Caleb Logemann
MATH 565 Continuous Optimization
Homework 1

1. Problem 6
   Consider the steepest descent method with exact linear searches applied to the convex quadractic function (3.24). Using the properties given in this chapter, show that if the initial point is such that $x_0 - x^*$ is parallel to an eigenvector of Q, then the steepest descent method will find the solution in one step.

2. Problem 8
   Let Q be a postive definite symmetric matrix. Prove that for any vector $\underline{X}$, we have

$$\frac{\left(\underline{x}^T\underline{x}\right)^2}{\left(\underline{x}^T\underline{\underline{Q}}\underline{x}\right)\left(\underline{x}^T\underline{\underline{Q}}^{-1}\underline{x}\right)} \geq \frac{4\lambda_n\lambda_1}{\left(\lambda_n+\lambda_1\right)^2}$$

   where λ_n and λ_1 are, respectively, the largest and smallest eigenvalues of $\underline{\underline{Q}}$. (This relation, which is known as the Kantorovich inequality, can be used to deduce (3.29) from (3.28).)

   Proof. □

3. Problem 12

4. For the quadractic function

$$f(\underline{x}) = \frac{1}{2}\underline{x}^T\underline{\underline{B}}\underline{x} - \underline{x}^T\underline{b}$$

   where $\underline{\underline{B}} \in \mathbb{R}^{n\times n}$ is symmetric positive definite, show that the Newton search direction with $\alpha = 1$ satisfies the sufficient decrease assumption (3.4) for any $c_1 \leq \frac{1}{2}$ and the curvature conditions (3.5) for any $c_2 > 0$.

5. Consider the function:

$$f(\underline{x}) = 20(x_2 - x_1^2)^2 + (1-x_1)^2$$

   Write a MATLAB steepest descent code to find the minimizer of this function. The function be in the form

```
xstar = SteepestDescent(f, x0, TOL, MaxIters)
```

   Use $\underline{x_0} = (1.2, 1.2)^T$. Use an exact line search to find α. You can use your $1D$ rootfinding code from Homework 1 to compute the exact line search solution α. Plot the contour lines of f and superimport the various guesses (each connected to the next via a line segment) made by the steepest descent algorithm. Produce a table of your approximations and the errors (if there are many iterations, you can just show the first 10 iterations and the last 10 iterations).

   In order to compute the exact line search I implemented Newton's Method in one dimension to compute roots of $\frac{\mathrm{d}\phi}{\mathrm{d}\alpha}$. Note that Newton's Method requires $\frac{\mathrm{d}^2\phi}{\mathrm{d}\alpha^2}$ as well. The other problem with this approach is that Newton's Method is only finding roots of $\frac{\mathrm{d}\phi}{\mathrm{d}\alpha}$ and not finding the absolute minimum of $\phi(\alpha)$. It may in fact find only a local minimum or even a maximum. However this method is right most of the time and allows the Steepest Descent method to converge in this case. My implementation of Newton's Method in one dimension is shown below.

```
import numpy as np
def newton1D(g, gd, x0, TOL, MaxIter):
    x = np.zeros(MaxIter+1)
    x[0] = x0
    nIter = 0
    mstop = 1

    gx = g(x[nIter])
    while nIter < MaxIter and mstop:
        gdx = gd(x[nIter])
        delta = -gx/gdx
        x[nIter+1] = x[nIter] + delta
        nIter+=1
        gx = g(x[nIter])
        if abs(gx) < TOL and abs(delta) < TOL:
            mstop = 0
    return x[:nIter + 1]
```

The next script implements and uses the Steepest Descent algorithm to find the global minimum of f.

