Question 4.1

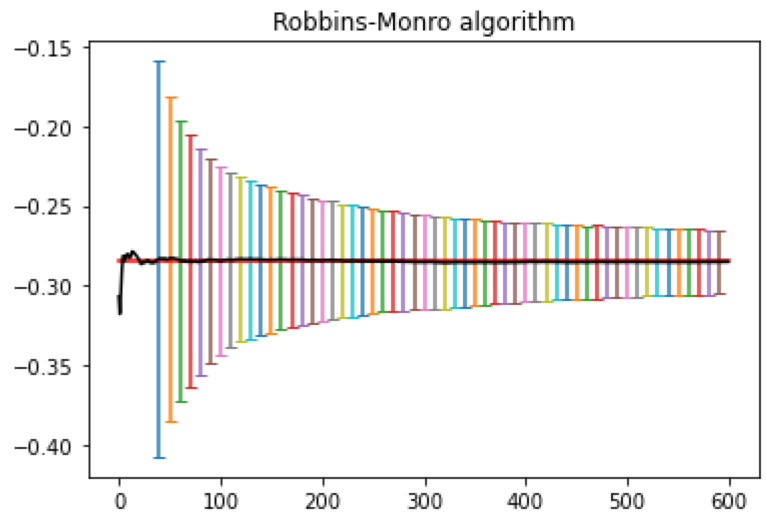
In [1]:

```python
import numpy as np
from matplotlib import pyplot as plt
```

.1 & .2

In [2]:

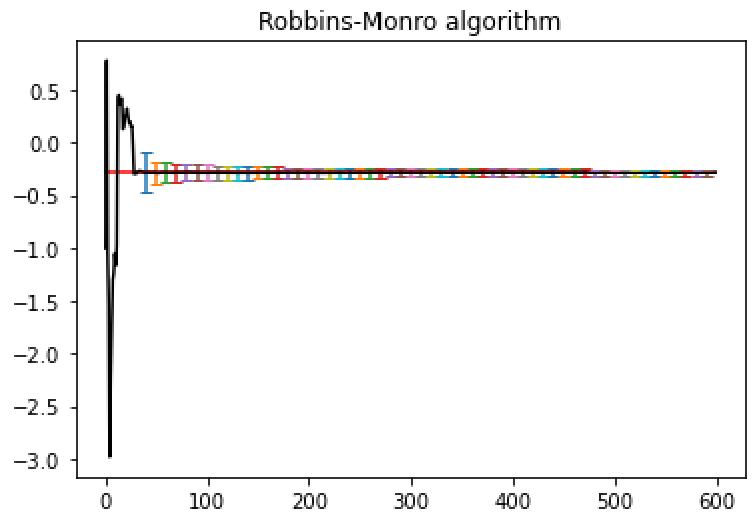
```python
N = 600   # Number of Data
L = 2   # Dimension of the unknown vector
theta = np.random.randn(L, 1)   # Unknown parameter
w = np.zeros((L, 1))   # Initialization


Iter_n = 1000 #number of iterations
noisev = 0.1
wtot = np.zeros((N, Iter_n))
inputvec = lambda n: X[:, n].copy()


for t in range(0, Iter_n):
    X = np.random.randn(L, N)
    noise = np.random.randn(N, 1) * np.sqrt(noisev)
    y = np.zeros((N, 1))
    y[0:N] = np.dot(X[:, 0:N].conj().T, theta)
    y = y + noise
    w = np.zeros((L, 1))
    for i in range(0, N):
        mu = 1 / (i+1)   # Step size
        e = y[i] - np.dot(w.conj().T, inputvec(i))   # Error computation
        w = w + mu * e * inputvec(i)
        wtot[i][t] = w[0][0]

theta1 = theta[0] * np.ones((N, 1))
plt.title("Robbins-Monro algorithm")
plt. plot(theta1, color='red')

mean_w = np.mean(wtot.conj().T, axis=0)
plt.plot(mean_w, color='k', linestyle='solid')

for i in range(0, N):
    if i % 10 == 0 and i > 30:
        plt.errorbar(i, mean_w[i], yerr=np.std(wtot[i, :], axis=0), capsize=3)

plt.show()
```



