# Some Applications of Python

*GE Qian*

*November 15, 2015*

## Contents

## 1 Introduction

This talks will give some applications of Python on optimization and numerical studies. In the first section, we may discuss the concept of *Algorithm*. It builds a bridge for solution methods and the computer. Then we move on to some common problems in modern engineering disciplines, line search method for optimization problems. Some basic algorithms are provided and implemented later in this section. Finally, this document will provide some packages for different tasks.

## 2 Algorithm

The term algorithm is used in computer science to describe a finite, deterministic, and effective problem-solving method suitable for implementation as a computer program.

- step-by-step description
- easy to be implemented (from word to code)
- involves both data stuctures and the logic

For mathematical problem, we can describe an algorithm more explicitly in the following aspects:

- what is the initial condition?
- how to find a better solution in the feasible region?
- when to stop?

## 3 Optimization - Line Search Method

A simple unconstrained problem: how to find the minimum of $100(x_1^2 - x_2)^2 + (x_2 - 1)^2$?

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D

x = np.linspace(-2, 2, 100)
y = np.linspace(-2, 2, 100)
x, y = np.meshgrid(x, y)
z = 100*np.square(np.square(x)-y)+np.square(x-1)
fig = plt.figure()
ax = fig.gca(projection='3d')
surf = ax.plot_surface(x, y, z, rstride=1, cstride=1, cmap=cm.coolwarm, linewidth=\
       0, antialiased=False)
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.savefig('./res/surface.png')
```

A 3D drawing of this objective function is:

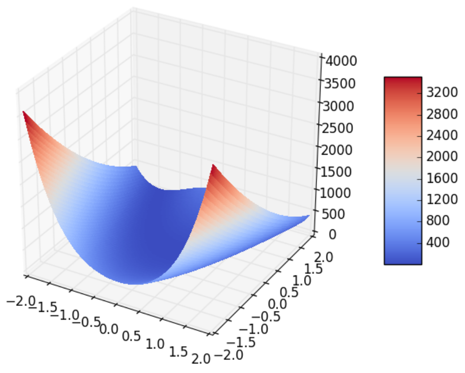

Figure 1: Surface of the function

A straightforward method for this specific problem is to take the first order derivatives of each variable, and let both equations equal to 0. It is an *analytical* method and we won't consider it here. What we need is a *numerical* method.

There are two basic iterative methods for optimization problems: the **line search** method and the **trust region** method. The line search approach first finds a descent direction along which the objective function $f$ will be reduced and then computes the step size that determines how far $\mathbf{x}$ should move along that direction.

The updating rule of line search method is $x_{k+1} = x_k + \alpha_k * d(x_k)$, we need to compute the direction $d(x_k)$ and step size $\alpha_k$ seperately.

- direction: gradient descent, Newton's method, etc.
- step size: Golden-section method, Armijo rule, Wolfe rule, etc.

## 3.1 Find the descent direction

This section will show how two direction-finding methods work: the gradient descent method and the Newton's method.

### 3.1.1 Gradient descent method(梯度下降法)

The algorithm of gradient descent method reads:

1. Initialization: find a initial value of $x_0$ in the feasible set
2. Convergence Test: calculate the gradient of current value $\nabla f(x)$, and know if it satisfy the convergent criteria.
3. Update: if not converge, update $x_{i+1} = x_i - r\nabla f(x)$, where $r$ is a small number to control the step size.

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def func(x):
    return 100*np.square(np.square(x[0])-x[1])+np.square(x[0]-1)

# first order derivatives of the function
def dfunc(x):
    df1 = 400*x[0]*(np.square(x[0])-x[1])+2*(x[0]-1)
    df2 = -200*(np.square(x[0])-x[1])
    return np.array([df1, df2])

def grad(x, max_int):
    miter = 1
    step = .0001/miter
    vals = []
    objectfs = []
    # you can customize your own condition of convergence, here we limit the number of iterations
    while miter <= max_int:
        vals.append(x)
        objectfs.append(func(x))
        temp = x-step*dfunc(x)
        if np.abs(func(temp)-func(x))>0.01:
            x = temp
        else:
            break
        print(x, func(x), miter)
        miter += 1
    return vals, objectfs, miter

start = [5, 5]
val, objectf, iters = grad(start, 50)

x = np.array([i[0] for i in val])
y = np.array([i[1] for i in val])
z = np.array(objectf)
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.scatter(x, y, z, label='gradient descent method')
ax.legend()
plt.savefig('./res/g-d.png')
```

```
## (array([ 0.9992,  5.4   ]), 1937.4076932352416, 1)
## (array([ 1.17512328,  5.31196801]), 1545.3486587826624, 2)
## (array([ 1.35986715,  5.23334695]), 1145.3483938946076, 3)
## (array([ 1.54387268,  5.16566478]), 774.31603689179224, 4)
## (array([ 1.71557359,  5.11002234]), 470.02709876928111, 5)
## (array([ 1.8641247 ,  5.06668575]), 254.10551268351739, 6)
## (array([ 1.98263882,  5.03485125]), 122.84597900325505, 7)
## (array([ 2.06999519,  5.01277136]), 54.127460033900505, 8)
## (array([ 2.13005045,  4.99821354]), 22.538207813017163, 9)
## (array([ 2.16911097,  4.98899156]), 9.4295327385708028, 10)
## (array([ 2.19351384,  4.98331258]), 4.3763298146343708, 11)
## (array([ 2.20834981,  4.97987639]), 2.522400568669064, 12)
## (array([ 2.2172125 ,  4.97781504]), 1.863329534402812, 13)
## (array([ 2.22244857,  4.97657936]), 1.6335223512808446, 14)
## (array([ 2.22552013,  4.97583333]), 1.5543108119816416, 15)
## (array([ 2.22731302,  4.97537546]), 1.5271837782347222, 16)
```

