University of Texas at Austin, Cockrell School of Engineering

Data Mining – EE 380L

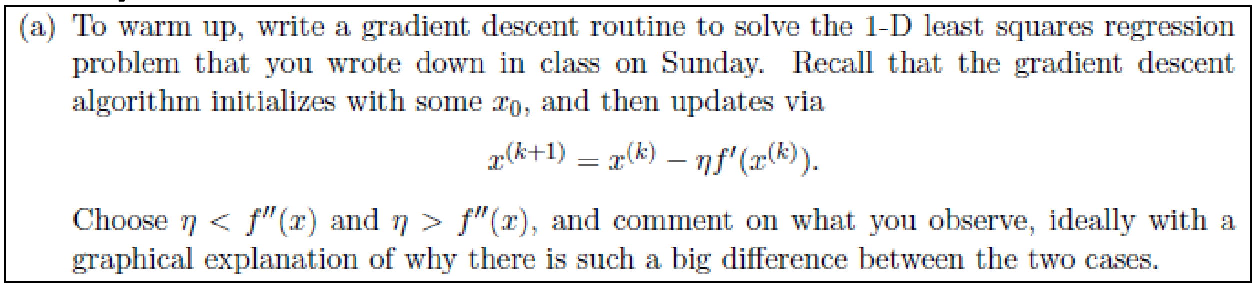
Problem Set # 1

February 08, 2016

Gabrielson Eapen

EID: EAPENGP

Q6 Part a]



(a) To warm up, write a gradient descent routine to solve the 1-D least squares regression problem that you wrote down in class on Sunday. Recall that the gradient descent algorithm initializes with some x_0, and then updates via

$$x^{(k+1)} = x^{(k)} - \eta f'(x^{(k)}).$$

Choose $\eta < f''(x)$ and $\eta > f''(x)$, and comment on what you observe, ideally with a graphical explanation of why there is such a big difference between the two cases.

Loss Function from class for

$f(\beta) = \frac{5}{2}\beta^2 + 4.975\beta + 2.48$	$f'(\beta) = 5\beta + 4.975$	$f''(\beta) = 5$

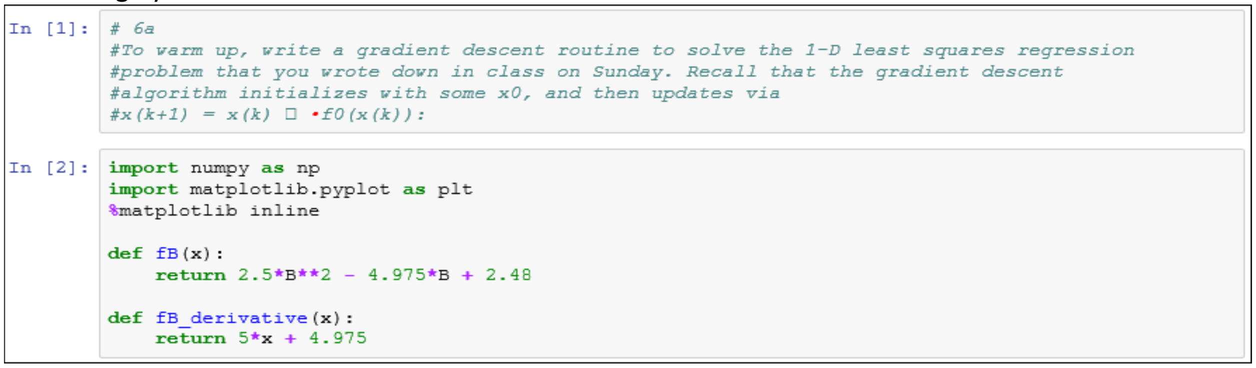
Solving for minimum using $f'(\beta) = 0$ (closed form)

$5\beta + 4.975 = 0$

$5\beta = -4.975$

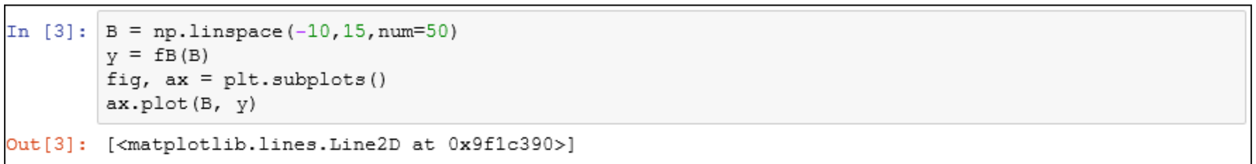
$\beta = -0.995$

Now using Python and Iterative Gradient Descent

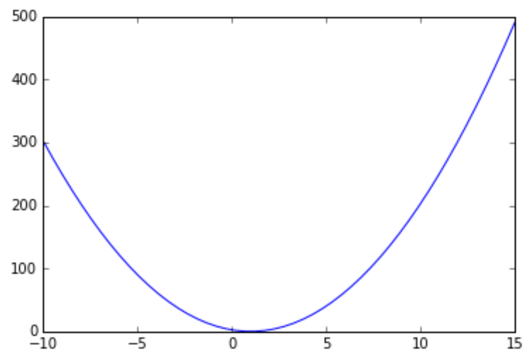


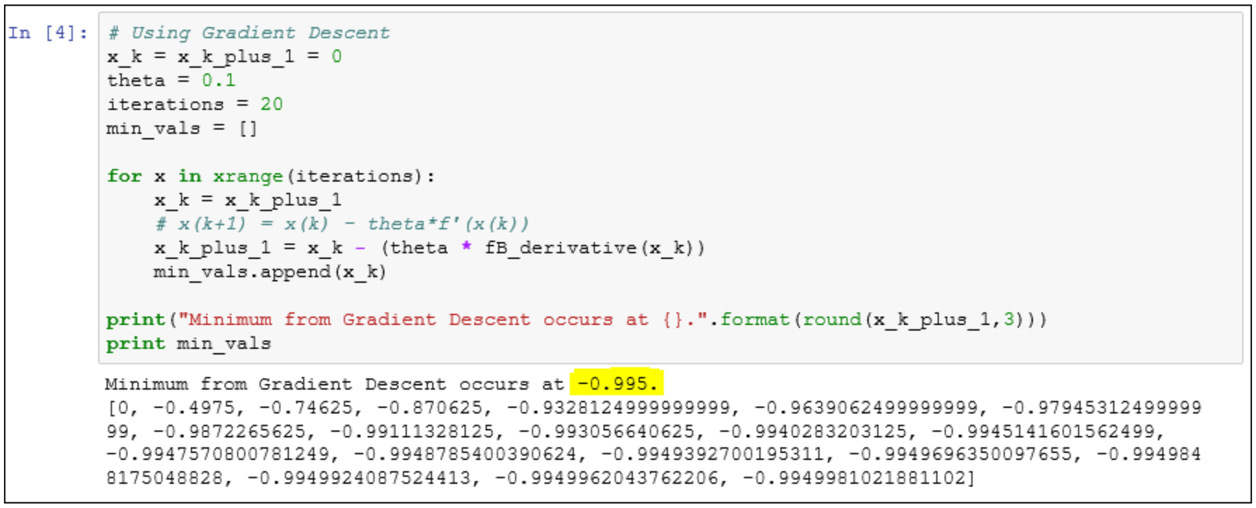
```
In [1]: # 6a
        #To warm up, write a gradient descent routine to solve the 1-D least squares regression
        #problem that you wrote down in class on Sunday. Recall that the gradient descent
        #algorithm initializes with some x0, and then updates via
        #x(k+1)  = x(k) □ •f0(x(k)):
```

```
In [2]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

        def fB(x):
            return 2.5*B**2 - 4.975*B + 2.48

        def fB_derivative(x):
            return 5*x + 4.975
```



