University of Texas at Austin, Cockrell School of Engineering
Data Mining – EE 380L

Problem Set # 2 (2a)

March 07, 2016

Gabrielson Eapen
EID: EAPENGP

Discussed Homework with Following Students:
1. Mudra Gandhi
2. Rayo Landeros

Q1]

Consider the quadratic function

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^\top Q\boldsymbol{x} + \boldsymbol{q}^\top \boldsymbol{x},$$

where

$$Q = \begin{pmatrix} 3 & 1 \\ 1 & 2 \end{pmatrix}, \qquad \boldsymbol{q} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}.$$

Start from the point $\boldsymbol{x}_0 = \begin{pmatrix} 10 \\ 10 \end{pmatrix}$, and solve the problem using gradient descent. Make sure to be clear about what step size you choose!

Then, solve again, but this time at each iteration, replace $\nabla f(\boldsymbol{x})$ by $\tilde{\boldsymbol{g}} = \nabla f(\boldsymbol{x}) + \boldsymbol{w}$, where $\boldsymbol{w} \sim N(0, I/10)$. That is, at each point in time, add independent Gaussian noise $N(0, 1/10)$ to each coordinate of $\nabla f(\boldsymbol{x})$.

What step size do you need to choose for convergence? Plot the convergence together with the convergence for standard gradient descent. Gradient descent may be faster, but amazingly, the noisy version also converges!

Code:

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

In [2]:
```
def f(x, Q, q):
    return float(0.5* x.T * Q * x + q.T * x)
```

In [3]:
```
def f_grad(x, Q, q):
    return ( Q * x + q)
```

In [4]:
```
def f_grad_w_noise(x, Q, q):
    return ( Q * x + q) + np.matrix([[np.random.uniform(0, 0.1)], [np.random.uniform(0, 0.1)]])
```

In [5]:
```
#Use provided values
Q = np.matrix([[3.0, 1.0], [1.0, 2.0]])
q = np.matrix([[2.0], [1.0]])
print np.linalg.eig(Q)[0]
# Print optimal step size with no noise
print 1 / np.amax(np.linalg.eig(Q)[0])
```

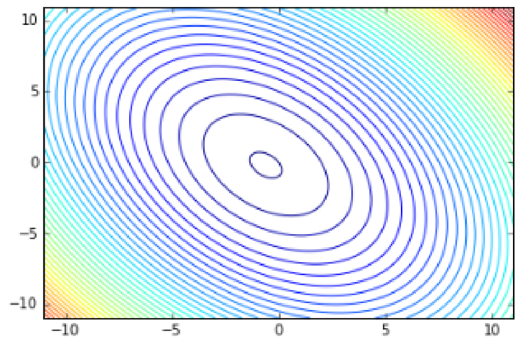
```
[ 3.61803399  1.38196601]
0.27639320225
```

In [6]:
```
x_step = np.matrix([[10.0],[10.0]])
iterations = 1000
# Choose 0.275 (based on largest Eigen value)
eta = .275

# Gradient Descent
for i in np.arange(iterations)+1:
    x_next = x_step[:,-1] - eta * f_grad(x_step[:,-1], Q, q)
    x_step = np.c_[x_step,x_next]
```

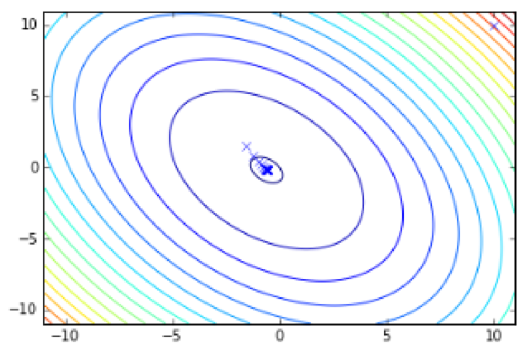
In [7]:
```
# Compute Levels
plt.figure()
size = 200
x1_ = np.linspace(-11, 11, num=size)
x2_ = np.linspace(-11, 11, num=size)
x1, x2 = np.meshgrid(x1_, x2_)

levels = np.zeros((len(x1_), len(x2_)))
for i in range(len(x1)):
    for j in range(len(x2)):
        x = np.matrix([[x1[i,j]], [x2[i,j]]])
        levels[i, j] = f(x, Q, q)
#print levels
plt.contour(x1, x2, levels, 50)
plt.show()
```