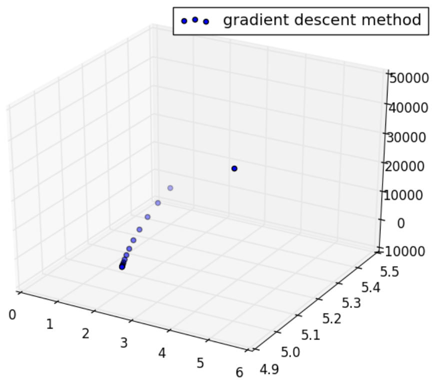

Figure 2: gradient descent method

### 3.1.2 Newton's method(牛顿法)

Similar to the gradient method, the algorithm of Newton's method for optimization reads:

1. Initialization: find a initial value of $x_0$ in the feasible set
2. Convergence Test: calculate the gradient and Hessian matrix of current value $\nabla f(x)$, and know if it satisfies the convergent criteria.
3. Update: if not converge, update $x_{i+1} = x_i - r\mathbf{H}^{-1}\nabla f(x)$, where $r$ is a small number to control the step size and $\mathbf{H}$ is the Hessian matrix.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from numpy.linalg import inv

def func(x):
    return 100*np.square(np.square(x[0])-x[1])+np.square(x[0]-1)
```

```python
# first order derivatives of the function
def dfunc(x):
    df1 = 400*x[0]*(np.square(x[0])-x[1])+2*(x[0]-1)
    df2 = -200*(np.square(x[0])-x[1])
    return np.array([df1, df2])

def invhess(x):
    df11 = 1200*np.square(x[0])-400*x[1]+2
    df12 = -400*x[0]
    df21 = -400*x[0]
    df22 = 200
    hess = np.array([[df11, df12], [df21, df22]])
    return inv(hess)

def newton(x, max_int):
    miter = 1
    step = .5
    vals = []
    objectfs = []
    # you can customize your own condition of convergence, here we limit the number of iterations
    while miter <= max_int:
        vals.append(x)
        objectfs.append(func(x))
        temp = x-step*(invhess(x).dot(dfunc(x)))
        if np.abs(func(temp)-func(x))>0.01:
            x = temp
        else:
            break
        print(x, func(x), miter)
        miter += 1
    return vals, objectfs, miter

start = [5, 5]
val, objectf, iters = newton(start, 50)

x = np.array([i[0] for i in val])
y = np.array([i[1] for i in val])
z = np.array(objectf)
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.scatter(x, y, z, label='newton method')
plt.savefig('./res/newton.png')
```

