DFO-LS Documentation

Release 1.0.2

Lindon Roberts

20 June 2018

CONTENTS:

Release: 1.0.2

Date: 20 June 2018

Author: Lindon Roberts

DFO-LS is a flexible package for finding local solutions to nonlinear least-squares minimization problems (with optional bound constraints), without requiring any derivatives of the objective. DFO-LS stands for Derivative-Free Optimizer for Least-Squares.

That is, DFO-LS solves

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & f(x) := \sum_{i=1}^{m} r_i(x)^2 \\ \text{s.t.} \quad & a \leq x \leq b \end{aligned}$$

Full details of the DFO-LS 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). DFO-LS is a more flexible version of DFO-GN.

If you are interested in solving general optimization problems (without a least-squares structure), you may wish to try Py-BOBYQA, which has many of the same features as DFO-LS.

CHAPTER

ONE

INSTALLING DFO-LS

1.1 Requirements

DFO-LS requires the following software to be installed:

- Python 2.7 or Python 3 (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 1.11 or higher (http://www.numpy.org/)
- SciPy 0.18 or higher (http://www.scipy.org/)
- Pandas 0.17 or higher (http://pandas.pydata.org/)

1.2 Installation using pip

For easy installation, use *pip* (http://www.pip-installer.org/) as root:

```
.. code-block:: bash

   $ [sudo] pip install DFO-LS
```

or alternatively *easy_install*:

```
.. code-block:: bash

   $ [sudo] easy_install DFO-LS
```

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

```
.. code-block:: bash

   $ pip install --user DFO-LS
```

which will install DFO-LS in your home directory.

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

```
.. code-block:: bash

   $ [sudo] pip install --upgrade DFO-LS
```

to upgrade DFO-LS to the latest version.

1.3 Manual installation

Alternatively, you can download the source code from Github and unpack as follows:

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

DFO-LS 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 DFO-LS for your private use, you can use:

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

instead.

To upgrade DFO-LS 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 DFO-LS manually, you can test your installation by running:

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

Alternatively, the HTML documentation provides some simple examples of how to run DFO-LS.

1.5 Uninstallation

If DFO-LS was installed using *pip* you can uninstall as follows:

```
$ [sudo] pip uninstall DFO-LS
```

If DFO-LS 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 DFO-LS

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

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & f(x) := \sum_{i=1}^{m} r_i(x)^2 \\ \text{s.t.} \quad & a \leq x \leq b \end{aligned}$$

We call $f(x)$ the objective function and $r_i(x)$ the residual functions (or simply residuals).

DFO-LS is a *derivative-free* optimization algorithm, which means it does not require the user to provide the derivatives of $f(x)$ or $r_i(x)$, 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 DFO-LS) 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 $r_i(x)$ 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 $r_i(x)$ involves performing a physical experiment.

If **the residuals are expensive to evaluate**, then estimating derivatives (which requires n evaluations of each $r_i(x)$ 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 $r_i(x)$, 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 Parameter Fitting

A very common problem in many quantitative disciplines is fitting parameters to observed data. Typically, this means that we have developed a model for some proccess, which takes a vector of (known) inputs $\text{obs} \in \mathbb{R}^N$ and some model parameters $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$, and computes a (predicted) quantity of interest $y \in \mathbb{R}$:

$$y = \text{model}(\text{obs}, x)$$

For this model to be useful, we need to determine a suitable choice for the parameters x, which typically cannot be directly observed. A common way of doing this is to calibrate from observed relationships.

Suppose we have some observations of the input-to-output relationship. That is, we have data

$$(\text{obs}_1, y_1), \ldots, (\text{obs}_m, y_m)$$

Then, we try to find the parameters x which produce the best possible fit to these observations by minimizing the sum-of-squares of the prediction errors:

$$\min_{x \in \mathbb{R}^n} \quad f(x) := \sum_{i=1}^{m} (y_i - \text{model}(\text{obs}_i, x))^2$$

which is in the least-squares form required by DFO-LS.

As described above, DFO-LS is a particularly good choice for parameter fitting when the model has noise (e.g. Monte Carlo simulation) or is expensive to evaluate.