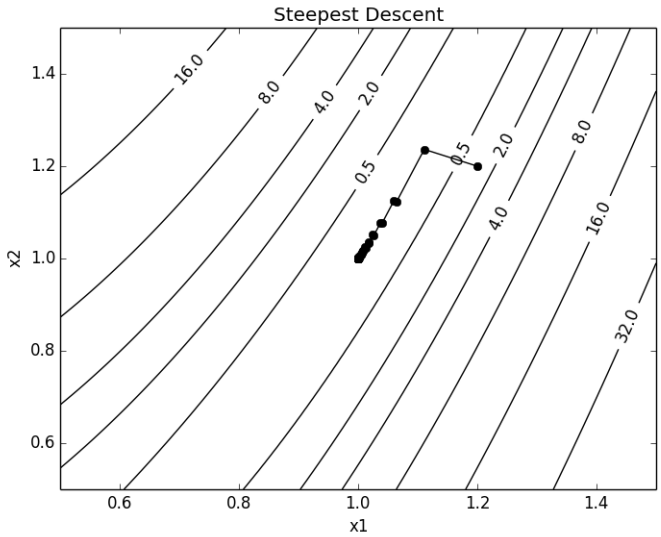
```
import numpy as np
import matplotlib.pyplot as plt
execfile('newton1d.py')
def steepestDescent(f, gradf, phid, phidd, x0, TOL, MaxIter):
    x = np.zeros([MaxIter+1, x0.size])
    x[0] = x0
    nIter = 0
    mstop = 1

    while nIter < MaxIter and mstop:
        p = -gradf(x[nIter])
        g = lambda a: phid(a, x[nIter], p)
        gd = lambda a: phidd(a, x[nIter], p)
        sol = newton1D(g, gd, 0, TOL, MaxIter)
        #if sol.size >= MaxIter:
            # print(nIter)
            # print(p)
            # print(x[nIter])
            # print(sol[-1])
            # raise Exception('Search for alpha did not converge')
        # alpha = backtrace(f, gradf, x[nIter], p)
        alpha = sol[-1]
        # print(nIter)
        # print(alpha)
        delta = alpha * p
        x[nIter+1] = x[nIter] + delta
        nIter+=1
        if sum(abs(delta)) < TOL and sum(abs(p)) < TOL:
            mstop = 0

    return x[:nIter+1]

def plotResults(f, sol, title):
    minX0 = .5
    maxX0 = 1.5
    minX1 = .5
    maxX1 = 1.5
    meshSize = 100
```

```
    x0list = np.linspace(minX0, maxX0, meshSize)
    x1list = np.linspace(minX1, maxX1, meshSize)
    X0, X1 = np.meshgrid(x0list, x1list)
    Z = np.zeros([meshSize,meshSize])
    for i in range(meshSize):
        for j in range(meshSize):
            Z[i,j] = f([X0[i,j], X1[i,j]])

    print('plotting')
    plt.figure()
    levels = [0.5, 2, 4, 8, 16, 32]
    contour = plt.contour(X0, X1, Z, levels, colors='k')
    plt.plot(sol[:,0], sol[:,1], '-k')
    plt.plot(sol[:,0], sol[:,1], 'ko')
    plt.clabel(contour, colors='k', fmt='%2.1f', fontsize=12)
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.title(title)
    plt.show()

f = lambda x: 20*np.square(x[1] - np.square(x[0])) + np.square(1 - x[0])
def gradf(x):
    dx0 = -80*x[0]*(x[1] - np.square(x[0])) - 2*(1 - x[0])
    dx1 = 40*(x[1] - np.square(x[0]))
    return np.array([dx0, dx1])

phid = lambda alpha, x, p: 40*(x[1] + alpha*p[1] - np.square(x[0] + alpha*p[0]))*(p[1]
    ↪ - 2*p[0]*(x[0] + alpha*p[0])) - 2*(1 - x[0] - alpha*p[0])*p[0]
phidd = lambda alpha, x, p: 40*np.square(p[1]) + 2*p[0]*(-80*p[1]*x[0] + p[0]*(1-40*x
    ↪ [1]-120*p[1]*alpha + 120*np.square(x[0] + p[0]*alpha)))

x0 = np.array([1.2, 1.2])
TOL = 1e-10
MaxIter = 1000
solSteepestDescent = steepestDescent(f, gradf, phid, phidd, x0, TOL, MaxIter)
plotResults(f, solSteepestDescent, 'Steepest Descent')
```

The following table lists the first 10 and last 10 approximations. The error in these tables is also the function values at the approximation as the minimum value is zero.

k	$(x_1)_k$	$(x_2)_k$	e_k
0	1.2	1.2	1.19199999999e+00
1	1.11100773654	1.23644734339	1.24116880777e-02
2	1.06441606231	1.12268600547	6.26939535114e-03
3	1.05988566879	1.12454145675	3.61432219908e-03
4	1.04015601545	1.07636821985	2.22995866730e-03
5	1.03761784087	1.07740774527	1.42656176856e-03
6	1.02655884997	1.05040537581	9.38985518567e-04
7	1.02496393975	1.05105858136	6.28349485527e-04
8	1.01809348915	1.03428323113	4.26932414201e-04
9	1.01703824335	1.03471541371	2.92732528178e-04
10	1.01253992599	1.02373202217	2.02555033610e-04
⋮	⋮	⋮	⋮
117	1.00000000023	1.00000000047	5.68480731488e-20
118	1.00000000018	1.00000000033	4.07321772830e-20
119	1.00000000017	1.00000000034	2.91848895231e-20
120	1.00000000012	1.00000000024	2.09013306391e-20
121	1.00000000012	1.00000000024	1.49689263278e-20
122	1.00000000009	1.00000000017	1.07321945388e-20
123	1.00000000008	1.00000000017	7.69439383459e-21
124	1.00000000006	1.00000000012	5.52176221112e-21
125	1.00000000006	1.00000000012	3.96265903408e-21
126	1.00000000004	1.00000000008	2.84303547930e-21

The following contour plot was also produced.



