Optimithon Documentation

Release 0.2.0

REBE, Government of Saskatchewan

Jun 25, 2018

Contents:

CHAPTER 1

Introduction

This code aims to provide flexible and extendable implementation of various standard and experimental optimization methods. The development is gradual and priority will be given to those methods that seem to be more exciting to implement (based on personal interests).

This project was mainly motivated by my lake of knowledge in numerical optimization, otherwise there are excellent (open source) tools for almost every task that a machine can do. This occurred while I was working on an other scientific project on global optimization irene.rtfd.io, which tends to provide a tool to find a lower bound for the global minimum of a function when the function and constraints are (algebraically) well presented.

1.1 Requirements

Dependencies are minimal as the code is mainly written in python:

- NumPy.
- Numdifftools (optional).

1.2 Download

Optimithon can be obtained from https://github.com/mghasemi/optimithon.

1.3 Installation

To install *Optimithon*, run the following in terminal:

```
sudo python setup.py install
```

1.4 Documentation

The documentation is produced by Sphinx and is intended to cover code usage as well as a bit of theory to explain each method briefly. For more details refer to the documentation at optimithon.rtfd.io.

1.5 License

This code is distributed under MIT license:

1.5.1 MIT License

CHAPTER 2

Optimization Problem

A typical optimization problem can be formulated as

$$\begin{cases} \min_x & f(x) \\ \text{subject to} & \\ & g_i(x) \geq 0 \quad i = 1, \ldots, m \\ & \text{and} \\ & h_j(x) = 0 \quad j = 1, \ldots, k \end{cases} \tag{2.1}$$

The problem (2.1) is called a constrained optimization. If the constraints $g_i(x) \geq 0$ and $h_j(x) = 0$ are absent then the (2.1) is called an unconstrained optimization problem.

This package attempts to implement various methods to solve the unconstrained optimization problem. Then by employing the barrier functions method, the resulted code for unconstrained problem is modified to solve the general form of (2.1).

The user interface for both unconstrained and constrained optimization problems is the same, but some parameters are ignored for unconstrained problems. A minimal code to solve (2.1) would look like the following:

```
from Optimithon import Base, QuasiNewton # import the essentials
f = # definition of the objective function
ineqs = [g_i for i in range(m)] # the list of inequality constraints: g_i >= 0.
eqs = [h_j for j in range(k)] # the list of equality constraints: h_j == 0.
OPTIM = Base(f, # the objective function (mandatory)
         ineq=ineqs, # inequality constraints
         eq = eqs, # equality constraints
         x0=x0, # an initial point, a numpy array
         )
OPTIM() # run the optimization procedure
print(OPTIM.solution) # show the outcome
```

CHAPTER 3

Iterative Optimization Methods

The iterative (unconstrained) optimization methods are the most popular optimization methods to approximate a (local) minimum of a given function. Generally, an iterative method uses a patter like the following:

- With the objective function f and an initial guess for the minimum $x = x_0$:
- **Repeat:**
  - Find a *descent direction* p_n at point x_n,
  - Find a positive value α such that $f(x_n + \alpha p_n)$ is a reasonable decrease compare to $f(x_n)$,
  - Update $x_{n+1} = x_n + \alpha p_n$,
- **Until** a termination criterion is satisfied.
- **Return** x_n as an approximation for a local minimum of f.

Variations of the iterative methods focus on finding a suitable *descent direction* p_n, as well as suitable value for α and a *termination strategy*.

To determine a suitable direction some methods use first or second (or even higher orders) derivatives of f. Those methods that do not use derivatives are called *derivative free* methods.

The derivative base methods are implemented in `QuasiNewton` module.

CHAPTER 4

Derivative Based Methods

Derivative Based Methods are a class of iterative optimization methods that calculate a descent direction using gradient and/or hessian of the objective function.

For more details on the methods introduced in this chapter refer to *[JNSJW]*.

4.1 Descent Direction

The following is the list of implemented methods to find a descent direction.

4.1.1 Gradient descent direction

This method chooses the backward gradient direction to achieve the direction which results in steepest decrease in the values of the objective function:

$$p_n = \nabla f(x_n)$$

4.1.2 Newton Conjugate Gradient method

The Newton conjugate gradient method uses the followin descent direction:

$$p_{n+1} = \nabla^2 f(x_n)^{-1} \nabla f(x_n),$$

where $\nabla^2 f(x_n)$ is the Hessian of f at x_n.