Plot with no noise version

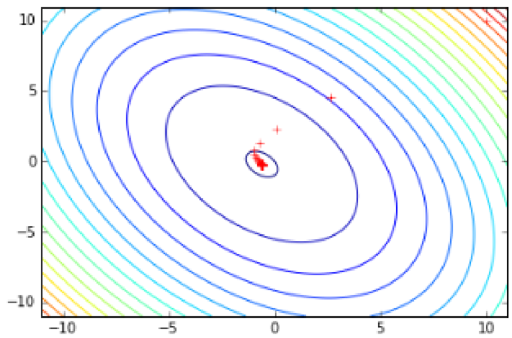
In [8]:
```
# Plot Gradient Descent
plt.plot(x_step[0],x_step[1],'bx')
plt.ylim(-11,11)
plt.xlim(-11,11)
plt.contour(x1, x2, levels, 20)
plt.show()
```



In [9]:
```
# Noise version
x_step2 = np.matrix([[10.0],[10.0]])
iterations = 1000
# Choose 0.276 - Error tolerance 0.1 ~= 0.175
eta = .175
# Gradient Descent with Noise
for i in np.arange(iterations)+1:
    x_next2 = x_step2[:,-1] - eta * f_grad_w_noise(x_step2[:,-1], Q, q)
    x_step2 = np.c_[x_step2,x_next2]
```

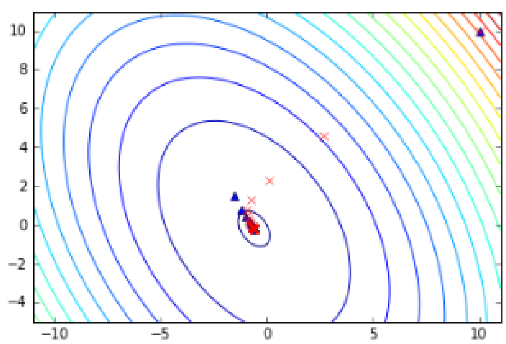
Plot with noise version. Step size (=0.175) had to be lowered to account for error tolerance.

In [10]:
```
#Plot Gradient Descent with Noise
plt.plot(x_step2[0],x_step2[1],'r+')
plt.ylim(-11,11)
plt.xlim(-11,11)
plt.contour(x1, x2, levels, 20)
plt.show()
```