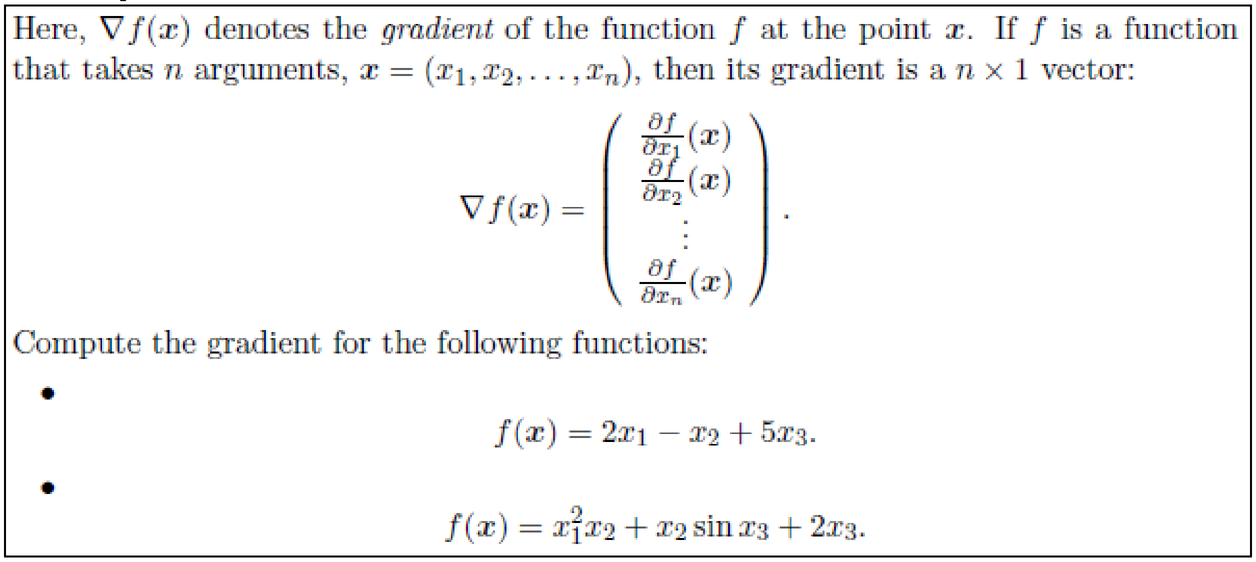
```
In [3]: B = np.linspace(-10,15,num=50)
        y = fB(B)
        fig, ax = plt.subplots()
        ax.plot(B, y)

Out[3]: [<matplotlib.lines.Line2D at 0x9f1c390>]
```





```
In [4]: # Using Gradient Descent
        x_k = x_k_plus_1 = 0
        theta = 0.1
        iterations = 20
        min_vals = []

        for x in xrange(iterations):
            x_k = x_k_plus_1
            # x(k+1) = x(k) - theta*f'(x(k))
            x_k_plus_1 = x_k - (theta * fB_derivative(x_k))
            min_vals.append(x_k)

        print("Minimum from Gradient Descent occurs at {}.".format(round(x_k_plus_1,3)))
        print min_vals

        Minimum from Gradient Descent occurs at -0.995.
        [0, -0.4975, -0.74625, -0.870625, -0.9328124999999999, -0.9639062499999999, -0.97945312499999
        99, -0.9872265625, -0.99111328125, -0.993056640625, -0.9940283203125, -0.994514160156249 9,
        -0.9947570800781249, -0.9948785400390624, -0.9949392700195311, -0.9949696350097655, -0.994984
        8175048828, -0.9949924087524413, -0.9949962043762206, -0.9949981021881102]
```

Q6 Part b]



Here, $\nabla f(\boldsymbol{x})$ denotes the *gradient* of the function f at the point $\boldsymbol{x}$. If f is a function that takes n arguments, $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$, then its gradient is a $n \times 1$ vector:

$$\nabla f(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1}(\boldsymbol{x}) \\ \frac{\partial f}{\partial x_2}(\boldsymbol{x}) \\ \vdots \\ \frac{\partial f}{\partial x_n}(\boldsymbol{x}) \end{pmatrix}.$$

Compute the gradient for the following functions:

- $$f(\boldsymbol{x}) = 2x_1 - x_2 + 5x_3.$$
- $$f(\boldsymbol{x}) = x_1^2 x_2 + x_2 \sin x_3 + 2x_3.$$

$f(x) = 2x_1 - x_2 + 5x_3$

$$\frac{\partial f}{\partial x_1}(x) = 2 \qquad \frac{\partial f}{\partial x_2}(x) = -1 \qquad \frac{\partial f}{\partial x_3}(x) = 5$$

$$\nabla f(x) = \begin{pmatrix} 2 \\ -1 \\ 5 \end{pmatrix}$$

$f(x) = {x_1}^2 x_2 + x_2 \sin x_3 + 2x_3$

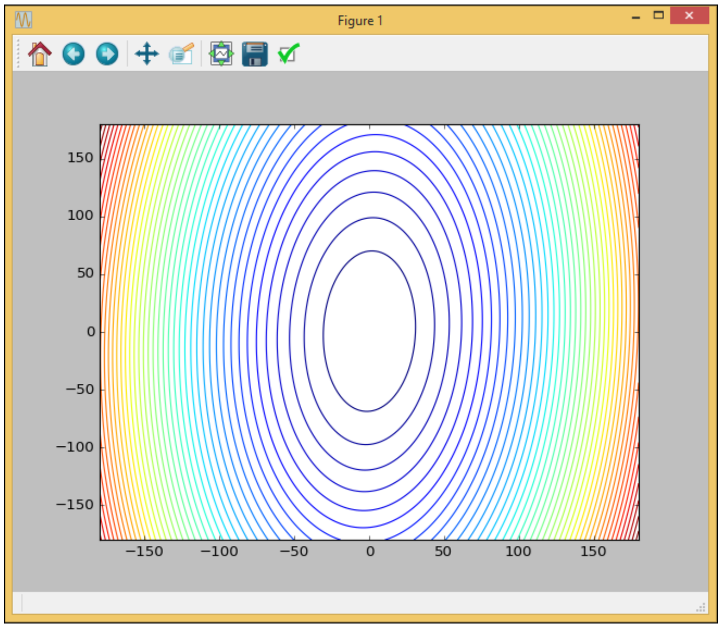
$$\frac{\partial f}{\partial x_1}(x) = 2\, x_1 x_2 \qquad \frac{\partial f}{\partial x_2}(x) = {x_1}^2 + \sin x_3 \qquad \frac{\partial f}{\partial x_3}(x) = x_2 \cos x_3 + 2$$

$$\nabla f(x) = \begin{pmatrix} 2\, x_1 x_2 \\ {x_1}^2 + \sin x_3 \\ x_2 \cos x_3 + 2 \end{pmatrix}$$