```
## (array([  4.99950012,  14.99500125]), 10015.996500999738, 1)
## (array([  4.99850075,  19.98500862]), 2515.9891319336307, 2)
## (array([  4.9965035 ,  22.46504264]), 640.97431563468137, 3)
## (array([  4.99251498,  23.67518762]), 172.19472181001353, 4)
## (array([  4.98456188,  24.22078475]), 54.948276827358256, 5)
## (array([  4.96875194,  24.37570969]), 25.534508005533738, 6)
## (array([  4.93753006,  24.22183522]), 17.980607577859715, 7)
## (array([  4.87690338,  23.70182702]), 15.708688997277847, 8)
## (array([  4.7659566 ,  22.66085333]), 14.468535895722859, 9)
## (array([  4.60498788,  21.153258  ]), 13.273196783901964, 10)
## (array([  4.44867175,  19.7399179 ]), 12.151019317491137, 11)
## (array([  4.29405728,  18.38964106]), 11.093732736283506, 12)
## (array([  4.14236045,  17.11149476]), 10.101532481072594, 13)
## (array([  3.99316572,  15.89928575]), 9.1714398441305462, 14)
## (array([  3.84669105,  14.7525338 ]), 8.3016589190627066, 15)
## (array([  3.70291355,  13.6686477 ]), 7.4899634874303267, 16)
## (array([  3.56190493,  12.64582277]), 6.7342892273516286, 17)
## (array([  3.42370436,  11.6819802 ]), 6.0325191095435988, 18)
## (array([  3.28836654,  10.77515251]), 5.3825608388318251, 19)
## (array([ 3.1559441 ,  9.92334647]), 4.7823198308364177, 20)
## (array([ 3.02649439,  9.12459271]), 4.2297091672677336, 21)
## (array([ 2.90007737,  8.37692968]), 3.7226467068415037, 22)
## (array([ 2.77675676,  7.67841059]), 3.2590566815235289, 23)
## (array([ 2.65660008,  7.02710263]), 2.8368698503035121, 24)
## (array([ 2.53967916,  6.42108902]), 2.4540242239312926, 25)
## (array([ 2.42607046,  5.85847036]), 2.1084657085379117, 26)
## (array([ 2.31585568,  5.33736646]), 1.7981489002130808, 27)
## (array([ 2.20912221,  4.85591835]), 1.5210379805225525, 28)
## (array([ 2.10596382,  4.41229066]), 1.2751077585294368, 29)
## (array([ 2.00648136,  4.00467423]), 1.0583448828563189, 30)
## (array([ 1.91078358,  3.63128922]), 