M693a_HW2

October 10, 2018

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from scipy import linalg as LA
        import time
        import random
        %matplotlib inline
```

Matteo Polimeno
MATH693a
Dr. Peter Blomgren
HW02

For this assignment, we try to minimize the Rosenbrock Function, by improving the code for the first HW assignment. We write down the function, its gradient, its hessian and also a function "p" to be used based on the Method implemented (either steepest descent or Newton). Also, we write down the derivative of the Rosenbrock function with respect to alpha and we call it $\phi'(\alpha)$.

```
In [2]: def f(x):
            rb = 100.*(x[1]-x[0]**2)**2+(1.-x[0])**2
            return rb

        def grad_f(x):
            df1 = 400.*x[0]*(x[0]**2-x[1])+2.*(x[0]-1.)
            df2 = 200.*(x[1]-x[0]**2)
            nabla = np.array([df1,df2])
            return nabla

        def hess_f(x):
            h11 = 1200.*x[0]**2-400.*x[1]+2.
            h12 = -400.*x[0]
            h21 = -400.*x[0]
            h22 = 200.
            hess = np.array([[h11,h12],[h21,h22]])
            return hess

        def invhess_f(x):
            inv = LA.solve(hess_f(x),grad_f(x))
            return inv
```

```
def p(x,method):
    if method=='Newton':
        p = -invhess_f(x)
    else:
        p = -grad_f(x)/(LA.norm(grad_f(x)))
    return p

def phi_prime(x,alpha,method):
    phip = grad_f(x+alpha*p(x,method)).dot(p(x,method))
    return phip
```

Now we write down the result of the interpolation obtained by using the cubic Hermite Polynomial seen in class.

```
In [3]: alpha = np.linspace(0.,1.,101)

        def sign(foo):  #own sign function to avoid return 0.
            if foo >= 0.:
                return 1.
            else:
                return -1.

        def H3(x,al,ah,alpha):
                H3 = (((1.+2*(alpha-al)/(ah-al))*((ah-alpha)/(ah-al))**2)*phi_(x,al,method)  #wr
                 + ((1.+2*(ah-alpha)/(ah-al))*((alpha-al)/(ah-al))**2)*phi_(x,ah,method)
                 + (alpha-al)*((ah-alpha)/(ah-al))**2*phi_prime(x,al,method)
                 + (alpha-ah)*((alpha-al)/(ah-al))**2*phi_prime(x,al,method))

        def interp(x,al,ah,method):  #return result of the interpolation
            d1 = phi_prime(x,al,method) + phi_prime(x,ah,method) - 3.*(f(x+al*p(x,method))-f(x+a
            d2 = sign(ah-al)*(np.sqrt(d1**2.-phi_prime(x,al,method)*phi_prime(x,ah,method)))
            akp1 = ah - (ah-al)*((phi_prime(x,ah,method)+d2-d1)/(phi_prime(x,ah,method)-phi_prim
            return akp1
```

Then, we write down the zoom function, following the lecture's slides.

```
In [4]: def zoom(x,al,ah,method):
            c1 = 1e-4
            c2 = 0.9
            while True:
                ajj = interp(x,al,ah,method)
                if (f(x+ajj*p(x,method)) > f(x)+c1*ajj*phi_prime(x,0.,method)) or (f(x+ajj*p(x,m
                    ah = ajj
                else:
                    if np.abs(phi_prime(x,ajj,method)) <= -c2*phi_prime(x,0.,method):
                        astar = ajj
                        break
                    if (phi_prime(x,ajj,method)*(ah-al)) >= 0.:
                        ah = al
```

```
            al = ajj
    return astar
```

Now we implement a code for the Strong Wolfe Conditions. If those are satisfied by our α value, it will be returned.