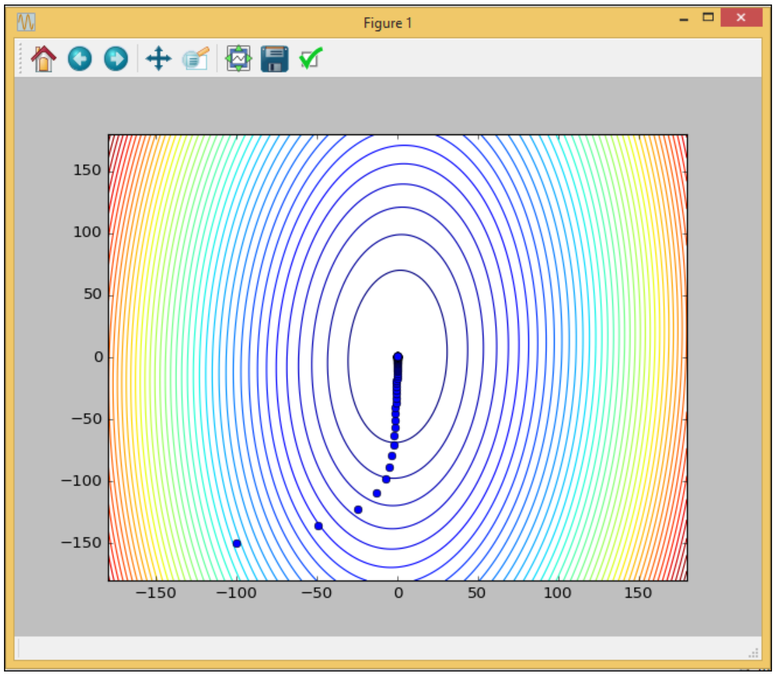
Q6 Part c]

(c) Have a look at the IPython notebook `2D Quadratic Gradient Descent`. Generate three (3) random examples, and while keeping the example fixed, try to find values of η, the step size, for which gradient descent converges, and for which it diverges.

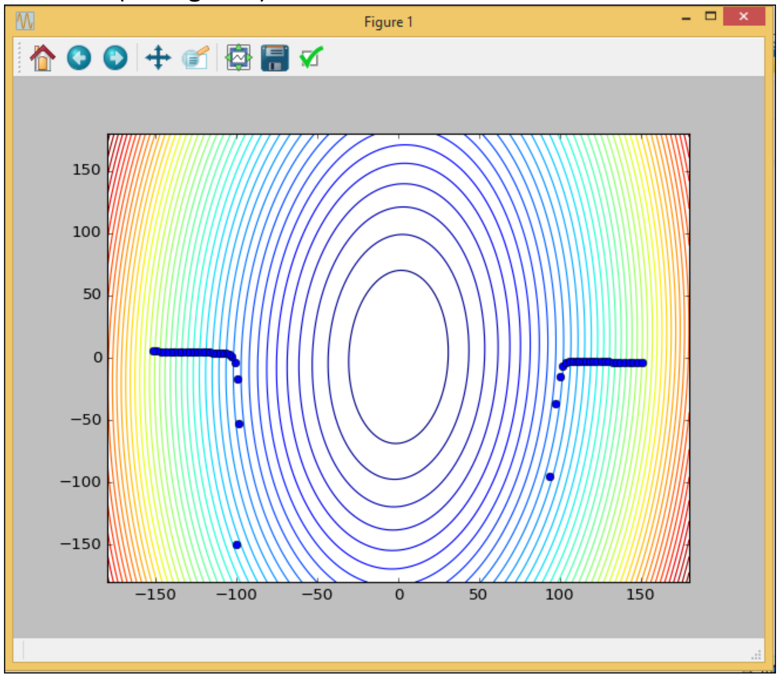
<u>Example 1</u>



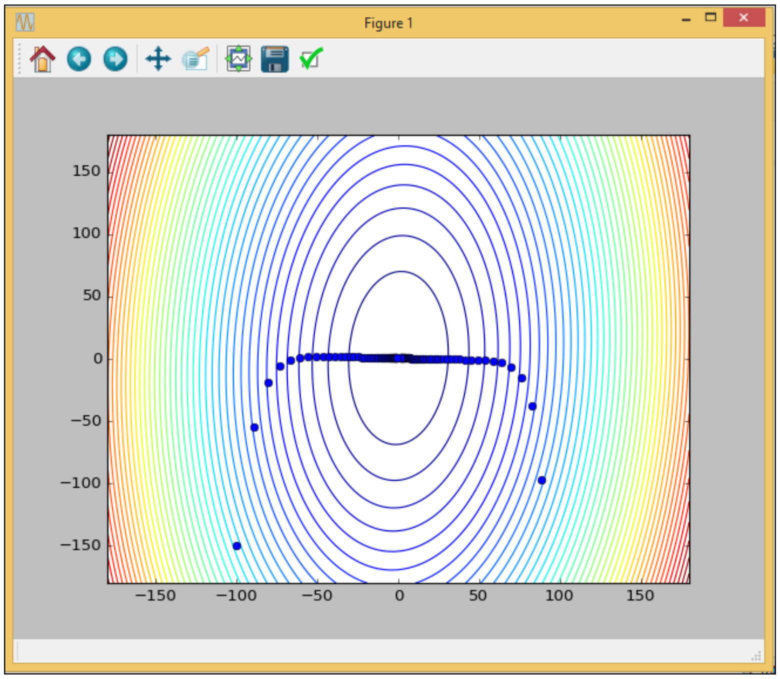
Iterations: 100
Start from (-100,-150)
eta: 0.1 (Convergence)



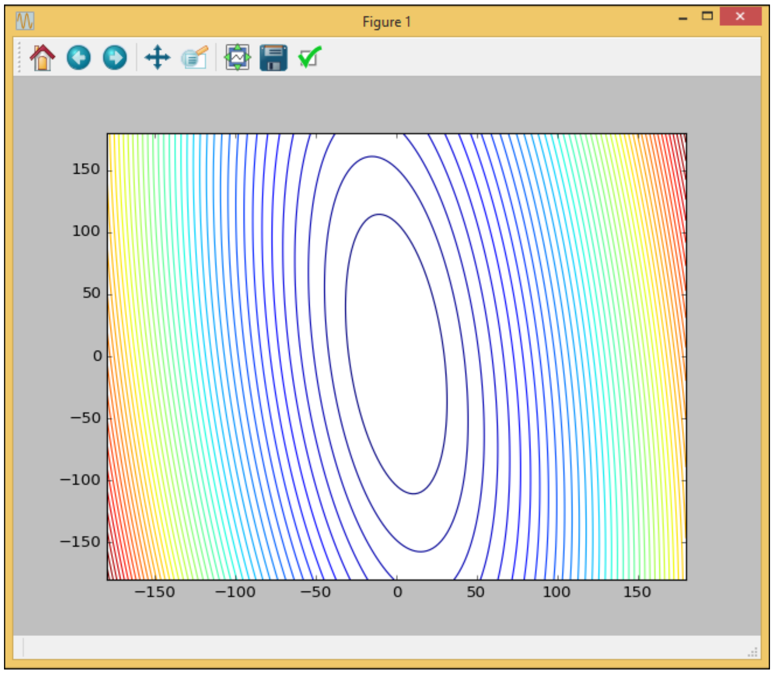
Iterations: 50
Start from (-100,-150)
eta: **0.38** (Divergence)