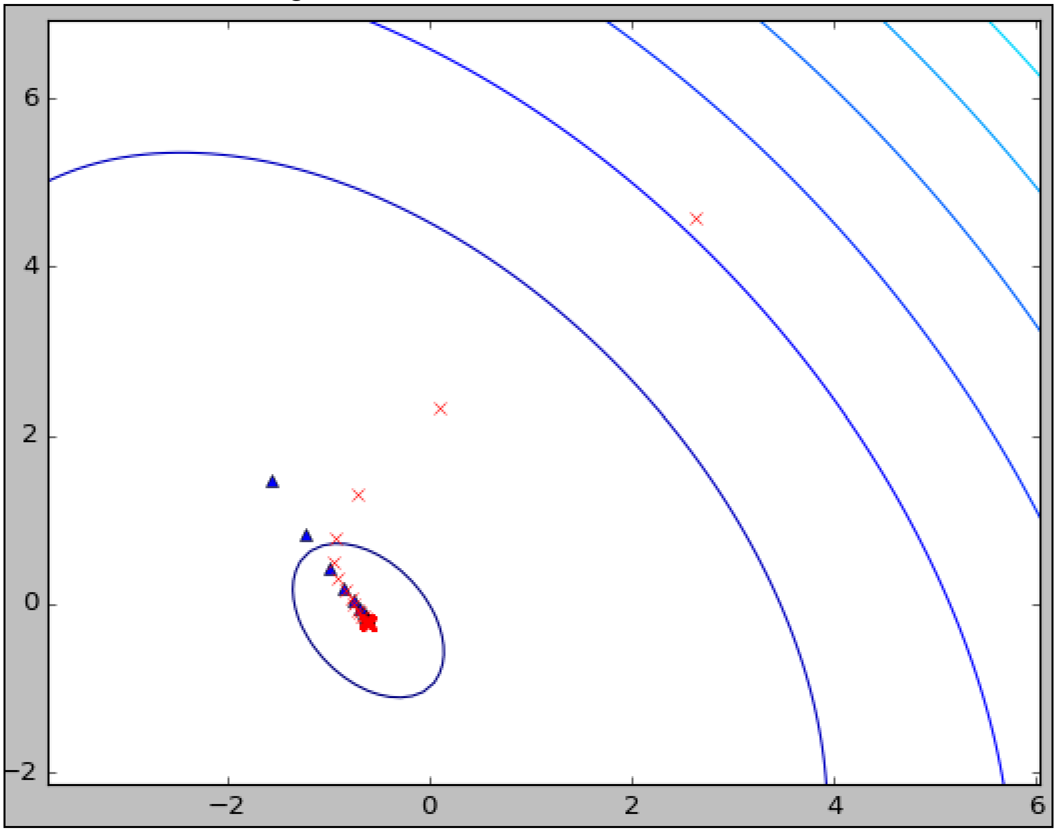


Plotting both versions together. Non-Noise version converges faster.

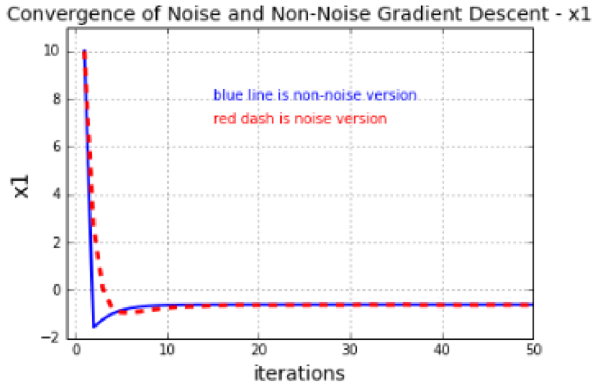
In [11]:
```
# Plot Noise and non-Version together
plt.plot(x_step[0],x_step[1],'b^',linewidth=2.0)
plt.plot(x_step2[0],x_step2[1],'rx',linewidth=1.0)
plt.ylim(-5,11)
plt.xlim(-11,11)
plt.contour(x1, x2, levels, 20)
plt.show()
```



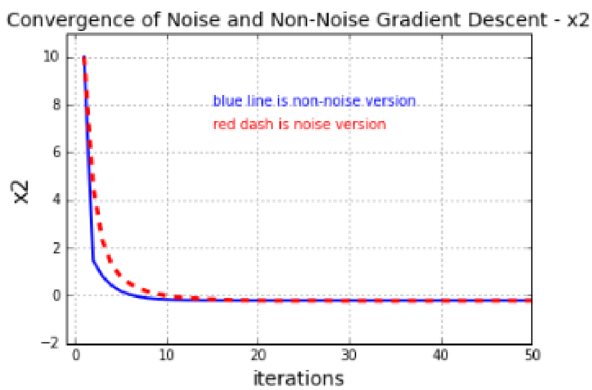
Zoomed IN. Blue (Gradient Descent) converges faster than Red (with Noise). Step size is **0.175** (0.275 – 0.1) to account for max error tolerance of 0.1 from the optimal step size of **0.276** otherwise the noise version does not converge



