**MIE1621 – Non-Linear Programming**

# **Course Project**

# **Fall 2017**

Jonathan Smith

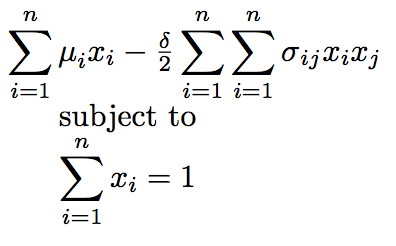
smithj70

999694521

Dec 7, 2017

# Part 1: Problem formulation

The problem at hand is the optimization of the risk adjusted return of a financial portfolio of n assets. The risk adjusted return is governed by the following equations:



Where:

is the return of asset i,

is the covariance of asset I and j,

is the fraction of wealth invested in asset i,

is the risk tolerance of the investor, set to a value between 3.5 and 4.5.

The goal of this optimization will be to maximize the expected return, given some risk tolerance. The equation is therefore multiplied by -1 in order to apply conventional minimization techniques.

This equation can be re-written in matrix form for scalability:

s.t. where

In order to reconfigure this equation as an unconstrained optimization, the Lagrangian of this problem is constructed, ensuring that the optimal solutions meet the constraint:

In order to satisfy the FONC, the following partial derivatives of L are found:

In order to solve these simultaneously, the following equation is constructed, where is an nxn matrix, and are an n-dimensional column vectors:

Lastly, the hessian can be shown as:

The gradient for a simple 2-stock portfolio is shown as below, however, this generalizes for n stocks.

# Part 2: Naïve Newton, SDM, BFGS Solution Methods

This portion of the report details attempts to solve the above problem using Newton’s Method, Steepest Descent Method, and BFGS Quasi-Newton Method, with a step length of 1. The functions to execute this code are shown in Appendix A. Note that these functions cannot be run independently of other accompanying functions. The full python notebook is submitted along with this report, or available at this link: <https://github.com/jonsmith359/MIE1621-Project>.

These methods all follow the same basic form of iterations:

where is a step length and is a descent direction. For this portion of the project, . The way that these three methods differ is in their calculation of descent direction.

Newton:

SDM:

BFGS:

In the case of BFGS, an approximation of the inverse hessian is used, based on the estimation formula:

These three methods were performed on the sample dataset given, with starting portfolio weights of [1,1,1]. The results are summarized below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Return | var\_1 | var\_2 | var\_3 |
| 0 | 0.1073 | 0.02778 | 0.00387 | 0.00021 |
| 1 | 0.0731 | 0.00387 | 0.01112 | -0.0002 |
| 2 | 0.0621 | 0.00021 | -0.0002 | 0.00115 |

Table : Sample Data Set

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Convergence | # Iterations | Time |
| Newton | Yes | 1 | 4.05 µs |
| SDM | No | 100 | 5.01 µs |
| BFGS | Yes | 17 | 5.03 µs |

Table 2: Part 2 Results

# Part 3 Newton, SDM, BFGS Solution Methods with Backtracking

# Part 4: Scaling up

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Return | var\_1 | var\_2 | var\_3 | var\_4 | var\_5 | var\_6 | var\_7 | var\_8 | var\_9 | var\_10 |
| 0 | 146.4697 | 0.0118 | 0.0082 | 0.0098 | 0.0033 | 0.0073 | 0.0174 | -0.0014 | 0.0071 | -0.0132 | 0.0110 |
| 1 | 903.0575 | 0.0082 | 0.0071 | 0.0082 | 0.0040 | 0.0055 | 0.0130 | -0.0006 | 0.0059 | -0.0101 | 0.0087 |
| 2 | 76.8606 | 0.0098 | 0.0082 | 0.0110 | 0.0038 | 0.0074 | 0.0177 | -0.0014 | 0.0064 | -0.0128 | 0.0111 |
| 3 | 94.0385 | 0.0033 | 0.0040 | 0.0038 | 0.0341 | 0.0018 | 0.0045 | 0.0003 | 0.0043 | -0.0054 | 0.0031 |
| 4 | 283.0916 | 0.0073 | 0.0055 | 0.0074 | 0.0018 | 0.0058 | 0.0139 | -0.0017 | 0.0051 | -0.0097 | 0.0074 |
| 5 | 205.5607 | 0.0174 | 0.0130 | 0.0177 | 0.0045 | 0.0139 | 0.0346 | -0.0046 | 0.0137 | -0.0243 | 0.0174 |
| 6 | 103.7449 | -0.0014 | -0.0006 | -0.0014 | 0.0003 | -0.0017 | -0.0046 | 0.0016 | -0.0015 | 0.0028 | -0.0009 |
| 7 | 2.4261 | 0.0071 | 0.0059 | 0.0064 | 0.0043 | 0.0051 | 0.0137 | -0.0015 | 0.0153 | -0.0122 | 0.0057 |
| 8 | 26.9836 | -0.0132 | -0.0101 | -0.0128 | -0.0054 | -0.0097 | -0.0243 | 0.0028 | -0.0122 | 0.0192 | -0.0120 |
| 9 | 835.0006 | 0.0110 | 0.0087 | 0.0111 | 0.0031 | 0.0074 | 0.0174 | -0.0009 | 0.0057 | -0.0120 | 0.0135 |