.3

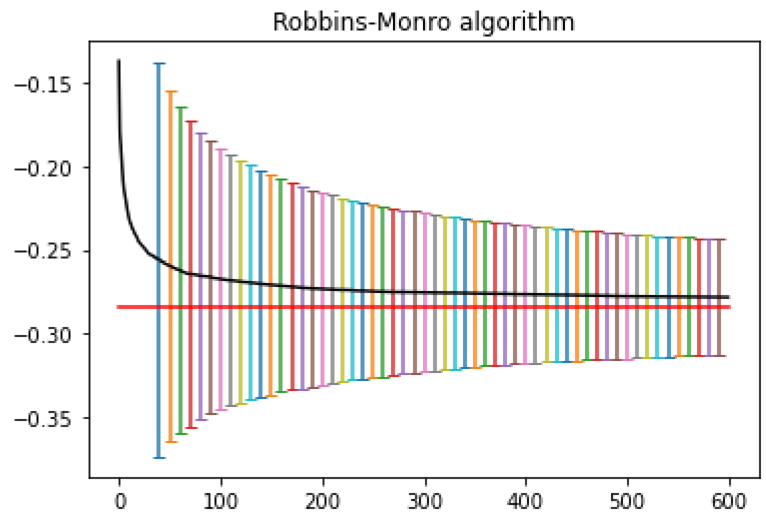
In [3]:

```python
for t in range(0, Iter_n):
    X = np.random.randn(L, N)
    noise = np.random.randn(N, 1) * np.sqrt(noisev)
    y = np.zeros((N, 1))
    y[0:N] = np.dot(X[:, 0:N].conj().T, theta)
    y = y + noise
    w = np.zeros((L, 1))
    for i in range(0, N):
        mu = 1 / (i+1)*4   # Step size
        e = y[i] - np.dot(w.conj().T, inputvec(i))   # Error computation
        w = w + mu * e * inputvec(i)
        wtot[i][t] = w[0][0]

theta1 = theta[0] * np.ones((N, 1))
plt.title("Robbins-Monro algorithm")
plt. plot(theta1, color='red')
meanw = np.mean(wtot.conj().T, axis=0)

plt.plot(meanw, color='k', linestyle='solid')


for i in range(0, N):
    if i % 10 == 0 and i > 30:
        plt.errorbar(i, meanw[i], yerr=np.std(wtot[i, :], axis=0), capsize=3)

plt.show()
```



In [4]:

```python
for t in range(0, Iter_n ):
    X = np.random.randn(L, N)
    noise = np.random.randn(N, 1) * np.sqrt(noisev)
    y = np.zeros((N, 1))
    y[0:N] = np.dot(X[:, 0:N].conj().T, theta)
    y = y + noise
    w = np.zeros((L, 1))
    for i in range(0, N):
        mu = 1 / ((i+1)*2) # Step size
        e = y[i] - np.dot(w.conj().T, inputvec(i))   # Error computation
        w = w + mu * e * inputvec(i)
        wtot[i][t] = w[0][0]

theta1 = theta[0] * np.ones((N, 1))
plt.title("Robbins-Monro algorithm")
plt. plot(theta1, color='red')
meanw = np.mean(wtot.conj().T, axis=0)
plt.plot(meanw, color='k', linestyle='solid')


for i in range(0, N):
    if i % 10 == 0 and i > 30:
        plt.errorbar(i, meanw[i], yerr=np.std(wtot[i, :], axis=0), capsize=3)

plt.show()
```

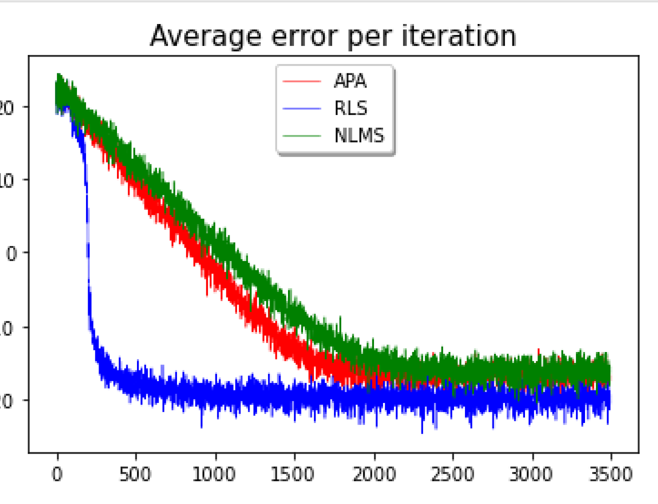


In this case it can be seen how the line converges with the unknown parameter quickly. Also, it can be concluded that the algorithm is quite susceptible to the step size sequence,the error is affected when the step size.