6. Consider the function:

$$f(\underline{x}) = 20(x_2 - x_1^2)^2 + (1 - x_1)^2$$

Write a MATLAB Newton descent code to find the minimizer of this function. The function be in the form

```
xstar = NewtonDescent(f, x0, TOL, MaxIters)
```

Use $\underline{x_0} = (1.2, 1.2)^T$. Use $\alpha = 1$. Use the backslash operator to invert the appropriate matrices. Plot the contour lines of f and superimport the various guesses (each connected to the next via a line segment) made by the Newton descent algorithm. Produce a table of your approximations and the errors (if there are many iterations, you can just show the first 10 iterations and the last 10 iterations).

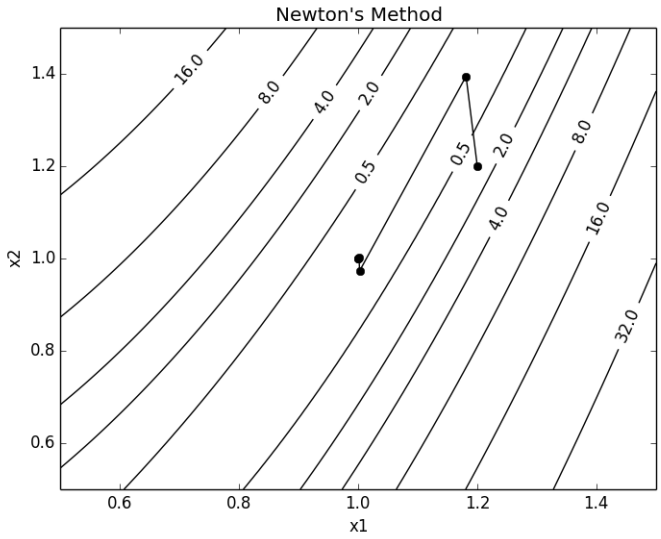
The following script implements and runs Newton's Method in multi dimensions to find the minimum of f. Note that this script uses the plotResults function from the script in the previous problem.

```
import numpy as np
import matplotlib.pyplot as plt
def newtonMultiD(f, gradf, hessianf, x0, TOL, MaxIter):
    x = np.zeros([MaxIter+1, x0.size])
    x[0] = x0
    nIter = 0
    mstop = 1
    gradfx = gradf(x0)
    while nIter < MaxIter and mstop:
        p = -np.dot(np.linalg.inv(hessianf(x[nIter])), gradfx)
        x[nIter+1] = x[nIter] + p
        nIter+=1
        gradfx = gradf(x[nIter])
        if sum(np.square(p)) < TOL and sum(np.square(gradfx)) < TOL:
            mstop = 0

    return x[:nIter+1]

f = lambda x: 20*np.square(x[1] - np.square(x[0])) + np.square(1 - x[0])
def gradf(x):
    dx0 = -80*x[0]*(x[1] - np.square(x[0])) - 2*(1 - x[0])
    dx1 = 40*(x[1] - np.square(x[0]))
    return np.array([dx0, dx1])

def hessianf(x):
    dx0x0 = 2 + 240*np.square(x[0]) - 80*x[1]
    dx0x1 = -80*x[0]
    dx1x1 = 40
    return np.array([[dx0x0, dx0x1], [dx0x1, dx1x1]])

x0 = np.array([1.2, 1.2])
TOL = 1e-10
MaxIter = 1000
solNewton = newtonMultiD(f, gradf, hessianf, x0, TOL, MaxIter)
plotResults(f, solNewton, 'Newton\'s Method')
```

The following table describes the list of approximations and errors. Note that the error at step k is equal to the function value at $(x_1)_k$ and $(x_2)_k$, because the minimum value is zero.

k	$(x_1)_k$	$(x_2)_k$	e_k
0	1.2	1.2	1.1919999999
1	1.181132075471	1.39471698113	3.2811363464e-02
2	1.002543096882	0.97319863783	2.0351041753e-02
3	1.001425625865	1.00285203539	2.0324402951e-06
4	1.000000071205	0.99999811020	8.2602301879e-11
5	1.000000000005	1.00000000001	3.3499320311e-23

The following contour plot was also produced.


Newton's Method
16.0
8.0
4.0
2.0
0.5
0.5
2.0
4.0
8.0
16.0
32.0
1.4
1.2
1.0
0.8
0.6
x2
0.6
0.8
1.0
1.2
1.4
x1