0.86874926405827646, 31)
## (array([ 1.81898804,  3.29038872]), 0.70433575590871178, 32)
## (array([ 1.73122212,  2.9802628 ]), 0.56313615952749274, 33)
## (array([ 1.64762421,  2.69924331]), 0.44320163392601775, 34)
## (array([ 1.56834489,  2.44570936]), 0.34260562386752896, 35)
## (array([ 1.4935482 ,  2.21809352]), 0.25944745379287382, 36)
## (array([ 1.42341286,  2.01488885]), 0.19185678962341826, 37)
## (array([ 1.3581331 ,  1.83465641]), 0.13799924506512215, 38)
## (array([ 1.29791881,  1.67603292]), 0.096083514589131752, 39)
## (array([ 1.24299405,  1.53773732]), 0.064370562606530568, 40)
## (array([ 1.19359251,  1.41857412]), 0.041185597168423842, 41)
## (array([ 1.14994716,  1.31742909]), 0.024933802008029717, 42)
## (array([ 1.1122697,  1.2332496]), 0.014121033914283581, 43)
```

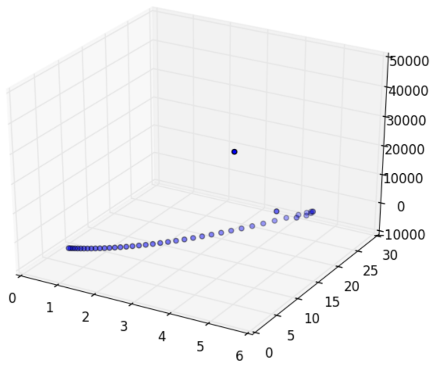

Figure 3: Newton's method

## 3.2 Find the step size

This section will depict three common used methods to determine the step size and their implementations, Golden section method (exact method), Armijo rule and Wolfe condition (inexact methods).

### 3.2.1 Golden section method (黄金分割搜索)

The Golden section method can be used to find the optima of unimodal function, the method firstly set the lower and upper bound of $x$ and then iterative determine the next point.

1. Initialization: set the lower $x_1$ and upper bound $x_2$ of the interval to search, calculate the golden number $\phi = \frac{\sqrt{5}-1}{2}$
2. Convergence test: calculate the absolute difference of current $f(x)$ at current lower bound and the upper bound, to check if it satisfies the convergence condition
3. Update: if not converge

- calculate $t_1 = x_2 - \phi(x_2 - x_1)$, $t_2 = x_1 + \phi(x_2 - x_1)$ and $f(t_1), f(t_2)$
- if $f(t_1) < f(t_2)$, then keeps $x_1$ and $x_2 = t_2$, $t_2 = t_1$
- otherwise, keeps $x_2$ and $x_1 = t_1$, $t_1 = t_2$

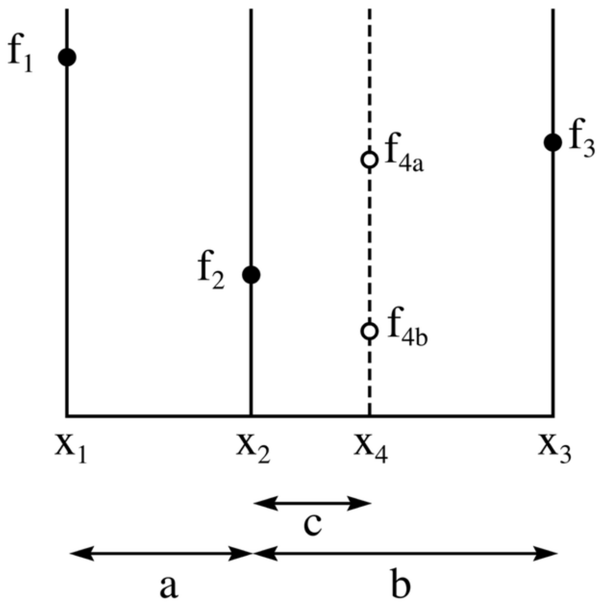

Figure 4: golden section method