4.1.3 Fletcher-Reeves method

The direction suggested by R. Fletcher and C. M. Reeves in 1964. Let $p_0 = -\nabla f(x_0)$ and

$$p_n = \frac{\nabla f(x_n)^T \nabla f(x_n)}{\|\nabla f(x_{n-1})\|^2} p_{n-1} - \nabla f(x_n).$$

4.1.4 Polak–Ribiere method

Suggested by E. Polak and G. Ribiere in 1969. Let $p_0 = -\nabla f(x_0)$ and

$$p_n = \frac{\nabla f(x_n)^T(\nabla f(x_n) - \nabla f(x_{n-1}))}{\|\nabla f(x_{n-1})\|^2} p_{n-1} - \nabla f(x_n).$$

4.1.5 Hestenes-Stiefel method

Suggested by M. R. Hestenes and E. Stiefel in 1953. Let $p_0 = -\nabla f(x_0)$ and

$$p_n = \frac{\nabla f(x_n)^T(\nabla f(x_n) - \nabla f(x_{n-1}))}{(\nabla f(x_n) - \nabla f(x_{n-1}))^T p_{n-1}} p_{n-1} - \nabla f(x_n).$$

4.1.6 Dai-Yuan method

Suggested by Y.-H. Dai and Y. Yuan in 1999. Let $p_0 = -\nabla f(x_0)$ and

$$p_n = \frac{\|\nabla f(x_n)\|^2}{(\nabla f(x_n) - \nabla f(x_{n-1}))p_{n-1}} p_{n-1} - \nabla f(x_n).$$

4.1.7 Davidon-Fletcher-Powell method

Let $H_0 = \nabla^2 f(x_0)^{-1}$ and

$$\begin{aligned} H_n &= H_{n-1} + \frac{(x_n - x_{n-1})^T (x_n - x_{n-1})}{(x_n - x_{n-1})^T (\nabla f(x_n) - \nabla f(x_{n-1}))} \\ &- \frac{H_{n-1}(\nabla f(x_n) - \nabla f(x_{n-1}))(\nabla f(x_n) - \nabla f(x_{n-1}))^T H_{n-1}}{(\nabla f(x_n) - \nabla f(x_{n-1}))^T H_{n-1} (\nabla f(x_n) - \nabla f(x_{n-1}))}, \end{aligned}$$

then $p_n = -H_n \nabla f(x_n)$.

4.1.8 Broyden-Fletcher-Goldfarb-Shanno method

Let $H_0 = \nabla^2 f(x_0)^{-1}$ and

$$\begin{aligned} H_n &= \left(I - \frac{(x_n - x_{n-1})(\nabla f(x_n) - \nabla f(x_{n-1}))^T}{(\nabla f(x_n) - \nabla f(x_{n-1}))^T (x_n - x_{n-1})}\right) \\ &\times H_{n-1} \\ &\times \left(I - \frac{(\nabla f(x_n) - \nabla f(x_{n-1}))(x_n - x_{n-1})^T}{(\nabla f(x_n) - \nabla f(x_{n-1}))^T (x_n - x_{n-1})}\right) \\ &+ \frac{(x_n - x_{n-1})(x_n - x_{n-1})^T}{(\nabla f(x_n) - \nabla f(x_{n-1}))^T (x_n - x_{n-1})}, \end{aligned}$$

then $p_n = -H_n \nabla f(x_n)$.

4.1.9 Broyden's method

Let $H_0 = \nabla^2 f(x_0)^{-1}$ and

$$\begin{aligned} H_n &= H_{n-1} \\ &+ \frac{((x_n - x_{n-1}) - H_{n-1}(\nabla f(x_n) - f(x_{n-1})))(x_n - x_{n-1})^T H_{n-1}}{(x_n - x_{n-1})^T H_{n-1} (\nabla f(x_n) - \nabla f(x_{n-1}))}, \end{aligned}$$

then $p_n = -H_n \nabla f(x_n)$.