2.3 Solving Nonlinear Systems of Equations

Suppose we wish to solve the system of nonlinear equations: find $x \in \mathbb{R}^n$ satisfying

$$\begin{aligned} r_1(x) &= 0 \\ r_2(x) &= 0 \\ &\vdots \\ r_m(x) &= 0 \end{aligned}$$

Such problems can have no solutions, one solution, or many solutions (possibly infinitely many). Often, but certainly not always, the number of solutions depends on whether there are more equations or unknowns: if $m < n$ we say the system is underdetermined (and there are often multiple solutions), if $m = n$ we say the system is square (and there is often only one solution), and if $m > n$ we say the system is overdetermined (and there are often no solutions).

This is not always true – there is no solution to the underdetermined system when $m = 1$ and $n = 2$ and we choose $r_1(x) = \sin(x_1 + x_2) - 2$, for example. Similarly, if we take $n = 1$ and $r_i(x) = i(x - 1)(x - 2)$, we can make m as large as we like while keeping $x = 1$ and $x = 2$ as solutions (to the overdetermined system).

If no solution exists, it makes sense to instead search for an x which approximately satisfies each equation. A common way to do this is to minimize the sum-of-squares of the left-hand-sides:

$$\min_{x \in \mathbb{R}^n} \quad f(x) := \sum_{i=1}^{m} r_i(x)^2$$

which is the form required by DFO-LS.

If a solution does exist, then this formulation will also find this (where we will get $f = 0$ at the solution).

Which solution? DFO-LS, and most similar software, will only find one solution to a set of nonlinear equations. Which one it finds is very difficult to predict, and depends very strongly on the point where the solver is started from. Often it finds the closest solution, but there are no guarantees this will be the case. If you need to find all/multiple solutions for your problem, consider techniques such as deflation.

2.4 Details of the DFO-LS Algorithm

DFO-LS 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 DFO-LS, we construct our approximation $m_k(s)$ by interpolating a linear approximation for each residual $r_i(x)$ 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).

A complete description of the DFO-LS algorithm is given in our paper *[CFMR2018]*.

2.5 References

CHAPTER

THREE

USING DFO-LS

This section describes the main interface to DFO-LS and how to use it.

3.1 Nonlinear Least-Squares Minimization

DFO-LS is designed to solve the local optimization problem

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & f(x) := \sum_{i=1}^{m} r_i(x)^2 \\ \text{s.t.} \quad & a \leq x \leq b \end{aligned}$$

where the bound constraints $a \leq x \leq b$ are optional.

DFO-LS 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 DFO-LS

The main interface to DFO-LS is via the function `solve`

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

The input `objfun` is a Python function which takes an input $x \in \mathbb{R}^n$ and returns the vector of residuals $[r_1(x) \ \cdots \ r_m(x)] \in \mathbb{R}^m$. Both the input and output of `objfun` must be one-dimensional NumPy arrays (i.e. with `x.shape == (n,)` and `objfun(x).shape == (m,)`).

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 DFO-LS is a local solver, providing different values for `x0` may cause it to return different solutions, with possibly different objective values.

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

- `soln.x` - an estimate of the solution, $x_{min} \in \mathbb{R}^n$, a one-dimensional NumPy array.
- `soln.resid` - the vector of residuals at the calculated solution, $[r_1(x_{min}) \ \cdots \ r_m(x_{min})]$, a one-dimensional NumPy array.
- `soln.f` - the objective value at the calculated solution, $f(x_{min})$, a Float.
- `soln.jacobian` - an estimate of the Jacobian matrix of first derivatives of the residuals, $J_{i,j} \approx \partial r_i(x_{min})/\partial x_j$, a NumPy array of size $m \times n$.

- `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 DFO-LS (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` - DFO-LS 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` - DFO-LS 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:

```
dfols.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. This feature is new, and not yet avaiable in the PyPI version of DFO-LS; instead, use Python's built-in function `lambda`.
- `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 `len(x0)+1`). If using restarts, this is the number of points to use in the first run of the solver, before any restarts (and may be optionally increased via settings in `user_params`).
- `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 $0 \leq x \leq 1$. 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.

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 $x_{min} = (1, 1)$. We can write this as a least-squares problem as:

$$\min_{(x_1,x_2)\in\mathbb{R}^2} \quad [10(x_2 - x_1^2)]^2 + [1 - x_1]^2$$

A commonly-used starting point for testing purposes is $x_0 = (-1.2, 1)$. The following script shows how to solve this problem using DFO-LS:

```
# DFO-LS example: minimize the Rosenbrock function
from __future__ import print_function
import numpy as np
import dfols

# Define the objective function
def rosenbrock(x):
    return np.array([10.0 * (x[1] - x[0] ** 2), 1.0 - x[0]])

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

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

# Call DFO-LS
soln = dfols.solve(rosenbrock, x0)

# Display output
print(soln)
```

Note that DFO-LS 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 DFO-LS finds the correct solution, is

```
****** DFO-LS Results ******
Solution xmin = [ 1.  1.]
Residual vector = [ -2.22044605e-15   0.00000000e+00]
Objective value f(xmin) = 4.930380658e-30
Needed 36 objective evaluations (at 36 points)
Approximate Jacobian = [[ -1.98957443e+01   1.00000000e+01]
 [ -1.00000000e+00   8.37285083e-16]]
Exit flag = 0
Success: Objective is sufficiently small
****************************
```

This and all following problems can be found in the examples directory on the DFO-LS 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 DFO-LS (with bounds)
soln = dfols.solve(rosenbrock, x0, bounds=(lower, upper))
```

DFO-LS correctly finds the solution to the constrained problem:

```
****** DFO-LS Results ******
Solution xmin = [ 0.9   0.81]
Residual vector = [ 0.   0.1]
Objective value f(xmin) = 0.01
Needed 65 objective evaluations (at 65 points)
Approximate Jacobian = [[ -1.79999998e+01   9.99999990e+00]
 [ -9.99999998e-01  -2.53940698e-09]]
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
```

DFO-LS 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 DFO-LS 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 dfols.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 64 at point 64 has f = 0.0100000029949496 at x = [ 0.89999999 ␣
↪0.81      ]
Function eval 65 at point 65 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')
```

3.6 Example: Noisy Objective Evaluation

As described in *Overview*, derivative-free algorithms such as DFO-LS 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:

```
# DFO-LS example: minimize the noisy Rosenbrock function
from __future__ import print_function
import numpy as np
import dfols

# Define the objective function
def rosenbrock(x):
    return np.array([10.0 * (x[1] - x[0] ** 2), 1.0 - x[0]])

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

# 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 DFO-LS
soln = dfols.solve(rosenbrock_noisy, x0)

# Display output
print(soln)

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

print("")
print("** SciPy results **")
print("Solution xmin = %s" % str(soln.x))
print("Objective value f(xmin) = %.10g" % (2.0 * soln.cost))
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) = [-4.4776183   2.20880346]
objfun(x0) = [-4.44306447  2.24929965]
objfun(x0) = [-4.48217255  2.17849989]
objfun(x0) = [-4.44180389  2.19667014]
objfun(x0) = [-4.39545837  2.20903317]

****** DFO-LS Results ******
Solution xmin = [ 1.  1.]
Residual vector = [  3.51006670e-08   2.00158313e-10]
Objective value f(xmin) = 1.232096886e-15
Needed 46 objective evaluations (at 46 points)
Approximate Jacobian = [[ -2.04330578e+01   1.00296466e+01]
 [ -9.88260906e-01  -3.77364910e-03]]
Exit flag = 0
Success: Objective is sufficiently small
****************************


** SciPy results **
Solution xmin = [-1.2  1. ]
Objective value f(xmin) = 23.96809472
Needed 5 objective evaluations
Exit flag = 3
`xtol` termination condition is satisfied.
```

DFO-LS is able to find the solution with only 10 more function evaluations than in the noise-free case. However SciPy's derivative-based solver, which has no trouble solving the noise-free problem, is unable to make any progress.

As noted above, DFO-LS 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 DFO-LS with

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

Using this setting, we find the correct solution faster:

```
****** DFO-LS Results ******
Solution xmin = [ 1.  1.]
Residual vector = [ -5.80172077e-09   2.10781076e-09]
Objective value f(xmin) = 3.810283004e-17
Needed 29 objective evaluations (at 29 points)
Approximate Jacobian = [[ -1.96671666e+01   9.88784341e+00]
 [ -1.00451147e+00   1.43596001e-04]]
Exit flag = 0
Success: Objective is sufficiently small
****************************
```

3.7 Example: Parameter Estimation/Data Fitting

Next, we show a short example of using DFO-LS to solve a parameter estimation problem (taken from here). Given some observations (t_i, y_i), we wish to calibrate parameters $x = (x_1, x_2)$ in the exponential decay model

$$y(t) = x_1 \exp(x_2 t)$$

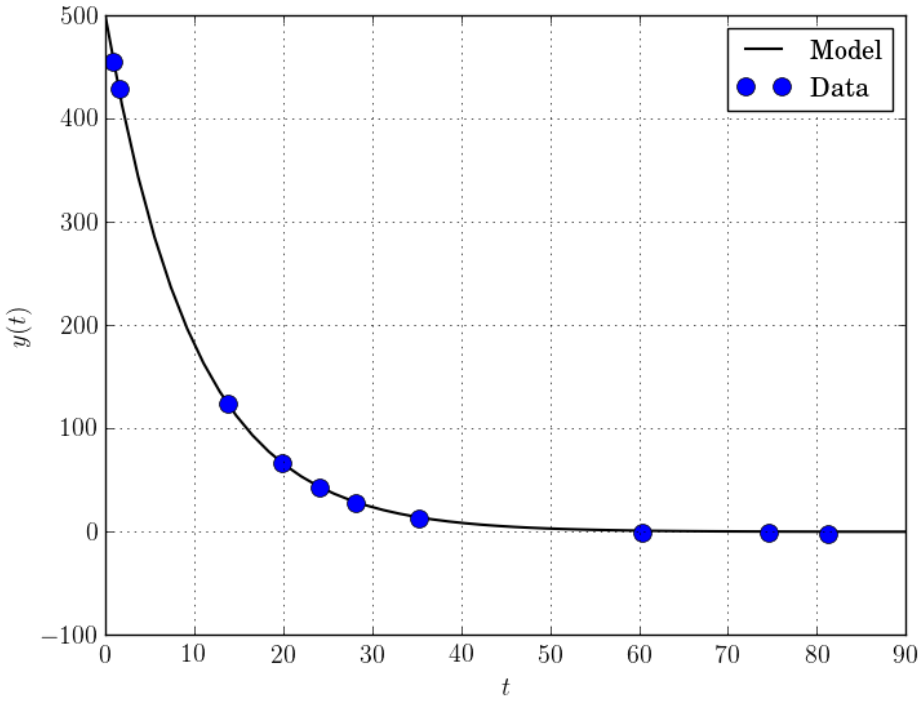
The code for this is:

```
# DFO-LS example: data fitting problem
# Originally from:
# https://uk.mathworks.com/help/optim/ug/lsqcurvefit.html
from __future__ import print_function
import numpy as np
import dfols

# Observations
tdata = np.array([0.9, 1.5, 13.8, 19.8, 24.1, 28.2, 35.2,
                  60.3, 74.6, 81.3])
ydata = np.array([455.2, 428.6, 124.1, 67.3, 43.2, 28.1, 13.1,
                  -0.4, -1.3, -1.5])

# Model is y(t) = x[0] * exp(x[1] * t)
def prediction_error(x):
    return ydata - x[0] * np.exp(x[1] * tdata)

# Define the starting point
x0 = np.array([100.0, -1.0])

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

# We expect exponential decay: set upper bound x[1] <= 0
upper = np.array([1e20, 0.0])

# Call DFO-LS
soln = dfols.solve(prediction_error, x0, bounds=(None, upper))

# Display output
print(soln)
```

The output of this is (noting that DFO-LS moves x_0 to be far away enough from the upper bound)

```
RuntimeWarning: x0 too close to upper bound, adjusting
****** DFO-LS Results ******
Solution xmin = [  4.98830860e+02  -1.01256863e-01]
Residual vector = [-0.18167084  0.06098401  0.76276294  0.11962349 -0.265898␣
↪  -0.59788818
 -1.02611899 -1.51235371 -1.56145452 -1.63266662]
Objective value f(xmin) = 9.504886892
Needed 99 objective evaluations (at 99 points)
Approximate Jacobian = [[ -9.12901557e-01  -4.09843510e+02]
 [ -8.59085471e-01  -6.42808522e+02]
 [ -2.47253894e-01  -1.70205399e+03]
 [ -1.34675403e-01  -1.33017159e+03]
 [ -8.71359818e-02  -1.04752827e+03]
 [ -5.75305576e-02  -8.09280563e+02]
 [ -2.83185322e-02  -4.97239478e+02]
 [ -2.22993603e-03  -6.70749492e+01]
 [ -5.24135530e-04  -1.95045149e+01]
 [ -2.65977795e-04  -1.07858009e+01]]
Exit flag = 0
Success: rho has reached rhoend
*****************************
```

This produces a good fit to the observations.



To generate this plot, run:

```
# Plot calibrated model vs. observations
ts = np.linspace(0.0, 90.0)
ys = soln.x[0] * np.exp(soln.x[1] * ts)

import matplotlib.pyplot as plt
plt.figure(1)
ax = plt.gca()  # current axes
```

```
ax.plot(ts, ys, 'k-', label='Model')
ax.plot(tdata, ydata, 'bo', label='Data')
ax.set_xlabel('t')
ax.set_ylabel('y(t)')
ax.legend(loc='upper right')
ax.grid()
plt.show()
```

3.8 Example: Solving a Nonlinear System of Equations

Lastly, we give an example of using DFO-LS to solve a nonlinear system of equations (taken from here). We wish to solve the following set of equations

$$x_1 + x_2 - x_1 x_2 + 2 = 0,$$
$$x_1 \exp(-x_2) - 1 = 0.$$

The code for this is:

```
# DFO-LS example: Solving a nonlinear system of equations
# Originally from:
# http://support.sas.com/documentation/cdl/en/imlug/66112/HTML/default/
↪viewer.htm#imlug_genstatexpls_sect004.htm

from __future__ import print_function
from math import exp
import numpy as np
import dfols

# Want to solve:
#   x1 + x2 - x1*x2 + 2 = 0
#   x1 * exp(-x2) - 1   = 0
def nonlinear_system(x):
    return np.array([x[0] + x[1] - x[0]*x[1] + 2,
                     x[0] * exp(-x[1]) - 1.0])

# Warning: if there are multiple solutions, which one
#          DFO-LS returns will likely depend on x0!
x0 = np.array([0.1, -2.0])

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

# Call DFO-LS
soln = dfols.solve(nonlinear_system, x0)

# Display output
print(soln)
```

The output of this is

```
****** DFO-LS Results ******
Solution xmin = [ 0.09777309 -2.32510588]
Residual vector = [  2.89990254e-13   3.31557004e-12]
Objective value f(xmin) = 1.107709904e-23
Needed 18 objective evaluations (at 18 points)
```