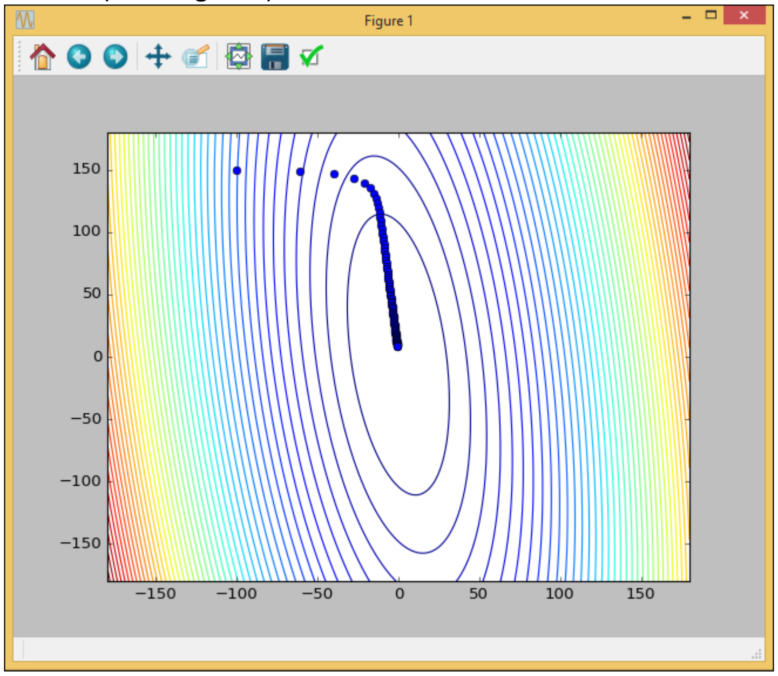
Iterations: 100
Start from (-100,-150)
eta: **0.37** (Overshoots but Convergence)



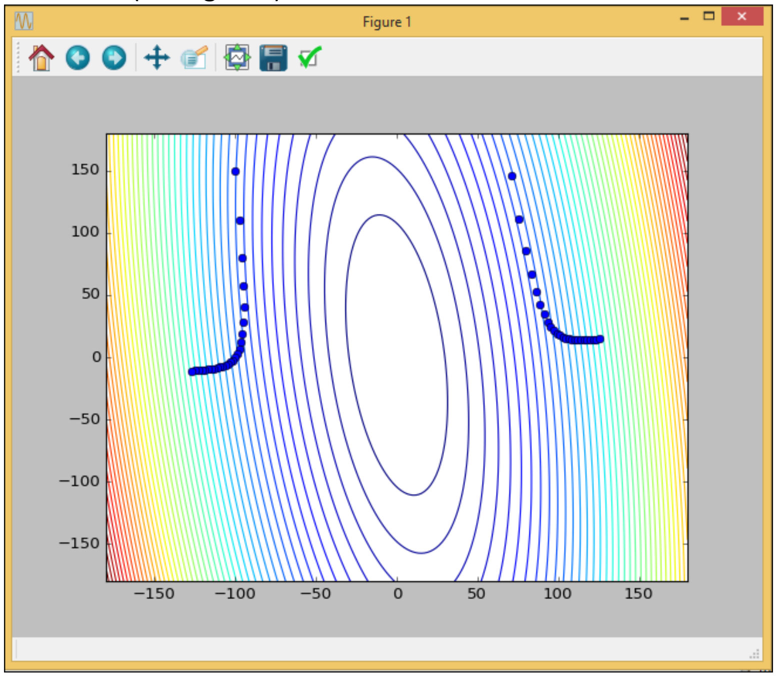
Example 2



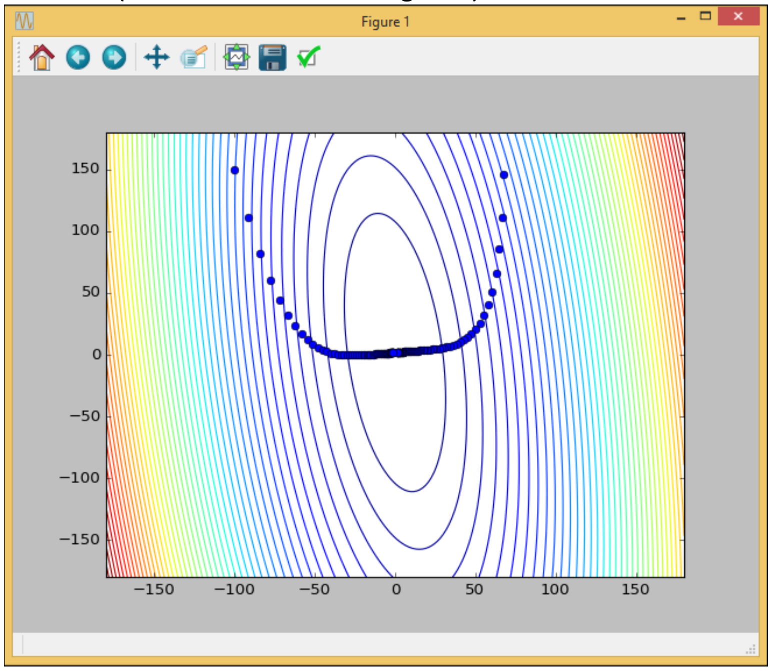
Iterations: 100
Start from (-100,150)
eta: 0.1 (Convergence)