4.1.10 Symmetric Rank-One (SR1) method

Let $H_0 = \nabla^2 f(x_0)^{-1}$ and

$$\begin{aligned} H_n &= H_{n-1} \\ &+ \frac{((x_n - x_{n-1}) - H_{n-1}(\nabla f(x_n) - \nabla f(x_{n-1}))((x_n - x_{n-1}) - H_{n-1}(\nabla f(x_n) - \nabla f(x_{n-1}))^T}{((x_n - x_{n-1}) - H_{n-1}(\nabla f(x_n) - \nabla f(x_{n-1}))^T (\nabla f(x_n) - \nabla f(x_{n-1}))}, \end{aligned}$$

then $p_n = -H_n \nabla f(x_n)$.

4.2 Line Search methods

In every iteration, beside finding a descent direction, the algorithm also requires the magnitude of the descent, denoted by α in the algorithm. One popular method to find α is called line search. The following is the list of line search methods implemented.

4.2.1 Barzilai-Borwein method

The length of the descent direction suggested by Barzilai-Borwein method is calculated with the following formula:

$$\alpha = \frac{(x_n - x_{n-1})(\nabla f(x_n) - \nabla f(x_{n-1}))^T}{\|\nabla f(x_n) - \nabla f(x_{n-1})\|^2}.$$

4.2.2 Backtrack line search method

Backtrack line search is a generic algorithm relying in various conditions to approximate a suitable magnitude for the descent direction.

Starting with a maximum candidate step size value $\alpha_0 > 0$, using search control parameters $\tau \in (0, 1)$ and $c \in (0, 1)$, the backtracking line search algorithm can be expressed as follows:

- Set $t = -cp_n \cdot \nabla f(x_n)$ and iteration counter $j = 0$.
- Until a condition $\dagger(\alpha_j, t)$ is satisfied, repeatedly increment j and set $\alpha_j = \tau \alpha_{j-1}$.
- Return α_j as the solution.

The † condition is usually one of the following:

- **Wolfe condition:** $p_n \cdot \nabla f(x_n + \alpha_j p_n) \geq t$
- **Armijo condition:** $\alpha_j t \geq f(x_n + \alpha_j p_n) - f(x_n)$
- **Goldstein condition:**
  - $f(x_n) + (1 - c)\alpha_j t \leq f(x_n + \alpha_j p_n)$ and
  - $f(x_n + \alpha_j p_n) \leq f(x_n) + \alpha_j t$
- **Strong Wolfe condition:**
  - $f(x_n + \alpha_j p_n) \leq f(x_n) + c_1 \alpha_j t$ and
  - $|p_n \nabla f(x_n + \alpha_j p_n)| \leq c_2 |t|$ for $0 < c_1 < c_2 < 1$
- **Binary Search method**: $f(x_n + \alpha_j p_n) < f(x_n)$

4.3 Termination criterion

At the end of every iteration a termination criterion is evaluated to decide continuation or break of the loop. The following is a list of implemented methods:

4.3.1 Cauchy condition

Given the sequence of calculated points (x_n), this condition checks whether the values of the objective are making enough progress or reached a limit point. In symbols, for $\varepsilon > 0$,

$$|f(x_n) - f(x_{n-1})| < \varepsilon.$$

4.3.2 Cauchy_x condition

Given the sequence of calculated points (x_n), this condition checks whether this sequence is making enough progress or reached an approximate limit point. In symbols, for $\varepsilon > 0$,

$$\|x_n - x_{n+1}\| < \varepsilon.$$

4.3.3 ZeroGradient condition

This condition checks the size of gradient vector at each point found at the end of iteration. If the gradient vector is close enough to zero, then it means that the values of the objective will not make significant progress at any direction. In symbols, for $\varepsilon > 0$,

$$\|\nabla f(x_n)\| < \varepsilon.$$

Note that this condition may not be suitable to solve constrained optimization problems.

CHAPTER 5

Constrained Optimization

It is possible to transform a constrained optimization problem (2.1) to an unconstrained one with some conditions on the solutions. A popular method to do so is known as barrier function where the objective function is modified to penalize the search method if any of the constraints is violated.

5.1 Barrier Function Method

Let ϕ be a function that takes over relatively small values (compare to values of f) over positive reals and big values for negative ones. Then the function $f(x) + \sum_{i=1}^{m} \phi(g_i(x))$ is fairly large values if x is outside of the feasibility region. Therefore, it is likely that a search method tends to focus on the values inside the feasibility region. Similarly, let ψ be a function that returns large positive values apart from 0. Then $f(x) + \sum_{j=1}^{k} \psi(h_j(x))$ is large outside of the feasibility region and rather small on the feasibility region.