Table : Scaled-Up Data

# Appendix A: Numerical Method Functions

## A1 Newton

def newton (df,d,init,tol=0.001,limit=100, backtrack=False, Rho=0.5, gamma=0.01):

'''

Perform Newton Method to find optimal solution for some function f

df-dataframe containing return and covariance

d-risk tolerance factor

init-initial guess

tol-stopping tolerance between interations

limit-maximum number of iterations

backtrack-perform backtrack line search if true, use step length 1 otherwise

Rho,gamma-backtrack parameters

'''

ret,var=split(df)

ret\_kkt, var\_kkt = kkt\_convert(df)

x=init

rho=Rho

delta=[tol]

i=0

print('Initial guess: ',x)

while (np.any(np.absolute(delta)>=tol))&(i<limit):

alpha=1

d=np.matmul(inv(var\_kkt),df\_kkt(x)) # From definition of newton

x\_k = x - alpha\*d

while (alpha>0.01)&(backtrack==True)&(f(x\_k[:-1])>(f(x[:-1])+alpha\*gamma\*np.matmul(d,df\_kkt(x)))):

alpha=alpha\*rho

x\_k = x - alpha\*d

print('alpha: ',alpha)

x\_k = x - alpha\*d

delta = x\_k-x

x=x\_k

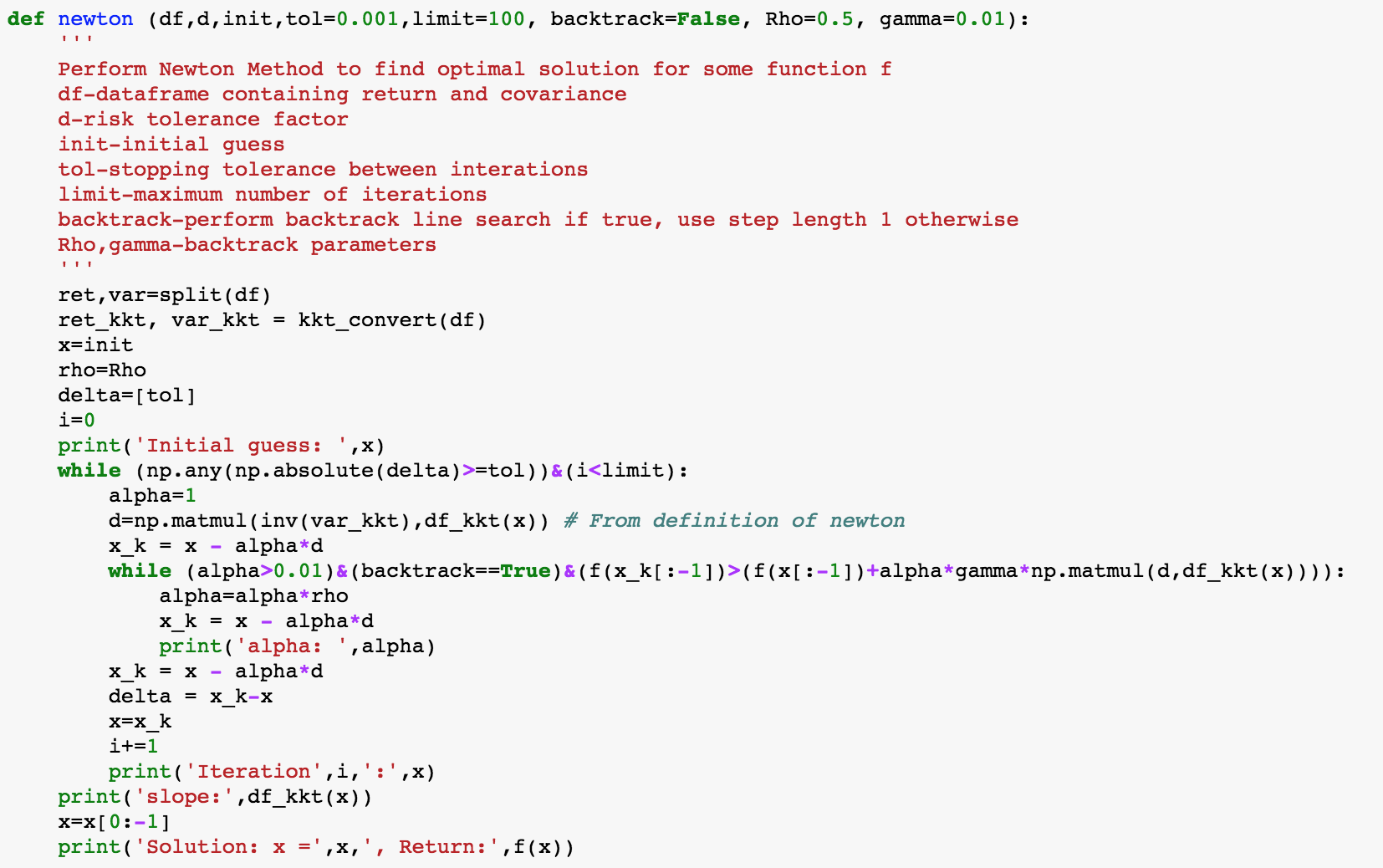