In [12]:
```
#Compare convergence for x1
plt.plot(np.arange(1001)+1,np.asarray(x_step)[0],'b',linewidth=2.0)
plt.plot(np.arange(1001)+1,np.asarray(x_step2)[0],'r--',linewidth=3.0)
plt.ylim(-2, 11)
plt.xlim(-1, 50)
plt.ylabel('x1',fontsize=16)
plt.xlabel('iterations',fontsize=14)
plt.title('Convergence of Noise and Non-Noise Gradient Descent - x1',fontsize=14)
plt.text(15, 7, 'red dash is noise version', color='red')
plt.text(15, 8, 'blue line is non-noise version', color='blue')
plt.grid(True)
plt.show()
```



In [13]:
```
#Compare convergence for x2
plt.plot(np.arange(1001)+1,np.asarray(x_step)[1],'b',linewidth=2.0)
plt.plot(np.arange(1001)+1,np.asarray(x_step2)[1],'r--',linewidth=3.0)
plt.ylim(-2, 11)
plt.xlim(-1, 50)
plt.ylabel('x2',fontsize=16)
plt.xlabel('iterations',fontsize=14)
plt.title('Convergence of Noise and Non-Noise Gradient Descent - x2',fontsize=14)
plt.text(15, 7, 'red dash is noise version', color='red')
plt.text(15, 8, 'blue line is non-noise version', color='blue')
plt.grid(True)
plt.show()
```



Q1 Part b]

Now let's design our stochastic gradient. Suppose that $\tilde{g}$ is a random variable. With probability 1/4 it is equal to $\nabla(\boldsymbol{x}_1 \cdot \beta - y_1)^2 = 2\boldsymbol{x}_1(\boldsymbol{x}_1 \cdot \beta - y_1)$. With probability 1/4 it is equal to $\nabla(\boldsymbol{x}_2 \cdot \beta - y_2)^2 = 2\boldsymbol{x}_1(\boldsymbol{x}_2 \cdot \beta - y_2)$. With probability 1/4 it is equal to $\nabla(\boldsymbol{x}_3 \cdot \beta - y_3)^2 = 2\boldsymbol{x}_3(\boldsymbol{x}_3 \cdot \beta - y_3)$, and with probability 1/4 it is equal to $\nabla(\boldsymbol{x}_4 \cdot \beta - y_4)^2 = 2\boldsymbol{x}_4(\boldsymbol{x}_4 \cdot \beta - y_4)$.

b. Show that the expected value of $\tilde{g}$ is equal to the full gradient.

Data = $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)\}$

$$\nabla f(\beta) = \frac{1}{4}\left(\nabla(\boldsymbol{x}_1 \cdot \beta - y_1)^2 + \nabla(\boldsymbol{x}_2 \cdot \beta - y_2)^2 + \nabla(\boldsymbol{x}_3 \cdot \beta - y_3)^2 + \nabla(\boldsymbol{x}_4 \cdot \beta - y_4)^2\right)$$

$$\nabla \mathrm{f}(\beta) = \frac{1}{4}\big(2x_1(x_1 \cdot \beta - y_1) + 2x_2(x_2 \cdot \beta - y_2) + 2x_3(x_3 \cdot \beta - y_3) + 2x_4(x_4 \cdot \beta - y_4)\big)$$

$$\mathrm{E}[\mathbf{g}] = \tfrac{1}{4}p_1 + \tfrac{1}{4}p_2 + \tfrac{1}{4}p_3 + \tfrac{1}{4}p_4$$

$$\mathrm{E}[\mathbf{g}] = \frac{1}{4}(p_1 + p_2 + p_3 + p_4)$$

where

$p_1 = 2x_1(x_1 \cdot \beta - y_1)$, $p_2 = 2x_2(x_2 \cdot \beta - y_2)$, $p_3 = 2x_3(x_3 \cdot \beta - y_3)$, $p_4 = 2x_4(x_4 \cdot \beta - y_4)$

$$\mathrm{E}[\mathbf{g}] = \frac{1}{4}\big(2x_1(x_1 \cdot \beta - y_1) + 2x_2(x_2 \cdot \beta - y_2) + 2x_3(x_3 \cdot \beta - y_3) + 2x_4(x_4 \cdot \beta - y_4)\big) = \nabla \mathrm{f}(\beta)$$

Q1 Part c]

c. Solve the problem you created using gradient descent. How did you choose your step size?