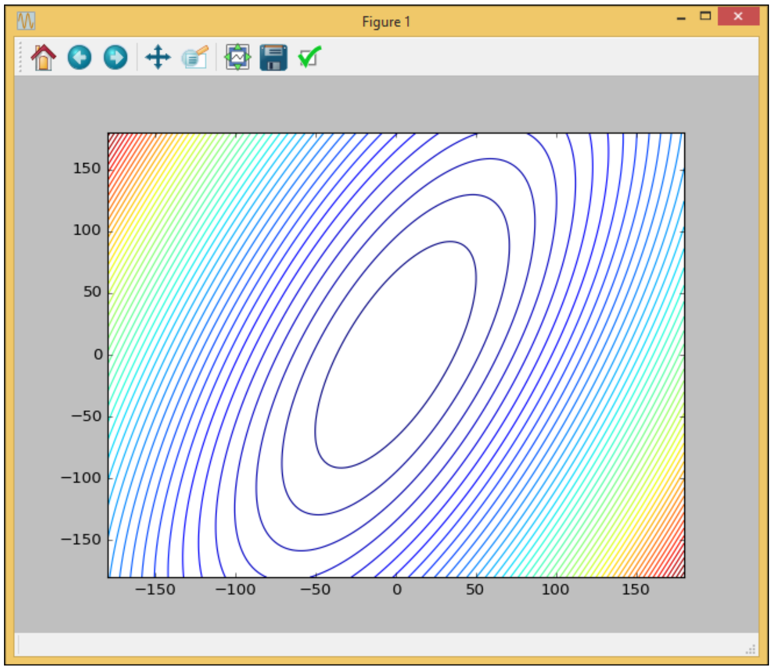
Iterations: 50
Start from (-100,150)
eta: **0.438** (Divergence)



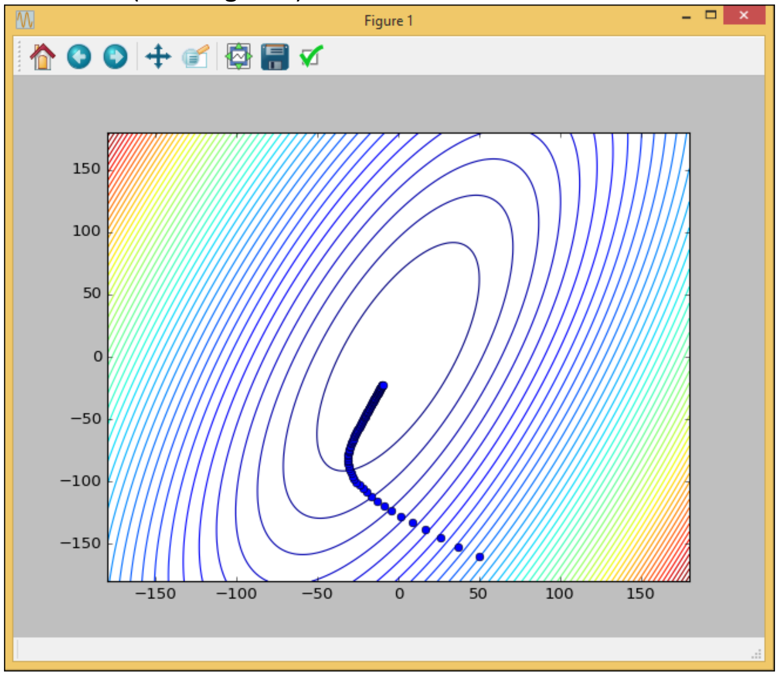
Iterations: 150
Start from (-100,150)
eta: **0.43** (Overshoots but Convergence)



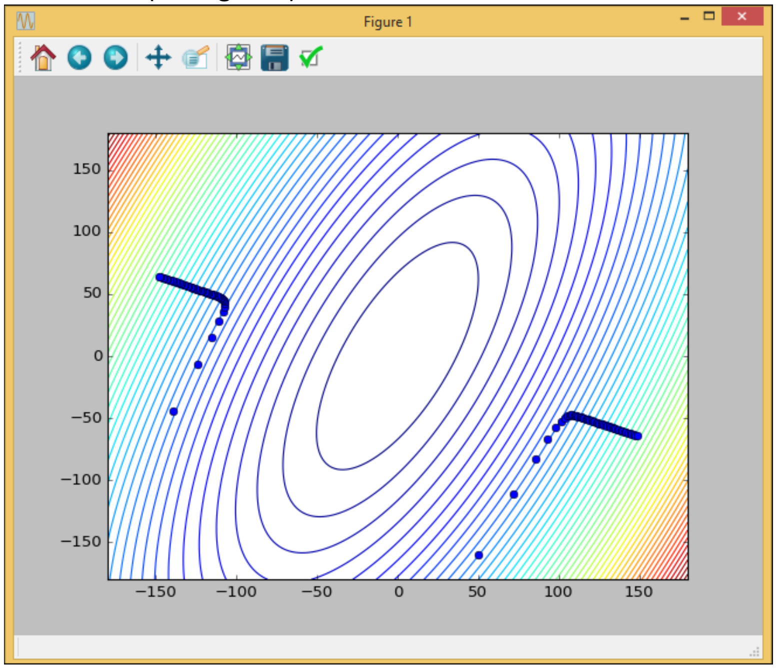
Example 3



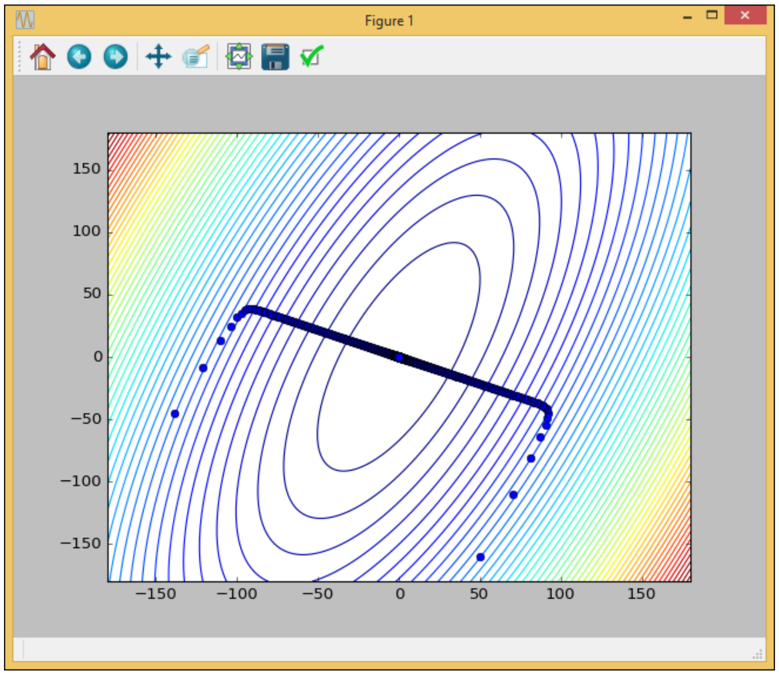
Iterations: 100
Start from (50,-160)
eta: 0.001 (Convergence)



Iterations: 100
Start from (50,-160)
eta: **0.1468** (Divergence)