i+=1

print('Iteration',i,':',x)

print('slope:',df\_kkt(x))

x=x[0:-1]

print('Solution: x =',x,', Return:',f(x))



## A2 SDM

def steepest\_descent (df,d,init,tol=0.001,limit=100, backtrack=False, rho=0.5, gamma=0.01):

'''

Perform Steepest Descent Method to find optimal solution for some function f

df-dataframe containing return and covariance

d-risk tolerance factor

init-initial guess

tol-stopping tolerance between interations

limit-maximum number of iterations

backtrack-perform backtrack line search if true, use step length 1 otherwise

Rho,gamma-backtrack parameters

'''

ret,var=split(df)

ret\_kkt, var\_kkt = kkt\_convert(df)

x=init

delta=[tol]

i=0

print('Initial guess: ',x)

while (np.any(np.absolute(delta)>=tol))&(np.all(df\_kkt(x)<0))&(i<limit):

alpha=1

d=df\_kkt(x) # From definition of steepest descent

x\_k = x - alpha\*d

while (alpha>0.01)&(backtrack==True)&((f(x\_k[:-1]))>(f(x[:-1])+alpha\*gamma\*np.matmul(d,(df\_kkt(x))))):

alpha=alpha\*rho

x\_k = x - alpha\*d

print('alpha: ',alpha)