Since the values of continuous functions changes gradually, it seems implausible to be able to choose a function that successfully accomplish the above task. So, we employ an increasing sequence of positive numbers (σ_n) that approaches to $+\infty$ and find the optimum value for the function

$$\Lambda_n(x) = f(x) + \frac{1}{\sigma_n} \sum_{i=1}^{m} \phi(g_i(x)) + \sigma_n \sum_{j=1}^{k} \psi(h_j(x)),$$

then the optimum values of Λ_n approaches the optimum value of f inside the feasibility region. Note that if the optimum value is located on the boundary of the feasibility region, this method would only produce an approximate interior substitute. This is why the method is referred to as interior point method.

The following options for the barrier function are implemented:

- **For the inequality conditions:**
  - *Carrol*: is the standard barrier function defined by $-\frac{1}{g_i(x)}$
  - *Logarithmic*: is the standard barrier function defined by $-\log(g_i(x))$
  - *Expn*: is the standard barrier function defined by $e^{-g_i(x)+\epsilon}$
- **For the equality condition:**

- *Courant*: function which is simply defined by $h_j^2(x)$.

Note: The value of ϵ in the code is taken to be equal to $\epsilon = \frac{1}{\sigma_n}$.

CHAPTER 6

Benchmark Problems

We employ the *QuasiNewton* class to solve a few benchmark optimization problems. These benchmarks that are mainly taken from *[MJXY]*.

6.1 Rosenbrock Function

The original Rosenbrock function is $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$ which is a sums of squares and attains its minimum at $(1, 1)$. The global minimum is inside a long, narrow, parabolic shaped flat valley. To find the valley is trivial. To converge to the global minimum, however, is difficult. The same holds for a generalized form of Rosenbrock function which is defined as:

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2.$$

Since f is a sum of squares, and $f(1, \ldots, 1) = 0$, the global minimum is equal to 0. The following code optimizes the Rosenbrock function over $9 - x_i^2 \geq 0$ for $i = 1, \ldots, 9$:

```
from Optimithon import Base
from Optimithon import QuasiNewton
from numpy import array

NumVars = 9
fun = lambda x: sum([100 * (x[i + 1] - x[i]**2)**2 +
                     (1 - x[i])**2 for i in range(NumVars - 1)])
x0 = array([0 for _ in range(NumVars)])

OPTIM = Base(fun, ineq=[lambda x: 9 - x[i]**2 for i in range(NumVars)],
             br_func='Carrol',
             penalty=1.e6,
             method=QuasiNewton, x0=x0,
             t_method='Cauchy',
             dd_method='BFGS',
```

(continues on next page)

(continued from previous page)

```
            ls_method='Backtrack',
            ls_bt_method='Armijo',
            )
OPTIM.Verbose = False
OPTIM.MaxIteration = 1500
OPTIM()
print(OPTIM.solution)
```

The result is:

```
objective: 1.55528520803e-11
x: [ 1.0000001   1.00000013  1.00000013  1.00000005  0.99999996  0.99999993
    1.00000007  0.99999997  0.99999989]
NumIteration: 55
NumFuncEval: 256
success: True
message: Progress in objective values less than error tolerance (Cauchy condition)
RunTime: 0.0378611087799
Family: Quasi-Newton method
Step Size: Backtrack
Backtrack Stop Criterion: Armijo
Descent Direction: BFGS
Termination Criterion: Cauchy
Barrier Function: Carrol
Penalty Factor: 1000000.0
```

6.2 Giunta Function

Giunta is an example of continuous, differentiable, separable, scalable, multimodal function defined by:

$$\begin{aligned} f(x_1, x_2) &= \frac{3}{5} + \sum_{i=1}^{2} [\sin(\frac{16}{15} x_i - 1) \\ &+ \sin^2(\frac{16}{15} x_i - 1) \\ &+ \frac{1}{50} \sin(4(\frac{16}{15} x_i - 1))]. \end{aligned}$$