```
Approximate Jacobian = [[  3.32510429   0.90222738]
 [ 10.22774647  -0.9999939 ]]
Exit flag = 0
Success: Objective is sufficiently small
****************************
```

Here, we see that both entries of the residual vector are very small, so both equations have been solved to high accuracy.

3.9 References

CHAPTER

FOUR

ADVANCED USAGE

This section describes different optional user parameters available in DFO-LS.

In the last section (*Using DFO-LS*), we introduced `dfols.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 `dfols.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 $\|s_k\| \leq \text{const}\|x_k - x_b\|$, 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`.
- `logging.save_rk` - If saving diagnostic information, whether to include the full vector $[r_1(x_k) \cdots r_m(x_k)]$. The value $f(x_k)$ is always included. 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^{-12}.
- `model.rel_tol` - Relative tolerance on $f(x_k)$; quit if $f(x_k)/f(x_0)$ 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^{-4}.
- `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`.

4.9 Regression Model Management

- `regression.num_extra_steps` - In successful iterations, the number of extra points (other than accepting the trust region step) to move, useful when $|Y_k| > n + 1$ (n is `len(x0)`). Default is 0.
- `regression.increase_num_extra_steps_with_restart` - The amount to increase `regression.num_extra_steps` by with each restarts, for instance if increasing the number of points with each restart. Default is 0.
- `regression.momentum_extra_steps` - If moving extra points in successful iterations, whether to use the 'momentum' method. If not, uses geometry-improving steps. Default is `False`.

4.10 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.increase_npt` - Whether to increase $|Y_k|$ with each restart. Default is `False`.
- `restarts.increase_npt_amt` - Amount to increase $|Y_k|$ by with each restart. Default is 1.
- `restarts.hard.increase_ndirs_initial_amt` - Amount to increase `growing.ndirs_initial` by with each hard restart. To avoid a growing phase, it is best to set it to the same value as `restarts.increase_npt_amt`. Default is 1.
- `restarts.hard.use_old_rk` - 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.max_npt` - Maximum allowed value of $|Y_k|$, useful if increasing with each restart. Default is `npt`, the input parameter to `dfols.solve()`.

- `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 `dfols.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_chgJ_slope` - Minimum rate of increase of $\log(\|J_k - J_{k-1}\|_F)$ over the past iterations to cause a restart. Default is 0.015.
- `restarts.auto_detect.min_correl` - Minimum correlation of the data set $(k, \log(\|J_k - J_{k-1}\|_F))$ required to cause a restart. Default is 0.1.

4.11 Dynamically Growing Initial Set

- `growing.ndirs_initial` - Number of initial points to add (excluding x_k). Default is `npt-1`.
- `growing.num_new_dirns_each_iter` - Number of new search directions to add with each iteration where we do not have a full set of search directions. Default is 1.
- `growing.delta_scale_new_dirns` - When adding new search directions, the length of the step as a multiple of Δ_k. Default is 1, but if setting `growing.perturb_trust_region_step=True` should be made smaller (e.g. 0.1).
- `growing.do_geom_steps` - While still growing the initial set, whether to do geometry-improving steps in the trust region algorithm, as per the usual algorithm. Default is `False`.
- `growing.safety.do_safety_step` - While still growing the initial set, whether to perform safety steps, or the regular trust region steps. Default is `True`.
- `growing.safety.reduce_delta` - While still growing the initial set, whether to reduce Δ_k in safety steps. Default is `False`.
- `growing.safety.full_geom_step` - While still growing the initial set, whether to do a full geometry-improving step within safety steps (the same as the post-growing phase of the algorithm). Since this involves reducing Δ_k, cannot be `True` if `growing.safety.reduce_delta` is `True`. Default is `False`.
- `growing.full_rank.use_full_rank_interp` - Whether to perturb the interpolated J_k to make it full rank, allowing the trust region step to include components in the full search space. If `True`, setting `growing.num_new_dirns_each_iter` to 0 is recommended. Default is `False`.
- `growing.full_rank.scale_factor` - Magnitude of extra components added to J_k. Default is 10^{-2}.
- `growing.full_rank.svd_scale_factor` - Floor singular values of J_k at this factor of the last nonzero value. Default is 1.
- `growing.full_rank.min_sing_val` - Absolute floor on singular values of J_k. Default is 10^{-6}.
- `growing.full_rank.svd_max_jac_cond` - Cap on condition number of J_k after applying floors to singular values (effectively another floor on the smallest singular value, since the largest singular value is fixed). Default is 10^8.
- `growing.reset_delta` - Whether or not to reset trust region radius Δ_k to its initial value at the end of the growing phase. Default is `False`.

- `growing.reset_rho` - Whether or not to reset trust region radius lower bound ρ_k to its initial value at the end of the growing phase. Default is `False`.
- `growing.gamma_dec` - Trust region decrease parameter during the growing phase. Default is `tr_radius.gamma_dec`.
- `growing.perturb_trust_region_step` - Whether to perturb the trust region step by an orthogonal direction not yet searched. This is an alternative to `growing.full_rank.use_full_rank_interp`. Default is `False`.

4.12 References

CHAPTER

FIVE

DIAGNOSTIC INFORMATION

In *Using DFO-LS*, we saw that the output of DFO-LS 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:

```
# Previously: define objfun and x0

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

# Call DFO-LS
soln = dfols.solve(objfun, x0, user_params=user_params)

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

Depending on exactly how DFO-LS 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`.
- `rk` - The vector of residuals at the current iterate. This is only saved if `user_params['logging.save_rk'] = 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_J_norm` - The Frobenius norm of the change in Jacobian at this iteration, $\|J_k - J_{k-1}\|_F$.
- `interpolation_total_residual` - The total residual from the interpolation optimization problem.
- `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 DFO-LS and the updates between them.

6.1 Version 1.0 (6 Feb 2018)

- Initial release of DFO-LS

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 DFO-LS.

6.3 Version 1.0.2 (20 Jun 2018)

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

CHAPTER

SEVEN

ACKNOWLEDGEMENTS

This software was 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.

BIBLIOGRAPHY

[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).

[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).