Question 4.2

In [1]:
```python
import os
import sys
import numpy as np
import warnings
warnings.filterwarnings("ignore", category=RuntimeWarning)
from matplotlib import pyplot as plt
sys.path.append(os.getcwd())
sys.path.append('../')
```

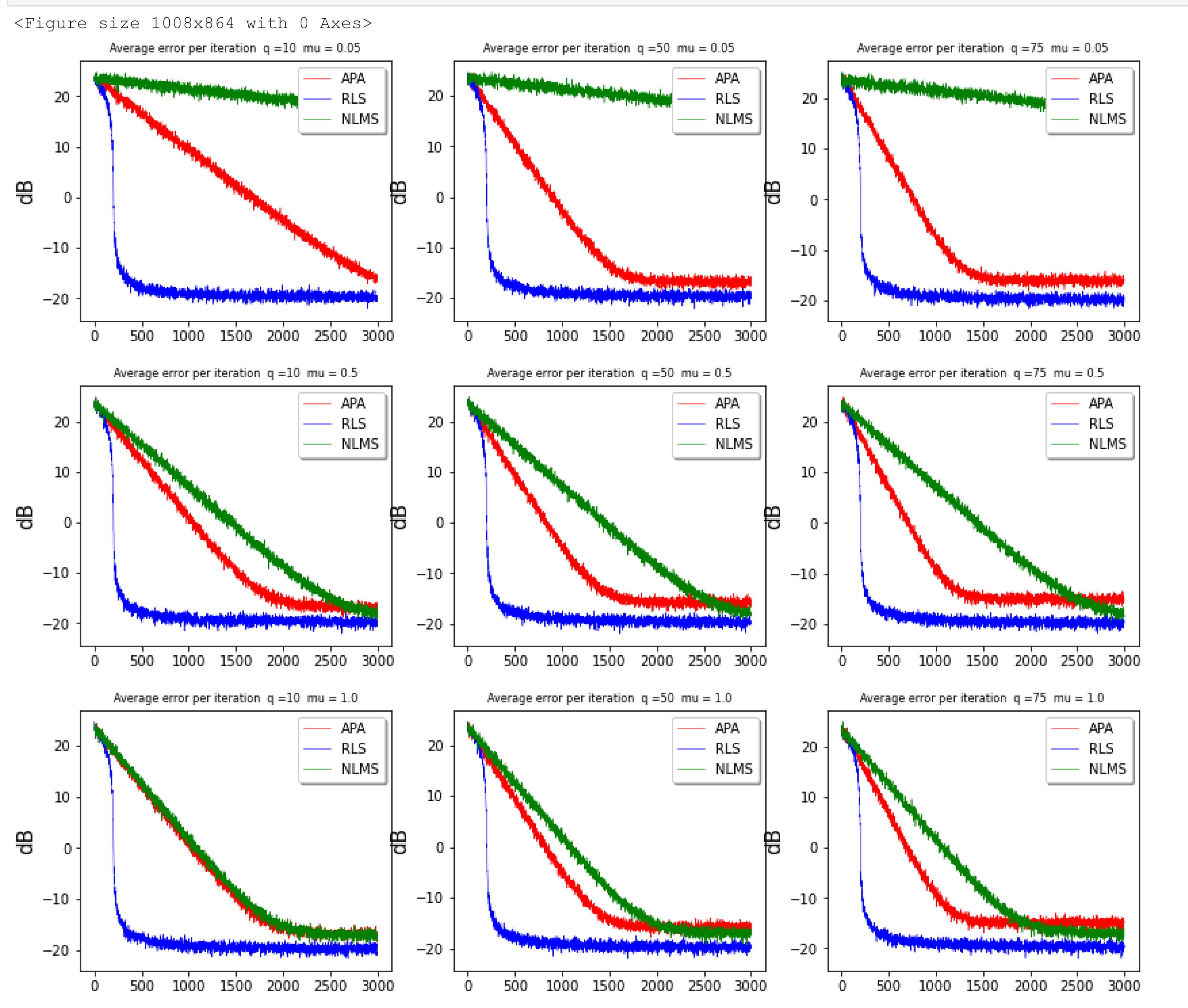
In [2]:
```python
L = 200  # Dimension of the unknown vector
N = 3500  # Number of Data
theta = np.random.randn(L, 1)  # Unknown parameter

Iter_n = 30
MSE1 = np.zeros((N, Iter_n))
MSE2 = np.zeros((N, Iter_n))
MSE3 = np.zeros((N, Iter_n))

noisevar = 0.01
epsilon = np.sqrt(2) * noisevar


for t in range(0, Iter_n):

    X = np.random.randn(L, N)
    inputvec = lambda n: np.array([X[:, n].copy()]).conj().T


    noise = np.random.randn(N, 1) * np.sqrt(noisevar)

    y = np.zeros((N, 1))
    y[0:N] = np.dot(X[:, 0:N].conj().T, theta)
    y = y + noise
    w = np.zeros((L, 1))
    mu = 0.2
    delta = 0.001
    q = 30
    for i in range(0, N):
        if i > q:
            qq = range(i, i - q, -1)

            yvec = y[qq]
            Xq = inputvec(qq)
            Xq = np.reshape(Xq, newshape=(Xq.shape[0], Xq.shape[1]))
            e = yvec - np.dot(Xq, w)
            eins = y[i] - np.dot(w.conj().T, inputvec(i))
            w = w + mu * np.dot(np.dot(Xq.conj().T, np.linalg.inv(delta*np.eye(q)+np.dot(Xq, Xq.conj().T))), e)
            MSE1[i, t] = eins ** 2

    # RLS recursion
    w = np.zeros((L,1))
    delta = 0.001
    P = (1/delta) * np.eye(L)
    for i in range(0, N):

        gamma = 1/(1+np.dot(inputvec(i).conj().T, np.dot(P, inputvec(i))))
        gi = np.dot(P, inputvec(i)) * gamma
        e = y[i] - np.dot(w.conj().T, inputvec(i))
        w = w + gi * e
        P = P - np.dot(gi, gi.conj().T)/gamma
        MSE2[i, t] = e ** 2

    # NLMS Recursion
    w = np.zeros((L, 1))
    delta = 0.001
    mu = 1.2

    for i in range(0, N):

        e = y[i] - np.dot(w.conj().T, inputvec(i))
        mun = mu / (delta+np.dot(inputvec(i).conj().T, inputvec(i)))
        w = w + mun * e * inputvec(i)
        MSE3[i, t] = e ** 2

MSEav1 = sum(MSE1.conj().T) / Iter_n
MSEav2 = sum(MSE2.conj().T) / Iter_n
MSEav3 = sum(MSE3.conj().T) / Iter_n

plt.plot(10 * np.log10(MSEav1), 'r', lw=0.5)
plt.plot(10 * np.log10(MSEav2), 'b', lw=0.5)
plt.plot(10 * np.log10(MSEav3), 'g', lw=0.5)
plt.title("Average error per iteration", fontsize=15)
plt.ylabel('dB', fontsize=15)
plt.legend(('APA', 'RLS', 'NLMS'),
           loc='upper center', shadow=True)
plt.show()
```