The following code optimizes f when $1 - x_i^2 \geq 0$:

```
from Optimithon import Base
from Optimithon import QuasiNewton
from numpy import array, sin

fun = lambda x: .6 + (sin((16. / 15.) * x[0] - 1) + (sin((16. / 15.) * x[0] - 1))**2␣
↪+ .02 * sin(4 * ((16. / 15.) * x[0] - 1))) + (
    sin((16. / 15.) * x[1] - 1) + (sin((16. / 15.) * x[1] - 1))**2 + .02 * sin(4 *␣
↪((16. / 15.) * x[1] - 1)))
x0 = array([0 for _ in range(2)])

OPTIM = Base(fun, ineq=[lambda x: 1 - x[i]**2 for i in range(2)],
             br_func='Carrol',
             penalty=1.e6,
             method=QuasiNewton, x0=x0,
             t_method='Cauchy_x',
             dd_method='BFGS',
             ls_method='Backtrack',
             ls_bt_method='Armijo',
```

(continues on next page)

(continued from previous page)

```
            )
OPTIM.Verbose = False
OPTIM.MaxIteration = 1500
OPTIM()
print(OPTIM.solution)
```

The output looks like:

```
objective: 0.0644704205391
x: [ 0.46732003  0.46731857]
NumIteration: 8
NumFuncEval: 20
success: True
message: The progress in values of points is less than error tolerance (0.000000)
RunTime: 0.0011510848999
Family: Quasi-Newton method
Step Size: Backtrack
Backtrack Stop Criterion: Armijo
Descent Direction: BFGS
Termination Criterion: Cauchy_x
Barrier Function: Carrol
Penalty Factor: 1000000.0
```

6.3 Parsopoulos Function

Parsopoulos is defined as $f(x, y) = \cos^2(x) + \sin^2(y)$. The following code computes its minimum where $-5 \leq x, y \leq 5$:

```
from Optimithon import Base
from Optimithon import QuasiNewton
from numpy import array, sin, cos

fun = lambda x: cos(x[0])**2 + sin(x[1])**2
x0 = array((1., -2.))

OPTIM = Base(fun, ineq=[lambda x: 25. - x[j]**2 for j in range(2)],
             br_func='Carrol',
             penalty=1.e6,
             method=QuasiNewton, x0=x0,
             t_method='Cauchy_x',
             dd_method='BFGS',
             ls_method='BarzilaiBorwein',
             )
OPTIM.Verbose = False
OPTIM.MaxIteration = 1500
OPTIM()
print(OPTIM.solution)
```

The solution is the following:

```
objective: 7.48150734385e-16
x: [ 1.57079633 -3.14159263]
NumIteration: 33
NumFuncEval: 36
```

(continues on next page)

(continued from previous page)

```
success: True
message: The progress in values of points is less than error tolerance (0.000000)
RunTime: 0.00488901138306
Family: Quasi-Newton method
Step Size: BarzilaiBorwein
Descent Direction: BFGS
Termination Criterion: Cauchy_x
Barrier Function: Carrol
Penalty Factor: 1000000.0
```

6.4 Shubert Function

Shubert function is defined by:

$$f(x_1, \ldots, x_n) = \prod_{i=1}^{n} \left(\sum_{j=1}^{5} \cos((j+1)x_i + i) \right).$$

It is a continuous, differentiable, separable, non-scalable, multimodal function. The following code compares the result of five optimizers when $-10 \leq x_i \leq 10$ and $n = 2$:

```
from Optimithon import Base
from Optimithon import QuasiNewton
from numpy import array, cos

fun = lambda x: sum([cos((j + 1) * x[0] + j) for j in range(1, 6)]) * \
    sum([cos((j + 1) * x[1] + j) for j in range(1, 6)])
x0 = array((1., -1.))

OPTIM = Base(fun, ineq=[lambda x: 100. - x[i]**2 for i in range(2)],
             br_func='Carrol',
             penalty=1.e6,
             method=QuasiNewton, x0=x0,
             t_method='Cauchy',
             dd_method='Gradient',
             ls_method='Backtrack',
             ls_bt_method='Armijo',
             )
OPTIM.Verbose = False
OPTIM.MaxIteration = 1500
OPTIM()
print(OPTIM.solution)
```

which results in:

```
objective: -18.09556507
x: [-7.06139727 -1.47136939]
NumIteration: 51
NumFuncEval: 1021
success: True
message: Progress in objective values less than error tolerance (Cauchy condition)
RunTime: 2.48312807083
Family: Quasi-Newton method
Step Size: Backtrack
```