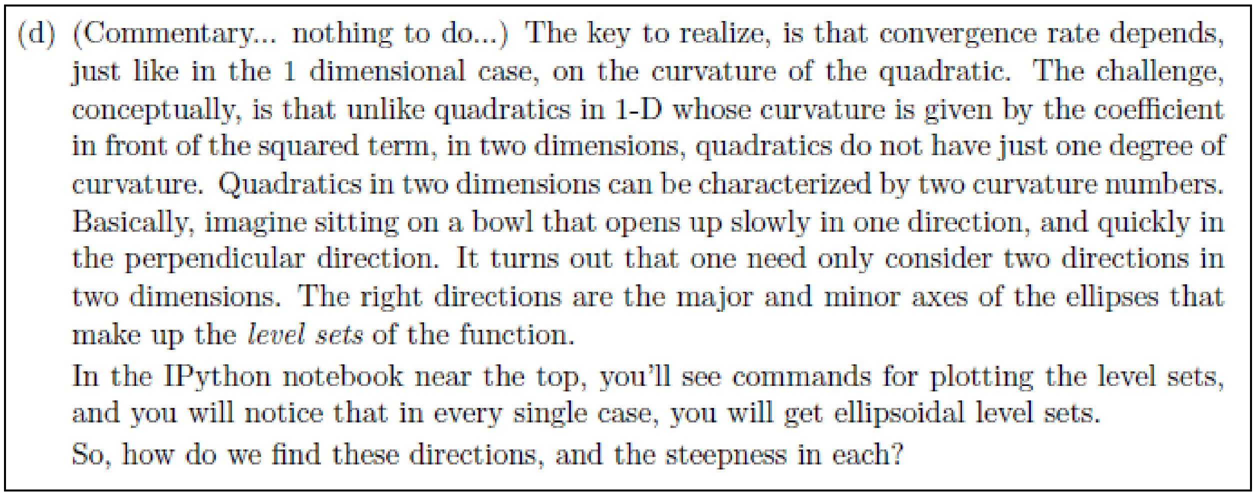


Iterations: 1000
Start from (50,-160)
eta: **0.1461** (Overshoots but Convergence)



Example 4

Q6 Part d]



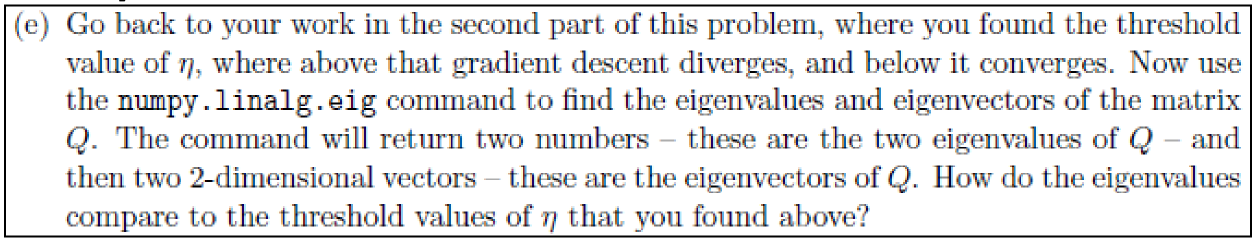
(d) (Commentary... nothing to do...) The key to realize, is that convergence rate depends, just like in the 1 dimensional case, on the curvature of the quadratic. The challenge, conceptually, is that unlike quadratics in 1-D whose curvature is given by the coefficient in front of the squared term, in two dimensions, quadratics do not have just one degree of curvature. Quadratics in two dimensions can be characterized by two curvature numbers. Basically, imagine sitting on a bowl that opens up slowly in one direction, and quickly in the perpendicular direction. It turns out that one need only consider two directions in two dimensions. The right directions are the major and minor axes of the ellipses that make up the *level sets* of the function.

In the IPython notebook near the top, you'll see commands for plotting the level sets, and you will notice that in every single case, you will get ellipsoidal level sets.

So, how do we find these directions, and the steepness in each?

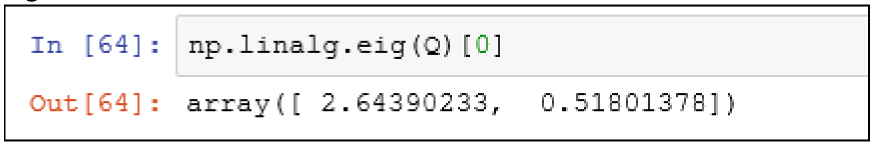
Yes, our results in part c validate that the major axis of the ellipses dictate the least curvature and the direction in which the gradient descent tries to approach the minimum point so long as the step size is appropriate. The gradient of the curvature determines the steepness similar to the 1-D case. This was very interesting to observe in each example from part C. I was looking at https://www.math.ucla.edu/~jteran/270c.1.11s/notes_wk4.pdf and it looks like if the initial guess is an eigen vector, then the search direction will align perfectly with the solution and steepest descent will converge exactly in the one iteration.

Q6 Part e]

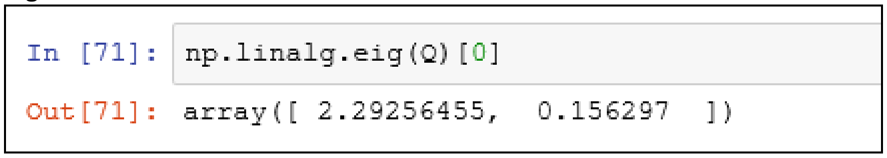

(e) Go back to your work in the second part of this problem, where you found the threshold value of η, where above that gradient descent diverges, and below it converges. Now use the `numpy.linalg.eig` command to find the eigenvalues and eigenvectors of the matrix Q. The command will return two numbers – these are the two eigenvalues of Q – and then two 2-dimensional vectors – these are the eigenvectors of Q. How do the eigenvalues compare to the threshold values of η that you found above?

In each of the examples in part c, it seems that the threshold eta value that converged was less than the largest eigenvalue. Anything higher than the largest eigen value caused divergence

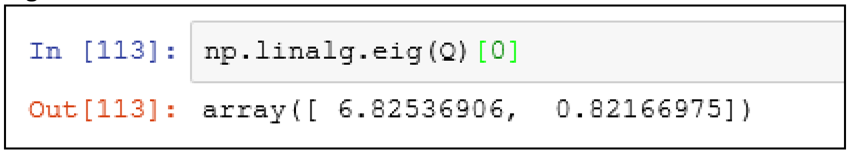
Example 1:
Eigen Values:


```
In [64]: np.linalg.eig(Q)[0]
Out[64]: array([ 2.64390233,  0.51801378])
```

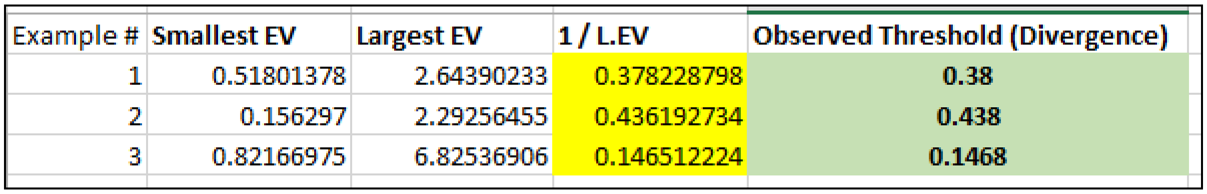
Example 2:
Eigen Values:


```
In [71]: np.linalg.eig(Q)[0]
Out[71]: array([ 2.29256455,  0.156297  ])
```

Example 3:
Eigen Values:

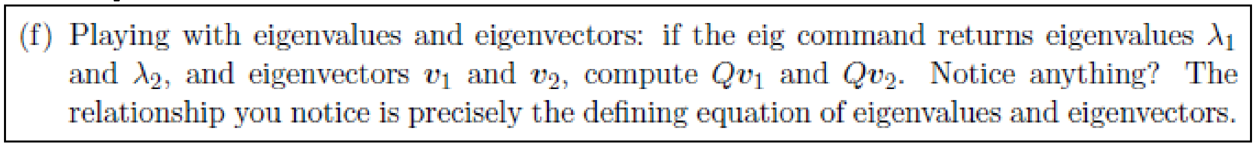

```
In [113]: np.linalg.eig(Q)[0]
Out[113]: array([ 6.82536906,  0.82166975])
```

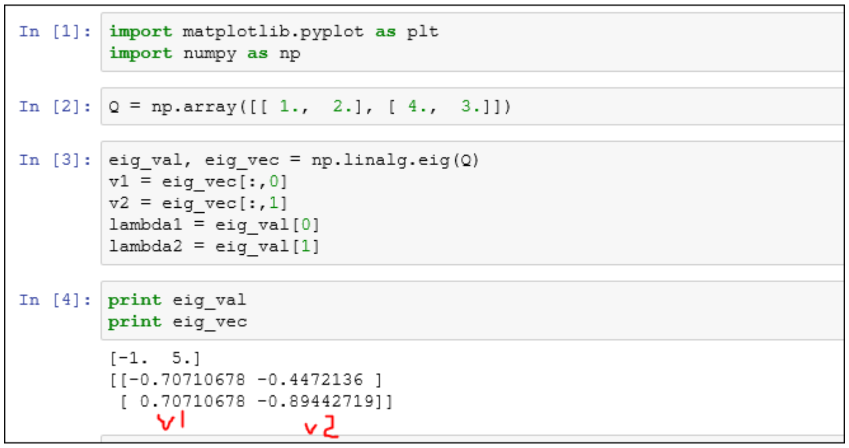
Results:

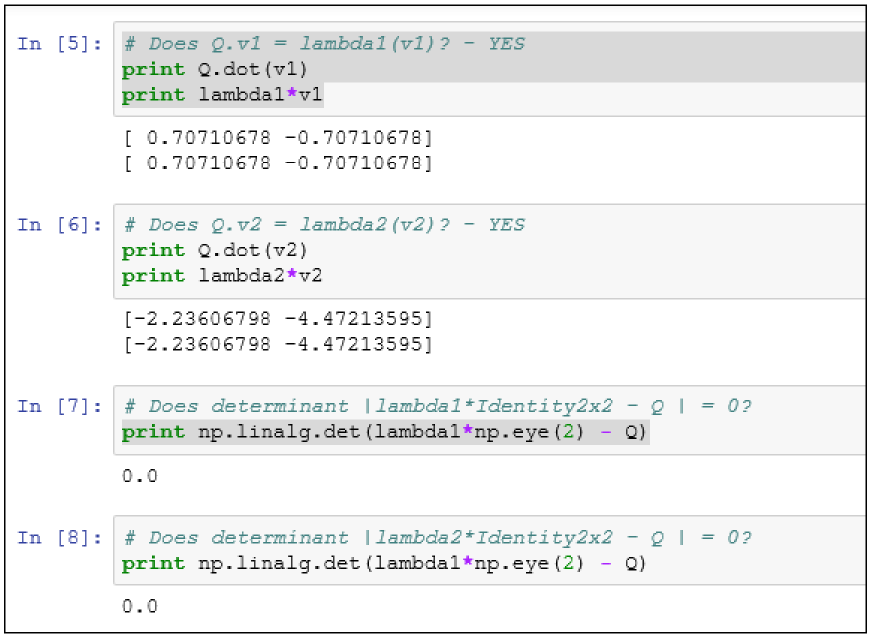


Example #	Smallest EV	Largest EV	1 / L.EV	Observed Threshold (Divergence)
1	0.51801378	2.64390233	0.378228798	**0.38**
2	0.156297	2.29256455	0.436192734	**0.438**
3	0.82166975	6.82536906	0.146512224	**0.1468**

Q6 Part f]


(f) Playing with eigenvalues and eigenvectors: if the eig command returns eigenvalues λ_1 and λ_2, and eigenvectors v_1 and v_2, compute Qv_1 and Qv_2. Notice anything? The relationship you notice is precisely the defining equation of eigenvalues and eigenvectors.


```
In [1]: import matplotlib.pyplot as plt
        import numpy as np

In [2]: Q = np.array([[ 1.,  2.], [ 4.,  3.]])

In [3]: eig_val, eig_vec = np.linalg.eig(Q)
        v1 = eig_vec[:,0]
        v2 = eig_vec[:,1]
        lambda1 = eig_val[0]
        lambda2 = eig_val[1]

In [4]: print eig_val
        print eig_vec

        [-1.  5.]
        [[-0.70710678 -0.4472136 ]
         [ 0.70710678 -0.89442719]]
            v1          v2
```


```
In [5]: # Does Q.v1 = lambda1(v1)? - YES
        print Q.dot(v1)
        print lambda1*v1

        [ 0.70710678 -0.70710678]
        [ 0.70710678 -0.70710678]

In [6]: # Does Q.v2 = lambda2(v2)? - YES
        print Q.dot(v2)
        print lambda2*v2

        [-2.23606798 -4.47213595]
        [-2.23606798 -4.47213595]

In [7]: # Does determinant |lambda1*Identity2x2 - Q | = 0?
        print np.linalg.det(lambda1*np.eye(2) - Q)

        0.0

In [8]: # Does determinant |lambda2*Identity2x2 - Q | = 0?
        print np.linalg.det(lambda1*np.eye(2) - Q)

        0.0
```

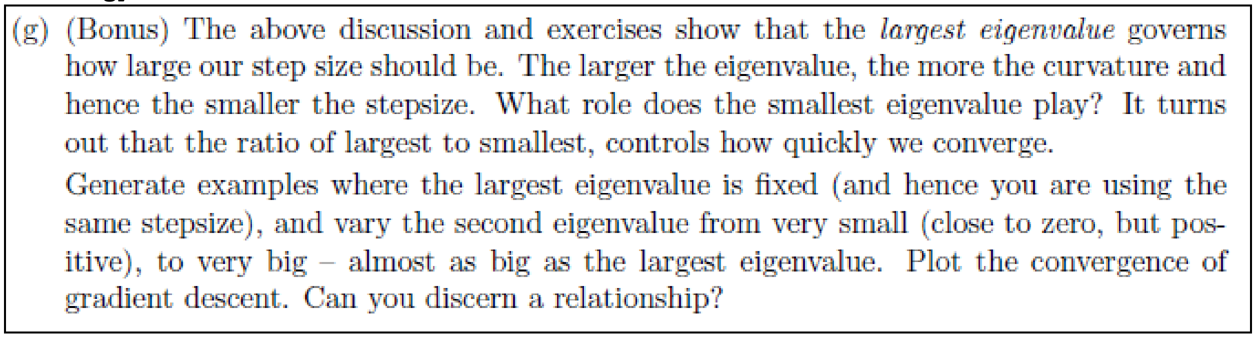
The following observations can be made:

$Q\overline{v} = \lambda\overline{v}$

$Q\overline{v} - \lambda\overline{v} = \overline{0}$

when λ is an eigenvalue, $\left|(\lambda\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} - Q)\right| = 0$

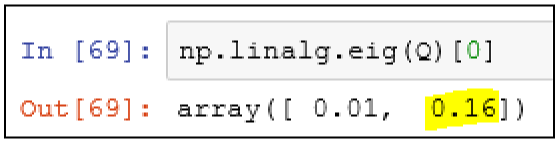
Q6 Part g]



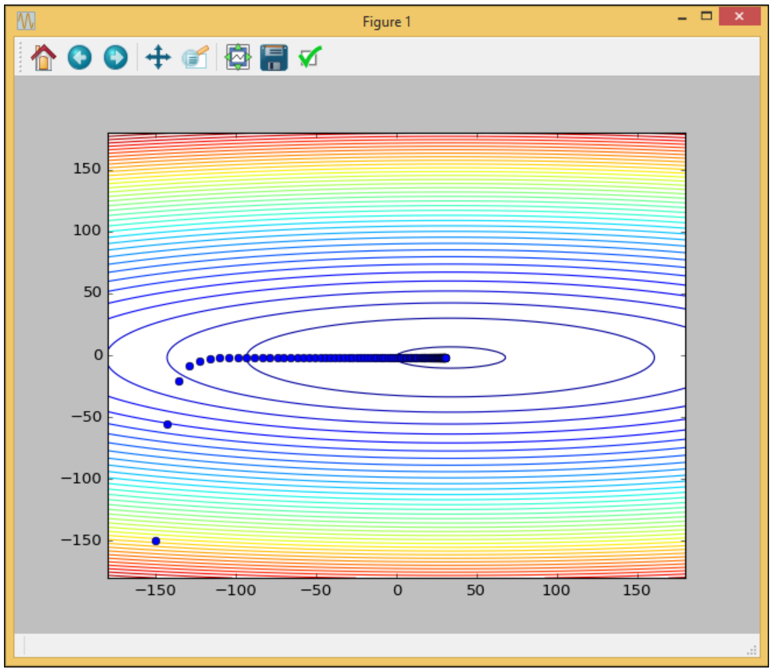
(g) (Bonus) The above discussion and exercises show that the *largest eigenvalue* governs how large our step size should be. The larger the eigenvalue, the more the curvature and hence the smaller the stepsize. What role does the smallest eigenvalue play? It turns out that the ratio of largest to smallest, controls how quickly we converge.

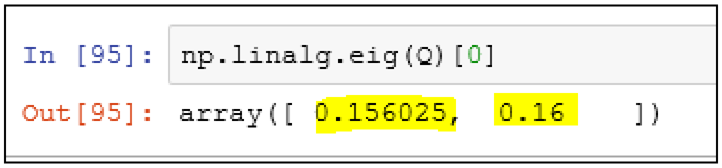
Generate examples where the largest eigenvalue is fixed (and hence you are using the same stepsize), and vary the second eigenvalue from very small (close to zero, but positive), to very big – almost as big as the largest eigenvalue. Plot the convergence of gradient descent. Can you discern a relationship?

Largest EigenValue fixed at **0.16**
Iterations: 500
Start from (-150,-150)
eta: **2** (Convergence)

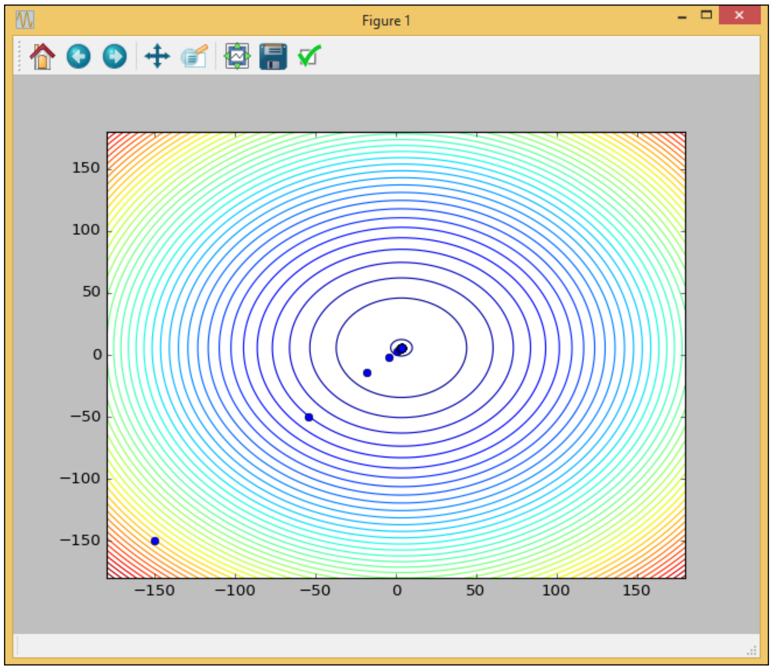


```
In [69]: np.linalg.eig(Q)[0]
Out[69]: array([ 0.01,   0.16])
```

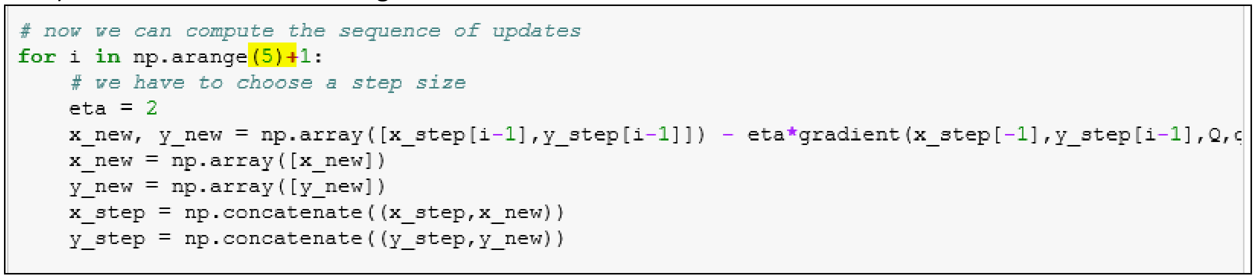




```
In [95]: np.linalg.eig(Q)[0]
Out[95]: array([ 0.156025,  0.16    ])
```



When the eigenvalues are very close to each other, the elliptical level sets are closer to circular which makes sense. But convergence happens a lot faster and in very few iterations (distance between blue dots). I was able to test convergence in 5 iterations.



```
# now we can compute the sequence of updates
for i in np.arange(5)+1:
    # we have to choose a step size
    eta = 2
    x_new, y_new = np.array([x_step[i-1],y_step[i-1]]) - eta*gradient(x_step[-1],y_step[i-1],Q,c
    x_new = np.array([x_new])
    y_new = np.array([y_new])
    x_step = np.concatenate((x_step,x_new))
    y_step = np.concatenate((y_step,y_new))
```