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

In [2]:
```
n=10000
p=100
# X is a nXp matrix
X=np.random.rand(n,p)
print X.shape
print np.amin(X)
print np.amax(X)
# True Beta is 1
BStar = np.ones((p,1), dtype=np.float)
BInit = np.full((p, 1), 0.7)
print BStar.shape
# Noise scaled to be between 0 and 0.1
w = np.random.rand(n,1) * 0.1
# Compute y with noise
y = np.dot(X,BStar) + w
print y.shape
```

```
(10000L, 100L)
7.12431794114e-07
0.999999464934
(100L, 1L)
(10000L, 1L)
```

```
In [4]: # Answer to part C - Gradient Descent
        beta_step = np.full((p, 1), 0.7)

        iterations = 100
        eta = 0.022
        time_step = np.array([0.])
        start_time = time.time()
        for i in np.arange(iterations)+1:
            beta_next = beta_step[:,-1] - eta *B_grad(X,beta_step[:,-1], y, n)
            beta_step = np.c_[beta_step,beta_next]
            time_step = np.concatenate((time_step,np.array([time.time() - start_time])))

        elapsed_time = time.time() - start_time
        print "Elapsed Time ",elapsed_time

        Elapsed Time  10.6190001965
```

I used an eta size of 0.022 and there is a little overshoot before convergence. We do not get an exact value for Beta of 1 due to the noise added. The algorithm converges faster to accurate value but the computation takes longer as all values are examined.

Output of beta values:

```
In [6]: beta_step[5,:]

Out[6]: array([ 0.7       ,  1.03374503,  0.99899654,  1.00260485,  1.0022206 ,
                1.00225198,  1.00224012,  1.0022328 ,  1.00222503,  1.00221735,
                1.00220968,  1.00220205,  1.00219444,  1.00218686,  1.0021793 ,
                1.00217178,  1.00216428,  1.00215681,  1.00214936,  1.00214195,
                1.00213456,  1.00212719,  1.00211986,  1.00211255,  1.00210527,
                1.00209801,  1.00209078,  1.00208358,  1.0020764 ,  1.00206925,
                1.00206213,  1.00205503,  1.00204796,  1.00204091,  1.00203389,
                1.00202689,  1.00201992,  1.00201298,  1.00200606,  1.00199916,
                1.00199229,  1.00198545,  1.00197863,  1.00197184,  1.00196507,
                1.00195832,  1.0019516 ,  1.00194491,  1.00193824,  1.00193159,
                1.00192497,  1.00191837,  1.00191179,  1.00190524,  1.00189872,
                1.00189221,  1.00188573,  1.00187928,  1.00187285,  1.00186644,
                1.00186005,  1.00185369,  1.00184735,  1.00184104,  1.00183474,
                1.00182847,  1.00182223,  1.001816  ,  1.0018098 ,  1.00180362,
                1.00179747,  1.00179133,  1.00178522,  1.00177913,  1.00177307,
                1.00176702,  1.001761  ,  1.001755  ,  1.00174902,  1.00174306,
                1.00173713,  1.00173121,  1.00172532,  1.00171945,  1.0017136 ,
                1.00170777,  1.00170196,  1.00169618,  1.00169041,  1.00168467,
                1.00167895,  1.00167325,  1.00166756,  1.0016619 ,  1.00165626,
                1.00165065,  1.00164505,  1.00163947,  1.00163391,  1.00162837,
                1.00162286])
```

Computed Step Size:

```
In [7]: # Compute StepSize
        1/np.amax(np.linalg.eig(np.dot(X.T,X))[0])
        #X.T.shape

Out[7]: 3.9851796348745497e-06
```

Q1 Part d]

d. Solve the problem using your implementation of the SGD algorithm. Again, take care in the choice of your step size.