(continues on next page)

(continued from previous page)

```
Backtrack Stop Criterion: Armijo
Descent Direction: Gradient
Termination Criterion: Cauchy
Barrier Function: Carrol
Penalty Factor: 1000000.0
```

6.5 McCormick Function

McCormick function is defined by

$$f(x, y) = \sin(x + y) + (x - y)^2 - 1.5x + 2.5y + 1.$$

Attains its minimum at $f(-.54719, -1.54719) \approx -1.9133$:

```
objective: -1.91322295498
x: [-0.54719755 -1.54719755]
NumIteration: 9
NumFuncEval: 23
success: True
message: The progress in values of points is less than error tolerance (0.000000)
RunTime: 0.000735998153687
Family: Quasi-Newton method
Step Size: Backtrack
Backtrack Stop Criterion: Armijo
Descent Direction: BFGS
Termination Criterion: Cauchy_x
Barrier Function: Carrol
Penalty Factor: 100000.0
```

CHAPTER 7

Code Documentation

7.1 'base' Module

This module implements general containers for typical optimization methods.

class base.**Base**(*obj*, ***kwargs*)

This is the base class that serves all the iterative optimization methods. An object derived from *Base* requires the following parameters:

Parameters

- **obj** – *MANDATORY*- is a real valued function to be minimized
- **x0** – an initial guess of the optimal point
- **method** – the optimization class which implements *iterate* and *terminate* procedures (default: *OptimTemplate* that returns the value of the function at the initial point *x0*)
- **Verbose** – *Boolean*- If *True* prompts messages at every stage of the iteration as well as termination
- **kwargs** – the rest of parameters that will ba passed to *method*

The object then passes all other given parameters to the *method* class for further processes. When a termination condition is satisfied, the object fills the results in the *solution* attribute which is an instance of *Solution* class. The given class *method* can pass arbitrary pieces of information to the solution by modifying its *MetaData* dictionary.

When an object *optim* of type *Base* initiated, the optimization process can be invoked by calling the object itself like a function:

```
optim = Base(f, method=QuasiNewton, x0=init_point)
optim()
print(optim.solution)
```

class base.**OptimTemplate**(*obj*, ***kwargs*)

Provides a template for an iterative optimization method.

Parameters

- **obj** – a real valued function (objective function)
- **x0** – an initial guess for a (local) minimum
- **jac** – a vector calculating the gradient of the objective function (optional, if not given will be numerically approximated)
- **difftool** – an object to calculate *Gradient* and *Hessian* of the objective (optional, default *NumericDiff.Simple*)

class base.**Solution**
A class to keep outcome and details of the optimization run.

7.2 'QuasiNewton' Module

This module contains implementations of variations of unconstrained optimization methods known as Quasi-Newton methods. A Quasi-Newton method is an iterative algorithm that approximates a local minima of the objective function. Starting with a given initial point *x0*, each iteration consists of three major subprocedures:

- Finding a descent direction (via *DescentDirection* class)
- Finding the length of descent (using *LineSearch* class)
- Check the termination condition (*Termination* class).

The right strategy to attempt an optimization problem must be pre-determined, otherwise, it uses a default set up to solve the problem.

class QuasiNewton.**Barrier**(*QNRef*, ***kwargs*)
Implementation of some barrier functions to be used for constrained optimization problems. Three barrier functions are implemented:

- *Carrol*: is the standard barrier function defined by $-\frac{1}{g_i(x)}$
- *Logarithmic*: is the standard barrier function defined by $-\log(g_i(x))$
- *Expn*: is the standard barrier function defined by $e^{-g_i(x)+\epsilon}$

The only barrier function implemented for the equality is the typical function known as *Courant* function which is simply $h_j^2(x)$.

