Py-BOBYQA Documentation

Release 1.0.2

Lindon Roberts

20 June 2018

CONTENTS:

Release: 1.0.2

Date: 20 June 2018

Author: Lindon Roberts

Py-BOBYQA is a flexible package for finding local solutions to nonlinear, nonconvex minimization problems (with optional bound 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

$$\min_{x \in \mathbb{R}^n} \quad f(x)$$
$$\text{s.t.} \quad a \leq x \leq b$$

Full details of the Py-BOBYQA algorithm are given in our paper: C. Cartis, J. Fiala, B. Marteau and L. Roberts, Improving the Flexibility and Robustness of Model-Based Derivative-Free Optimization Solvers, technical report, University of Oxford, (2018).

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.

Py-BOBYQA is released under the GNU General Public License. Please contact NAG for alternative licensing.

ONE

INSTALLING PY-BOBYQA

1.1 Requirements

Py-BOBYQA requires the following software to be installed:

- Python 2.7 or Python 3

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

- NumPy 1.11 or higher
- SciPy 0.18 or higher
- Pandas 0.17 or higher

1.2 Installation using pip

For easy installation, use pip as root:

```
$ [sudo] pip install Py-BOBYQA
```

If you do not have root privileges or you want to install Py-BOBYQA for your private use, you can use:

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

which will install Py-BOBYQA in your home directory.

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

```
$ [sudo] 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:

```
$ [sudo] pip install .
```

If you do not have root privileges or you want to install Py-BOBYQA for your private use, you can use:

```
$ pip install --user .
```

instead.

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
$ [sudo] pip install .  # with admin privileges
```

1.4 Testing

If you installed Py-BOBYQA manually, you can test your installation by running:

```
$ python setup.py test
```

1.5 Uninstallation

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

```
$ [sudo] 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).

OVERVIEW

2.1 When to use Py-BOBYQA

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

$$\min_{x \in \mathbb{R}^n} \quad f(x)$$
$$\text{s.t.} \quad a \leq x \leq b$$

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

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

$$\min_{x \in \mathbb{R}^n} \quad f(x)$$
$$\text{s.t.} \quad a \le x \le b$$

where the bound constraints $a \le x \le b$ are optional. The objective function $f(x)$ is usually nonlinear and non-quadratic. 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]*.

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

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

3.3 Optional Arguments

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

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

- `npt` - the number of interpolation points to use (default is `2*len(x0)+1`). Py-BOBYQA requires `n+1 <= npt <= (n+1)*(n+2)/2` for a problem with `len(x0)=n`. Larger values are particularly useful for noisy problems.

- `rhobeg` - the initial value of the trust region radius (default is $0.1 \max(\|x_0\|_\infty, 1)$).

- `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`.

- `scaling_within_bounds` - a flag to indicate whether the algorithm should internally shift and scale the entries of `x` so that the bounds become $0 \le x \le 1$. 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.

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:

```python
# 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])

# Set random seed (for reproducibility)
np.random.seed(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) = 2.964036794e-19
Needed 213 objective evaluations (at 213 points)
Approximate gradient = [ -2.57280154e-08   1.26855793e-08]
Approximate Hessian = [[ 802.90904563 -400.46022134]
 [-400.46022134  200.23335154]]
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 134 objective evaluations (at 134 points)
Approximate gradient = [ -1.99999226e-01  -4.31078784e-07]
Approximate Hessian = [[ 649.6790222  -360.18361979]
 [-360.18361979  200.00205196]]
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:

```python
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 (random 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 133 at point 133 has f = 0.0100000000000165 at x = [ 0.9
↪    0.81000001]
Function eval 134 at point 134 has f = 0.0099999999999997 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

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

3.6 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:

```python
# 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

# 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) = %s" % str(rosenbrock_noisy(x0)))
print("")

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

```python
# 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) = 24.6269006677
objfun(x0) = 24.2968380444
objfun(x0) = 24.4368545922
objfun(x0) = 24.7422961542
objfun(x0) = 24.6519490336

****** Py-BOBYQA Results ******
Solution xmin = [-1.02866429  1.07341548]
Objective value f(xmin) = 4.033118937
Needed 36 objective evaluations (at 36 points)
Approximate gradient = [-6921247.2999239  -3051622.27188687]
Approximate Hessian = [[  1.98604897e+15   5.75929121e+14]
 [ 5.75929121e+14   7.89533101e+14]]
Exit flag = 0
Success: rho has reached rhoend
******************************


** SciPy results **
Solution xmin = [-1.2  1. ]
Objective value f(xmin) = 23.80943672
Needed 104 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

```python
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) = 3.418770987e-18
```

```
Needed 300 objective evaluations (at 300 points)
Did a total of 4 runs
Approximate gradient = [ -1.36175005e-08   2.12249758e-09]
Approximate Hessian = [[ 805.93202374 -394.16671315]
 [-394.16671315  192.99451721]]
Exit flag = 1
Warning (max evals): Objective has been called MAXFUN times
******************************
```

3.7 References

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).

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 $\|s_k\| \leq \mathrm{const}\|x_k - x_b\|$, 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`.

- `init.random_directions_make_orthogonal` - If building initial interpolation set with random directions, whether or not these should be orthogonalized. Default is `True`.

- `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`.

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.

- `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.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 References

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. 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:

```python
# 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.

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.

ACKNOWLEDGEMENTS

[CFMR2018]   3. Cartis, J. Fiala, B. Marteau and L. Roberts, Improving the Flexibility and Robustness of Model-Based Derivative-Free Optimization Solvers, technical report, University of Oxford, (2018).

[Powell2009]   13. (a)   i. Powell, The BOBYQA algorithm for bound constrained optimization without derivatives, technical report DAMTP 2009/NA06, University of Cambridge, (2009).

[CFMR2018]   3. Cartis, J. Fiala, B. Marteau and L. Roberts, Improving the Flexibility and Robustness of Model-Based Derivative-Free Optimization Solvers, technical report, University of Oxford, (2018).

[CFMR2018]   3. Cartis, J. Fiala, B. Marteau and L. Roberts, Improving the Flexibility and Robustness of Model-Based Derivative-Free Optimization Solvers, technical report, University of Oxford, (2018).