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

In [2]:
```
n=10000
p=100
# X is a nXp matrix
X=np.random.rand(n,p)
print X.shape
print np.amin(X)
print np.amax(X)
# True Beta is 1
BStar = np.ones((p,1), dtype=np.float)
BInit = np.full((p, 1), 0.7)
print BStar.shape
# Noise scaled to be between 0 and 0.1
w = np.random.rand(n,1) * 0.1
# Compute y with noise
y = np.dot(X,BStar) + w
print y.shape
```

```
(10000L, 100L)
7.90058796163e-09
0.999999011402
(100L, 1L)
(10000L, 1L)
```

In [3]:
```
def B_grad_SGD(x, B, y, n):
    #pick i randomly for each invocation
    i = np.random.randint(n-1)
    sum = 2 * x[i,:] * (np.dot(x[i,:],B) - y[i])
    return sum
```

In [7]:
```
# Answer to part D - SGD Gradient Descent
beta_step_SGD = np.full((p, 1), 0.7)
#print beta_step_SGD
iterations = 100
eta = 0.022
time_step_SGD = np.array([0.])
start_time_SGD = time.time()

for i in np.arange(iterations)+1:
    beta_next_SGD = beta_step_SGD[:,-1] - eta *B_grad_SGD(X,beta_step_SGD[:,-1], y, n)
    beta_step_SGD = np.c_[beta_step_SGD,beta_next_SGD]
    time_step_SGD = np.concatenate((time_step_SGD,np.array([time.time() - start_time_SGD])))

elapsed_time_SGD = time.time() - start_time_SGD
print "Elapsed Time (SGD): ", elapsed_time_SGD
```

```
Elapsed Time (SGD):   0.0110001564026
```

I used an eta size of 0.022 and convergence is slower. We do not get an exact value for Beta of 1 due to the noise added. The algorithm converges slower to accurate value but the computation is much faster as not all values are examined. See difference in computed Elapsed Time.

```
In [8]: beta_step_SGD[5,:]

Out[8]: array([ 0.7       ,  1.0029192 ,  0.97430409,  0.97899571,  0.97109585,
                0.97427317,  0.97094655,  0.96451705,  0.95425736,  0.96221156,
                0.96403319,  0.96163299,  0.9253333 ,  0.91792793,  0.96648925,
                0.96081041,  0.99900153,  0.99830435,  0.9995133 ,  0.99524963,
                1.009679  ,  0.99768527,  1.01377386,  1.0109877 ,  1.00814729,
                1.00709177,  1.01151314,  1.02515857,  1.05099749,  1.0409894 ,
                1.02830904,  1.02750973,  1.06267006,  1.07953498,  1.06702313,
                1.06163641,  1.06552663,  1.05967071,  1.06443701,  1.06270825,
                1.06631736,  1.06392806,  1.0652994 ,  1.08179394,  1.0793106 ,
                1.08093341,  1.03072883,  1.01705445,  1.02149022,  1.02459623,
                1.02214395,  1.03358401,  1.02716231,  1.0276148 ,  1.04155422,
                1.01159977,  1.01741615,  1.00302342,  1.0011191 ,  0.99484571,
                0.99397814,  1.01812505,  1.00818234,  1.00892382,  1.02126279,
                0.99394478,  0.98377627,  1.01074449,  1.01603346,  1.00492586,
                1.01861032,  1.00566031,  1.02860407,  1.0237245 ,  1.02453346,
                0.99893885,  1.01823811,  1.03210487,  1.02979117,  1.02860006,
                1.04301924,  1.04250382,  1.03345811,  1.01989379,  1.03715015,
                1.0338363 ,  1.02875016,  1.05158076,  1.03711445,  1.05315685,
                0.98720647,  0.99196303,  0.99805012,  0.99234232,  1.01450325,
                1.00956278,  1.01719329,  1.02203354,  1.00308126,  1.02222065,
                1.02424019])
```

Computed Step Size:

```
In [9]: # Compute StepSize
        1/np.amax(np.linalg.eig(np.dot(X.T,X))[0])
        #X.T.shape

Out[9]: 3.9871010624356013e-06
```