```
import numpy as np
import matplotlib.pyplot as plt

seg = (np.sqrt(5)-1)/2

def golden(f, lower, upper, merror):
    error = 1000
    vals = []
    vals.append((lower+upper)/2)
    objectf = []
    objectf.append(f((lower+upper)/2))
    # you can customize your own condition of convergence, here we limit the error term
    while error >= merror:
        temp1 = upper-seg*(upper-lower)
        temp2 = lower+seg*(upper-lower)
        if f(temp1)-f(temp2)>0:
            upper = temp2
        else:
            lower = temp1
        error = np.abs(f(temp1)-f(temp2))
        vals.append((lower+upper)/2)
        objectf.append(f((lower+upper)/2))
    return (temp2+temp1)/2, f((temp2+temp1)/2), vals, objectf

# an example
obj = lambda x: np.power(x, 4)-5*np.power(x, 3)-2*np.power(x, 2)+24*x
(ub, lb) = (3, 0)

val, maxima, xs, ys = golden(obj, lb, ub, 0.003)

print("The value and maximal is "+str(val)+", "+str(maxima))
plt.plot(xs,ys,'ro')
plt.title('Golden section method')
plt.savefig('./res/gsr.png')
```

```
## The value and maximal is 1.39667443875, 19.8015554843
```

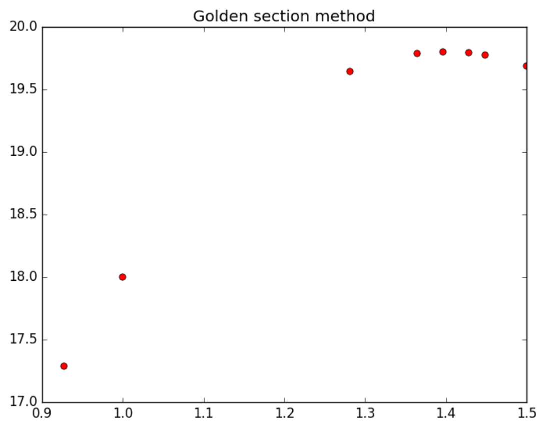

Figure 5: golden section method

### 3.2.2 Armijo condition(Armijo 条件)

The Armijo is used to determine a step size that is not too big. It tries to find a step size such that: $f(x_k + \alpha_k d_k) \leq f(x_k) + \alpha_k \beta \nabla f(x_k)^T d_k$, where $\beta \in (0, 1)$.

Here is an implementation of Armijo's condition:

- Initialization: set the values of $\alpha_0 = \theta^0$, $\beta$, etc.
- Convergence test: $f(x_k + \alpha_m d_k) \leq f(x_k) + \alpha_m \beta \nabla f(x_k)^T d_k$
- Update: $\alpha_m = \theta^m$

```python
import random
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

# the objective function
def func(x):
    return 100*np.square(np.square(x[0])-x[1])+np.square(x[0]-1)

# first order derivatives of the function
def dfunc(x):
    df1 = 400*x[0]*(np.square(x[0])-x[1])+2*(x[0]-1)
    df2 = -200*(np.square(x[0])-x[1])
    return np.array([df1, df2])

# the armijo algorithm
def armijo(valf, grad, niters):
    #beta = random.random()
    #sigma = random.uniform(0, .5)
    beta = 0.25
    sigma = 0.25
    (miter, iter_conv) = (0, 0)
    conval = [0,0]
    val = []
    objectf = []
    val.append(valf)
    objectf.append(func(valf))
    while miter <= niters:
        leftf = func(valf+np.power(beta, miter)*grad)
        rightf = func(valf) + sigma*np.power(beta, miter)*dfunc(valf).dot(grad)
        if leftf-rightf <= 0:
            iter_conv = miter
            conval = valf+np.power(beta, iter_conv)*grad
            break
        miter += 1
        val.append(conval)
        objectf.append(func(conval))
    return conval, func(conval), iter_conv, val, objectf

# initialization
start = np.array([-.3, .1])
direction = np.array([1, -2])
maximum_iterations = 30

converge_value, minimal, no_iter, val, objf = armijo(start, direction, maximum_iterations)
print("The value, minimal and number of iterations are " + str(converge_value) + \
    ", " + str(minimal) + ", " + str(no_iter))
x = np.array([i[0] for i in val])
y = np.array([i[1] for i in val])
```

```
z = np.array(objf)
fig = plt.figure()
ax = fig.gca(projection='3d')
ax.scatter(x, y, z, label='Armijo Rule')
ax.legend()
plt.savefig('./res/armijo.png')
```

```
## The value, minimal and number of iterations are [-0.284375  0.06875 ], 1.66430649757, 3
```

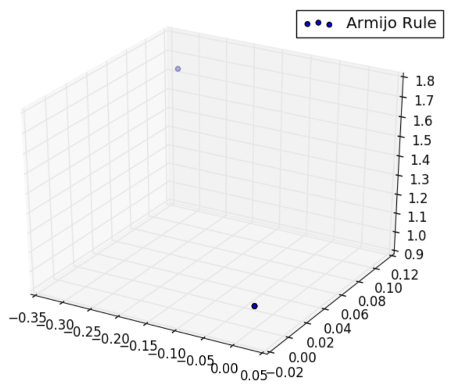

Figure 6: Armijo rule

### 3.2.3 Wolfe condition(Wolfe 条件)

Armijo condition only guarantee that the step size. Wolfe condition add a constraint on the slope.

- Armijo condition: as previous section, $f(x_k + \alpha_m d_k) \leq f(x_k) + \alpha_m c_1 \nabla f(x_k)^T d_k$
- curvature condition: $d_k \nabla f(x_k + \alpha_k d_k) \geq c_2 d_k^T \nabla f(x_k)$ where $1 < c_1 < c_2 < 1$

A bi-section implementation of this condition is:

- Initialization: choose $1 < c_1 < c_2 < 1$, and set $\alpha = 0,\ \beta = \infty,\ t = 1$
- Update: if $f(x+td) > f(x)+c_1 t f'(x)*d$, set $\beta = t$ and reset $t = (\alpha+\beta)/2$ else if $f'(x+td)*d \leq c_2 f'(x)*d$, set $\alpha = t$ and reset $t = 2\alpha$ if $\beta = \infty$ or $(\alpha + \beta)/2$ otherwise
- Convergence test: else: stop