The default barrier function is *Carrol* and the default penalty factor is 10^{-5}. To specify the barrier function and penalty factor initiate the optimizer with keywords *br_func* that accepts one of the above three values and *penalty* that must be a positive real number.

class QuasiNewton.**DescentDirection**(*QNRef*, ***kwargs*)
Implements various descent direction methods for Quasi-Newton methods. The descent method can be determined at initiation using *dd_method* parameter. The following values are acceptable:

- 'Gradient': (default) The steepest descent direction.
- 'Newton': Newton Conjugate Gradient method.
- 'FletcherReeves': Fletcher-Reeves method.
- 'PolakRibiere': Polak-Ribiere method.
- 'HestenesStiefel': Hestenes-Stiefel method.
- 'DaiYuan': Dai-Yuan method
- 'DFP': Davidon-Fletcher-Powell formula.

- 'BFGS': Broyden-Fletcher-Goldfarb-Shanno algorithm.
- 'Broyden': Broyden's method.
- 'SR1': Symmetric rank-one method.

To calculate derivatives, the *QuasiNewton* class uses the object provided as the value of the *difftool* variable at initiation.

BFGS()

Returns the descent direction determined by *Broyden-Fletcher-Goldfarb-Shanno* algorithm

Broyden()

Returns the descent direction determined by *Broyden's* method

DFP()

Returns the descent direction determined by *Davidon-Fletcher-Powell* formula

DaiYuan()

Returns the descent direction determined by *Dai-Yuan* method

FletcherReeves()

Returns the descent direction determined by *Fletcher-Reeves* method

Gradient()

Returns the gradient at current point

HestenesStiefel()

Returns the descent direction determined by *Hestenes-Stiefel* method

Newton()

Returns the descent direction determined by *Newton Conjugate Gradient* method

PolakRibiere()

Returns the descent direction determined by *Polak-Ribiere* method

SR1()

Returns the descent direction determined by *Symmetric rank-one* method

class QuasiNewton.**LineSearch**(*QNref*, ***kwargs*)

This class provides the step length at each iteration. The value of *ls_bt_method* at initiation determines whether to use *'BarzilaiBorwein'* method or a variation of *'Backtrack'* (default). It accepts a control parameter *tau* and *max_lngth* at initiation. *tau* must be a positive real less than 1. The variation of the backtrack is then determined by the value of *ls_bt_method* which can be selected among the following:

- 'Armijo': indicates *Armijo* condition and the parameter *c1* can be modified at initiation as well.
- 'Wolfe': indicates *Wolfe* condition and the parameters *c1* and *c2* can be modified at initiation.
- 'StrongWolfe': indicates *StrongWolfe* condition and the parameters *c1* and *c2* can be modified at initiation.
- 'Goldstein': indicates *Goldstein* condition and the parameter *c1* can be modified at initiation.

Armijo(*alpha*, *ft_x*, *tx*)

Implementation of *Armijo*.

Parameters

- **alpha** – current candidate for step length

- **ft_x** – value of the objective at the candidate point
- **tx** – the candidate point

Returns *True* or *False*

Backtrack()
A generic implementation of *Backtrack*.

Returns step length

BarzilaiBorwein()
Implementation of *Barzilai-Borwein*.

Returns step length

Goldstein(*alpha*, *ft_x*, *tx*)
Implementation of *Goldstein*.

Parameters

- **alpha** – current candidate for step length
- **ft_x** – value of the objective at the candidate point
- **tx** – the candidate point

Returns *True* or *False*

StrongWolfe(*alpha*, *ft_x*, *tx*)
Implementation of *Strong Wolfe*.

Parameters

- **alpha** – current candidate for step length
- **ft_x** – value of the objective at the candidate point
- **tx** – the candidate point

Returns *True* or *False*

Wolfe(*alpha*, *ft_x*, *tx*)
Implementation of *Wolfe*.

Parameters

- **alpha** – current candidate for step length
- **ft_x** – value of the objective at the candidate point
- **tx** – the candidate point

Returns *True* or *False*

class QuasiNewton.**QuasiNewton**(*obj*, ***kwargs*)

Parameters

- **obj** – the objective function
- **ineq** – *(optional)* list of inequality constraints, default: *[]*
- **eq** – *(optional)* list of equality constraints, default: *[]*
- **ls_method** – *(optional)* the line search strategy, default: *Backtrack*
- **ls_bt_method** – *(optional)* the backtrack termination condition, default: *Armijo*
- **dd_method** – *(optional)* the descent direction method, default: *Gradient*

- **t_method** – *(optional)* termination condition, default: *Cauchy*
- **br_func** – *(optional)* barrier function family, default: *Carrol*
- **penalty** – *(optional)* penalty factor for the barrier function, default: *1.e5*
- **max_iter** – *(optional)* maximum number of iterations, default: *100*

This class hosts a family of first and second order iterative methods to solve an unconstrained optimization problem. The general schema follows the following steps:

- Given the point x, find a suitable descent direction p.
- Find a suitable length α for the direction p such that $x + \alpha p$ results in an appropriate decrease in values of the objective.
- Update x to $x + \alpha p$ and repeat the above steps until a termination condition is satisfied.

