful, Safety, etc.)
- `ratio` - The ratio of actual to predicted objective reduction in the trust region step.
- `slow_iter` - Equal to 1 if the current iteration is successful but slow, 0 if is successful but not slow, and -1 if was not successful.

CHAPTER

SIX

VERSION HISTORY

This section lists the different versions of Py-BOBYQA and the updates between them.

6.1 Version 1.0 (6 Feb 2018)

- Initial release of Py-BOBYQA

6.2 Version 1.0.1 (20 Feb 2018)

- Minor bug fix to trust region subproblem solver (the output `crvmin` is calculated correctly) - this has minimal impact on the performance of Py-BOBYQA.

6.3 Version 1.0.2 (20 Jun 2018)

- Extra optional input `args` which passes through arguments for `objfun` (pull request from logangrado).
- Bug fixes: default parameters for reduced initialization cost regime, returning correct value from safety steps, retrieving dependencies during installation.

6.4 Version 1.1 (24 Dec 2018)

- Extra parameters to control the trust region radius over multiple restarts, designed for global optimization.
- New input flag `seek_global_minimum` to set sensible default parameters for global optimization. New example script to demonstrate this functionality.
- Bug fix: default trust region radius when scaling variables within bounds.

Initially released as version 1.1a0 on 17 Jul 2018.

6.5 Version 1.1.1 (5 Apr 2019)

- Link code to Zenodo, to create DOI - no changes to the Py-BOBYQA algorithm.

6.6 Version 1.2 (25 Feb 2020)

- Use deterministic initialisation by default (so it is no longer necessary to set a random seed for reproducibility of Py-BOBYQA results).
- Full model Hessian stored rather than just upper triangular part - this improves the runtime of Hessian-based operations.
- Faster trust-region and geometry subproblem solutions in Fortran using the trustregion package.
- Don't adjust starting point if it is close to the bounds (as long as it is feasible).
- Option to stop default logging behavior and/or enable per-iteration printing.
- Bugfix: correctly handle 1-sided bounds as inputs, avoid divide-by-zero warnings when auto-detecting restarts.

6.7 Version 1.3 (14 Apr 2021)

- Remove NumPy deprecation warnings from use of np.int and np.float

6.8 Version 1.4 (16 May 2023)

- Return diagnostic information as dictionary if Pandas not available (removes Pandas dependency)
- Handle Nan/Inf values in model gradient and Hessian by gracefully exiting trust-region subproblem
- Bugfix: automatically make model Hessian symmetric before trust-region subproblem with warning, instead of returning an error
- Bugfix: reset slow iteration counter when doing soft restarts

6.9 Version 1.4.1 (11 Apr 2024)

- Migrate package setup to pyproject.toml (required for Python version 3.12)
- Drop support for Python 2.7 and <=3.7 due to new setup process

6.10 Version 1.5 (9 Sep 2024)

- Added support for general convex constraints

CHAPTER

SEVEN

ACKNOWLEDGEMENTS

This software was initially developed under the supervision of Coralia Cartis, and was supported by the EPSRC Centre For Doctoral Training in Industrially Focused Mathematical Modelling (EP/L015803/1) in collaboration with the Numerical Algorithms Group. Development of Py-BOBYQA has also been supported by the Australian Research Council (DE240100006).

BIBLIOGRAPHY

[CFMR2018] Coralia Cartis, Jan Fiala, Benjamin Marteau and Lindon Roberts, Improving the Flexibility and Robustness of Model-Based Derivative-Free Optimization Solvers, *ACM Transactions on Mathematical Software*, 45:3 (2019), pp. 32:1-32:41 [preprint]

[Powell2009] Michael J. D. Powell, The BOBYQA algorithm for bound constrained optimization without derivatives, technical report DAMTP 2009/NA06, University of Cambridge, (2009).

[CFMR2018] Coralia Cartis, Jan Fiala, Benjamin Marteau and Lindon Roberts, Improving the Flexibility and Robustness of Model-Based Derivative-Free Optimization Solvers, *ACM Transactions on Mathematical Software*, 45:3 (2019), pp. 32:1-32:41 [preprint]

[CRO2022] Coralia Cartis, Lindon Roberts and Oliver Sheridan-Methven, Escaping local minima with derivative-free methods: a numerical investigation, *Optimization*, 71:8 (2022), pp. 2343-2373. [arXiv preprint: 1812.11343]

[R2024] Lindon Roberts, Model Construction for Convex-Constrained Derivative-Free Optimization, *arXiv preprint arXiv:2403.14960* (2024).

[B2017] Amir Beck, First-Order Methods in Optimization, SIAM (2017).

[CFMR2018] Coralia Cartis, Jan Fiala, Benjamin Marteau and Lindon Roberts, Improving the Flexibility and Robustness of Model-Based Derivative-Free Optimization Solvers, *ACM Transactions on Mathematical Software*, 45:3 (2019), pp. 32:1-32:41 [preprint]