```
import random
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

# bisection search of wolfe condition
def func(x):
    return 100*np.square(np.square(x[0])-x[1]) + np.square(x[0]-1)
```

```python
def dfunc(x):
    df1 = 400*x[0]*(np.square(x[0])-x[1])+2*(x[0]-1)
    df2 = -200*(np.square(x[0])-x[1])
    return np.array([df1, df2])

def wolfe(valf, direction, max_iter):
    (alpha, beta, step, c1, c2) = (0, 1000, 5.0, 0.15, 0.3)
    i = 0
    stop_iter = 0
    stop_val = valf
    minima = 0
    val = []
    objectf = []
    val.append(valf)
    objectf.append(func(valf))
    while i <= max_iter:
        # first confition
        leftf = func(valf + step*direction)
        rightf = func(valf) + c1*dfunc(valf).dot(direction)
        if leftf > rightf:
            beta = step
            step = .5*(alpha + beta)
            val.append(valf+step*direction)
            objectf.append(leftf)
        elif dfunc(valf + step*direction).dot(direction) < c2*dfunc(valf).dot(direction):
            alpha = step
            if beta > 100:
                step = 2*alpha
            else:
                step = .5*(alpha + beta)
            val.append(valf+step*direction)
            objectf.append(leftf)
        else:
            val.append(valf+step*direction)
            objectf.append(leftf)
            break
        i += 1
        stop_val = valf + step*direction
        stop_iter = i
        minima = func(stop_val)
    print(val, objectf)
    return stop_val, minima, stop_iter, step, val, objectf

start = np.array([.6, .5])
dirn = np.array([-.3, -.4])
converge_value, minimal, no_iter, size, val, objectf = wolfe(start, dirn, 30)
print("The value, minimal and iterations needed are " + str(converge_value) + ", " \
+ str(minimal) + ", " + str(no_iter) + ', ' + str(size))
x = np.array([i[0] for i in val])
y = np.array([i[1] for i in val])
z = np.array(objectf)
fig = plt.figure()
ax = fig.gca(projection='3d')
```

```
ax.scatter(x, y, z, label='Wolfe Rule')
ax.legend()
plt.savefig('./res/wolfe.png')
```

```
## ([array([ 0.6,  0.5]), array([-0.15, -0.5 ]), array([ 0.225,  0.  ]), array([ 0.225,  0.  ])], [2.12000000
## The value, minimal and iterations needed are [ 0.225  0.   ], 0.8569140625, 2, 1.25
```

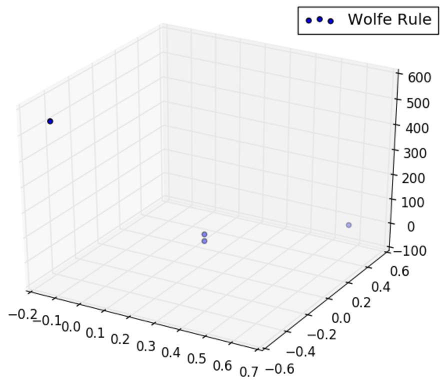

Figure 7: Bisection implementation of Wolfe's Condition

## 4 Packages

- optimization: *scipy.optimize, cvxopt*
- statistics and machine learning: *scikit-learn*
- write your own when no package available