The initial value for *x* can be set at initiation by passing *x0=init_point* to the *Base* instance. There are various methods to determine the descent direction *p* at each step. The *DescentDirection* class implements a variety of these methods. To choose one of these methods one should pass the method by its known name at initiation simply by setting *dd_method='method name'*. This parameter will be passed to *DescentDirection* class (see the documentation for *DescentDirection*). Also, to determine a suitable value for $alpha$ various options are available the class *LineSearch* is responsible for handling the computation for $alpha$. The parameters *ls_method* and *ls_bt_method* can be set at initiation to determine the details for line search. The termination condition also can vary and the desired condition can be determined by setting *t_method* at initiation which will be passed to the *Termination* class. Each of these classes may accept other parameters that can be set at initiation. To find out about those parameter see the corresponding documentation.

iterate()
This method updates the *iterate* method of the *OptimTemplate* by customizing the descent direction method as well as finding the descent step length. These method can be determined by the user.

Returns None

terminate()
This method updates the *terminate* method of the *OptimTemplate* which is given by user.

Returns *True* or *False*

class QuasiNewton.**Termination**(*QNRef*, ***kwargs*)
Implements various termination criteria for Quasi-Newton loop. A particular termination method can be selected at initiation of the *Base* object by setting *t_method* with the name of the method as an string. The following termination criteria are implemented:

- 'Cauchy': Checks of the changes in the values of the objective function are significant enough or not.
- 'ZeroGradient': Checks if the gradient of the objective is close to zero or not.

The value of the tolerated error is a property of *OptimTemplate* and hence can be modified as desired.

Cauchy()
Checks if the values of the objective function form a Cauchy sequence or not.

Returns *True* or *False*

Cauchy_x()
Checks if the sequence of points form a Cauchy sequence or not.

Returns *True* or *False*

ZeroGradient()
Checks if the gradient vector is small enough or not.

Returns *True* or *False*

7.3 'NumericDiff' Module

This module provides very basic way to numerically approximate partial derivatives, gradient and Hessian of a function.

class NumericDiff.**Simple**(***kwargs*)

A simple class to calculate partial derivatives of a given function. Passing a value for *Infinitesimal* forces the calculations to be done according to the infinitesimal value provided by user. Otherwise, the default value (1.e-7) is used.

Diff(*f*, *i=0*)

Parameters

- **f** – a real valued function
- **i** – the index variable for differentiation

Returns partial derivative of f with respect to :math:'i^{th}' variable as a function.

Gradient(*f*)

Parameters **f** – a real valued function

Returns a vector function that returns the gradient vector of f at each point.

Hessian(*f*)

Parameters **f** – a real valued function

Returns the Hessian matrix of f at each point

7.4 'excpt' Module

This module provides descriptive exception for the other modules.

exception excpt.**DiffEror**(**args*)

Errors that may have happened while calculating derivatives of functions.

exception excpt.**DirectionError**(**args*)

Handles the errors caused during the computation of a descent direction.

exception excpt.**Error**(**args*)

Generic errors that may occur in the course of a run.

exception excpt.**MaxIterations**(**args*)

Errors caused by reaching the preset maximum number of iterations.

exception excpt.**Undeclared**(**args*)

Raised when an undeclared function is used.

exception excpt.**ValueRange**(**args*)

Errors resulted from computations over unauthorized regions.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Bibliography

[JNSJW] J. Nocedal, S. J. Wright, *Numerical Optimization*, Second Ed., Springer, New York, NY, USA, 2006.

[MJXY] M. Jamil, Xin-She Yang, *A literature survey of benchmark functions for global optimization problems*, IJMMNO, Vol. 4(2), 2013.

Python Module Index

Index

U

V

W

Z