```
In [5]: def wolfe_strong(x,al,ah,amax,method): #strong wolfe conditions
            aii = np.array([al,ah])
            c1 = 1e-4
            c2 = 0.9
            ii = 1
            while True:
                if (f(x+aii[1]*p(x,method)) > f(x) + c1*aii[1]*f(x)) or (f(x+aii[1]*p(x,method))
                    anew = zoom(x,aii[0],aii[1],method)
                    break
                if np.abs(phi_prime(x,aii[1],method)) <= -c2*phi_prime(x,0.,method):
                    anew = aii[1]
                    break
                if phi_prime(x,aii[1],method) >= 0.:
                    anew = zoom(x,aii[1],aii[0],method)
                    break
                if (np.abs(aii[1]-aii[0]) < 1e-14) or (np.abs(aii[1]) < 1e-14):
                    anew = aii[1]
                    break
                aii[0] = aii[1]
                aii[1] = np.random.uniform(aii[1],amax)
                ii = ii + 1
            return anew
```

We finally write down an algorithm for the backtracking line search, implementing all the new conditions that we wrote above. We compare the steepest descent method and the Newton's method by running time and number of iterations.

```
In [22]: def backtrack(x,al,ah,amax,method):
             alpha_bar = 1.
             aij = np.zeros(10)
             tol = 1e-8
             jj = 1
             ii = 1
             xk = np.zeros([2,2])
             start = time.time()
             while (LA.norm(grad_f(x))) > tol:
                 a = wolfe_strong(x,al,ah,amax,method)
                 if method=='Newton':
                     ah = alpha_bar
                 else:
                     if ii==1:
                         ah = alpha_bar
                     else:
```

```
                    xk[:,jj-1] = x
                    ptkm1 = np.transpose(p(xk[:,jj-1],method))
                    gradkm1 = grad_f(xk[:,jj-1])
                    xk[:,jj] = x
                    ptk = np.transpose(p(xk[:,jj],method))
                    gradk = grad_f(xk[:,jj])
                    ah = ah*ptkm1.dot(gradkm1)/(ptk.dot(gradk))
            end = time.time()
            t = (end-start)
            xnew = x + a*p(x,method)
            x = xnew
            if ii <= 10:
                aij[ii-1] = a
            ah = alpha_bar
            ii = ii + 1
        print "First 10 alphas = " +  np.str(aij)
        print "Total number of iterations = " + np.str(ii)
        print "Minimun Found at = " + np.str(x)
        print "Value of function at minimum = " + np.str(f(x))
        print "Elapsed time for " + np.str(method) + " is t = " + np.str(t) + " sec"
```

```
In [23]: backtrack([1.2,1.2],0.,1.,5.,'Newton')
```

```
First 10 alphas = [1.         0.44356109 1.         1.         1.         1.
 1.         1.         0.         0.        ]
Total number of iterations = 9
Minimun Found at = [1. 1.]
Value of function at minimum = 1.3901228385074981e-22
Elapsed time for Newton is t = 0.013032913208 sec
```

```
In [24]: backtrack([1.2,1.2],0.,1.,5.,'Steepest Descent')
```

```
First 10 alphas = [0.10460801 0.01028286 0.00131497 0.00023855 0.00053122 0.00027096
 0.00056697 0.00027959 0.00056847 0.00027976]
Total number of iterations = 9923
Minimun Found at = [1.00000001 1.00000002]
Value of function at minimum = 9.658628923598997e-17
Elapsed time for Steepest Descent is t = 8.46759700775 sec
```

```
In [25]: backtrack([-1.2,1.],0.,1.,5.,'Newton')
```

```
First 10 alphas = [1.         0.13146358 1.         0.51012331 1.         1.
 1.         0.42004172 1.         0.62332568]
Total number of iterations = 23
Minimun Found at = [1. 1.]
Value of function at minimum = 1.2818989709841442e-30
Elapsed time for Newton is t = 0.088497877121 sec
```

```
In [26]: backtrack([-1.2,1.],0.,1.,5.,'Steepest Descent')

First 10 alphas = [0.19721409 0.01553408 0.0071383  0.00486584 0.01195968 0.0063378
 0.01465312 0.00713722 0.01465762 0.00720977]
Total number of iterations = 10353
Minimun Found at = [0.99999999 0.99999998]
Value of function at minimum = 9.667495090835293e-17
Elapsed time for Steepest Descent is t = 8.8970811367 sec
```

Once again, the Newton's method is really efficient and fast, however, the number of iterations for the steepest descent method has been cut off by a factor of over 50% compared to the algorithm implemented in HW1, and the running time has decreased, as well.