

---

**DFBGN**

***Release 0.1***

**Lindon Roberts**

**23 October 2020**



**CONTENTS:**

<b>1</b>	<b>Installing DFBGN</b>	<b>3</b>
1.1	Requirements . . . . .	3
1.2	Installation using pip . . . . .	3
1.3	Manual installation . . . . .	4
1.4	Testing . . . . .	4
1.5	Uninstallation . . . . .	4
<b>2</b>	<b>Using DFBGN</b>	<b>5</b>
2.1	Nonlinear Least-Squares Minimization . . . . .	5
2.2	How to use DFBGN . . . . .	5
2.3	A Simple Example . . . . .	6
2.4	More Output . . . . .	7
2.5	Example: Noisy Objective Evaluation . . . . .	8
2.6	Example: Solving a Nonlinear System of Equations . . . . .	9
<b>3</b>	<b>Acknowledgements</b>	<b>11</b>



**Release:** 0.1

**Date:** 23 October 2020

**Author:** [Lindon Roberts](#)

DFBGN is a package for finding local solutions to large-scale nonlinear least-squares minimization problems, without requiring any derivatives of the objective. DFBGN stands for Derivative-Free Block Gauss-Newton.

That is, DFBGN solves

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

If you wish to solve small-scale least-squares problems, you may wish to try [DFO-LS](#). If you are interested in solving general optimization problems (without a least-squares structure), you may wish to try [Py-BOBYQA](#).

DFBGN is released under the GNU General Public License. Please [contact NAG](#) for alternative licensing.



## INSTALLING DFBGN

### 1.1 Requirements

DFBGN 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* as root:

```
$ [sudo] pip install dfbgn
```

or alternatively *easy\_install*:

```
$ [sudo] easy_install dfbgn
```

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

```
$ pip install --user dfbgn
```

which will install DFBGN in your home directory.

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

```
$ [sudo] pip install --upgrade dfbgn
```

to upgrade DFBGN 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/dfbgn
$ cd dfbgn
```

DFBGN 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 DFBGN for your private use, you can use:

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

instead.

To upgrade DFBGN 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 DFBGN manually, you can test your installation by running:

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

Alternatively, the HTML documentation provides some simple examples of how to run DFBGN.

## 1.5 Uninstallation

If DFBGN was installed using `pip` you can uninstall as follows:

```
$ [sudo] pip uninstall dfbgn
```

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



## USING DFBGN

This section describes the main interface to DFBGN and how to use it.

### 2.1 Nonlinear Least-Squares Minimization

DFBGN is designed to solve the local optimization problem

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

DFBGN iteratively constructs an interpolation-based model for the objective, and determines a step using a trust-region framework.

### 2.2 How to use DFBGN

The main interface to DFBGN is via the function `solve`

```
soln = dfbgn.solve(objfun, x0, fixed_block=fixed_block)
```

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

The input `fixed_block` is the size of the exploration space. It should be an integer from 1 to `len(x0)` inclusive, set based on how fast you want the internal linear algebra calculations to be (smaller values are faster).

The output of `dfbgn.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.nf` - the number of evaluations of `objfun` that the algorithm needed, 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` - DFBGN 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` - DFBGN 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!")
```

## 2.3 A Simple Example

Suppose we wish to minimize the [Rosenbrock test function](#):

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

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:

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

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

```
# DFBGN example: minimize the Rosenbrock function
from __future__ import print_function
import numpy as np
import dfbgn

# 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])

# DFBGN is a randomized algorithm - set random seed for reproducibility
np.random.seed(0)
```

(continues on next page)

(continued from previous page)

```
# Call DFBGN
soln = dfbgn.solve(rosenbrock, x0, fixed_block=2)

# Display output
print(soln)
```

Note that DFBGN is a randomized algorithm: the subspace it searches is randomly generated. The output of this script, showing that DFBGN finds the correct solution, is

```
***** DFBGN Results *****
Solution xmin = [ 1.          0.99999998]
Residual vector = [ -1.61462722e-07  0.00000000e+00]
Objective value f(xmin) = 2.607021062e-14
Needed 72 objective evaluations
No approximate Jacobian available
Exit flag = 0
Success: Objective is sufficiently small
*****
```

This and all following problems can be found in the [examples](#) directory on the DFBGN Github page.

## 2.4 More Output

We can get DFBGN 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 dfbgn.solve)
```

And we can now see each evaluation of objfun:

```
Function eval 1 has f = 24.2 at x = [-1.2  1. ]
Function eval 2 has f = 63.2346372977649 at x = [-1.30493146  0.94178154]
Function eval 3 has f = 27.9653746738959 at x = [-1.25821846  1.10493146]
Function eval 4 has f = 6.33451236346909 at x = [-1.08861669  1.04465151]
...
Function eval 70 has f = 1.99643713755605e-12 at x = [ 1.          1.
↪00000014]
Function eval 71 has f = 110.765405382932 at x = [ 0.45748543 -0.84175933]
Function eval 72 has f = 2.60702106219341e-14 at x = [ 1.          0.
↪99999998]
```

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')
```

## 2.5 Example: Noisy Objective Evaluation

As described in info, derivative-free algorithms such as DFBGN 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:

```
# DFBGN example: minimize the noisy Rosenbrock function
from __future__ import print_function
import numpy as np
import dfbgn

# 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 DFBGN
soln = dfbgn.solve(rosenbrock_noisy, x0, fixed_block=2)

# 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]
```

(continues on next page)

(continued from previous page)

```

***** DFBGN Results *****
Solution xmin = [ 1.          0.99999994]
Residual vector = [ -6.31017296e-07   5.73947373e-10]
Objective value f(xmin) = 3.981831569e-13
Needed 82 objective evaluations
No approximate Jacobian available
Exit flag = 0
Success: Objective is sufficiently small
*****

** SciPy results **
Solution xmin = [-1.19999679  1.00000624]
Objective value f(xmin) = 23.47462704
Needed 8 objective evaluations
Exit flag = 3
`xtol` termination condition is satisfied.

```

DFBGN is able to find the solution with 10 more function evaluations as 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.

## 2.6 Example: Solving a Nonlinear System of Equations

Lastly, we give an example of using DFBGN to solve a nonlinear system of equations (taken from [here](#)). We wish to solve the following set of equations

$$\begin{aligned}x_1 + x_2 - x_1x_2 + 2 &= 0, \\x_1 \exp(-x_2) - 1 &= 0.\end{aligned}$$

The code for this is:

```

# DFBGN 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 dfbgn

# 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
#           DFBGN returns will likely depend on x0!
x0 = np.array([0.1, -2.0])

# DFBGN is a randomized algorithm - set random seed for reproducibility
np.random.seed(0)

```

(continues on next page)

(continued from previous page)

```
# Call DFBGN
soln = dfbgn.solve(nonlinear_system, x0, fixed_block=2)

# Display output
print(soln)
```

The output of this is

```
***** DFBGN Results *****
Solution xmin = [ 0.09777311 -2.32510592]
Residual vector = [ 2.38996951e-08 2.23316848e-07]
Objective value f(xmin) = 5.044160988e-14
Needed 18 objective evaluations
No approximate Jacobian available
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.

## ACKNOWLEDGEMENTS

This software was developed under the supervision of [Coralie 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](#).