x\_k = x-alpha\*d

delta = x\_k-x

# print(delta)

x=x\_k

i+=1

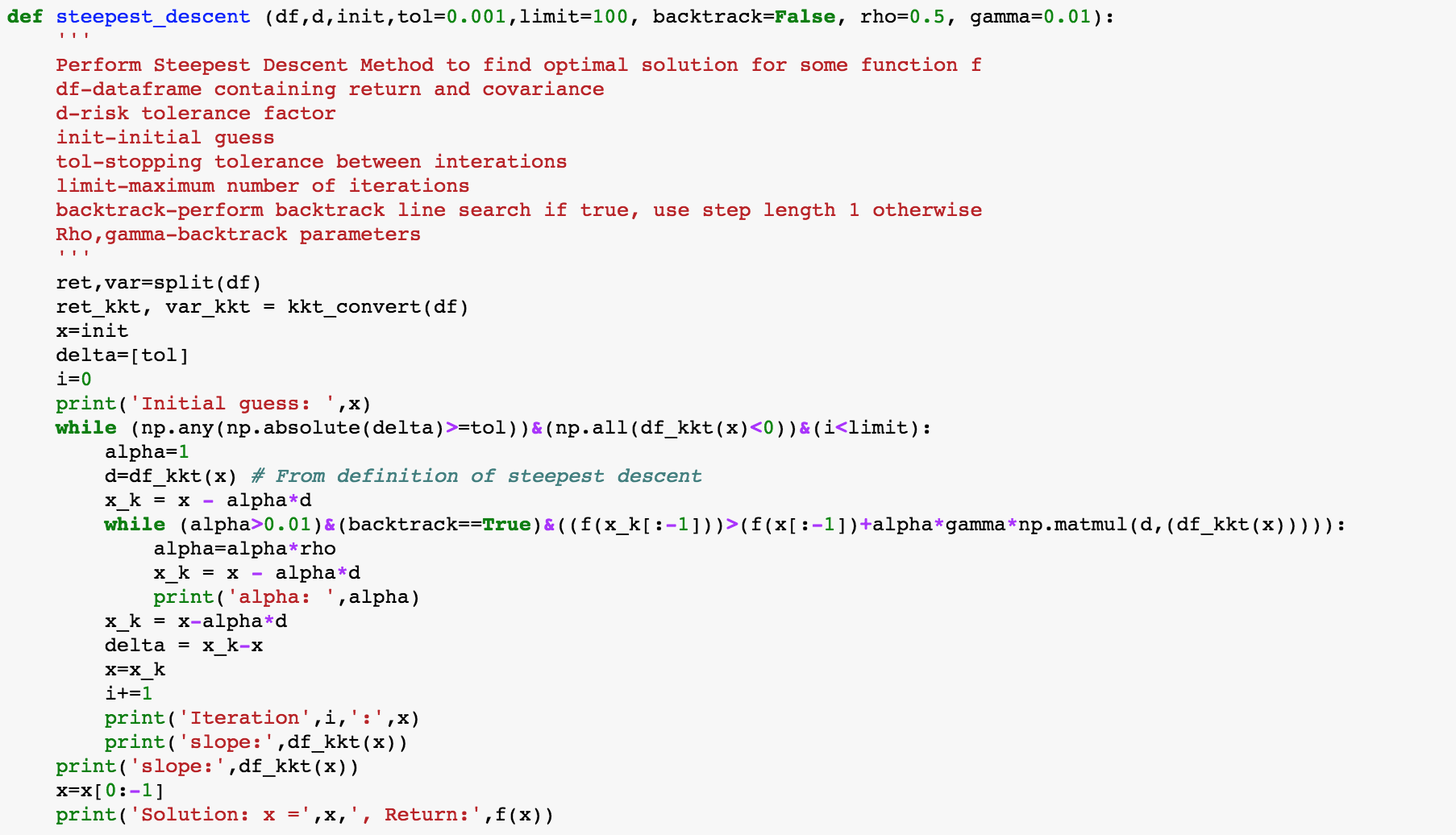
print('Iteration',i,':',x)

print('slope:',df\_kkt(x))

print('slope:',df\_kkt(x))

x=x[0:-1]

print('Solution: x =',x,', Return:',f(x))



## A3 BFGS

def BFGS (df,d,init,tol=0.001,limit=100, backtrack=False, rho=0.5, gamma=0.0001):

'''

Perform BFGS Quasi-Newton Method to find optimal solution for some function f

'''

ret,var=split(df)

ret\_kkt, var\_kkt = kkt\_convert(df)

x=init

H=inv(np.eye(1+len(var)))

I=np.eye(1+len(var))

delta=[tol]

i=0

print('Initial guess: ',x)

while (np.any(np.absolute(delta)>=tol))&(i<limit):

alpha=1

d=np.matmul(np.asarray(H),df\_kkt(x))

x\_k = x - alpha\*d

while (alpha>0.1)&(backtrack==True)&(f(x\_k[:-1])>(f(x[:-1])+alpha\*gamma\*np.matmul(d,df\_kkt(x)))):

alpha=alpha\*rho

x\_k = x - alpha\*d

print('alpha: ',alpha)

x\_k = x - alpha\*d

delta = x\_k-x

# find udpated H inverse

s=np.asmatrix(x\_k-x)

y=np.asmatrix(df\_kkt(x\_k)-df\_kkt(x))

a=np.asscalar(1/(y\*np.transpose(s)))

A=a\*(np.transpose(s)\*y)

B=a\*(np.transpose(y)\*s)

C=a\*(np.transpose(s)\*(s))

H=np.matmul(np.matmul((I-A),H),(I-B))+C

x=x\_k

i+=1

print('Iteration',i,':',x)

print('slope:',df\_kkt(x))

x=x[0:-1]

print('Solution: x =',x,', Return:',f(x))