From the graph above it is plausible to say that the fastest perfomnce comes from RLS

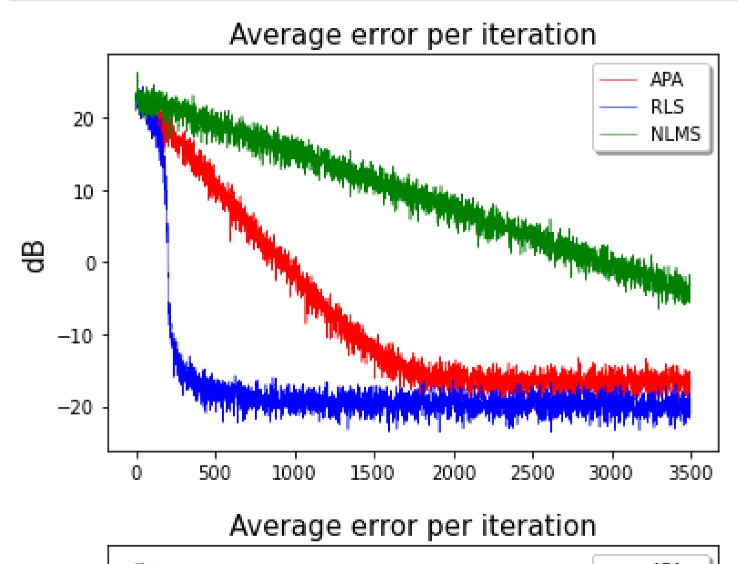
In [3]:
```python
#changing mu & q

L = 200  # Dimension of the unknown vector
N = 3000  # Number of Data
theta = np.random.randn(L, 1)  # Unknown parameter

Iter_n = 100
MSE1 = np.zeros((N, Iter_n))
MSE2 = np.zeros((N, Iter_n))
MSE3 = np.zeros((N, Iter_n))

noisevar = 0.01
epsilon = np.sqrt(2) * noisevar

mu= np.array([0.05,0.5,1])
q = np.array([10,50,75])
#mu[t] = 0.2
delta = 0.001

g=1
fig = plt.figure(figsize=(14,12))
for k in range(len(mu)):
    fig = plt.figure(figsize=(14,12))
    for j in range(len(q)):

        for t in range(0, Iter_n):

            X = np.random.randn(L, N)
            inputvec = lambda n: np.array([X[:, n].copy()]).conj().T


            noise = np.random.randn(N, 1) * np.sqrt(noisevar)

            y = np.zeros((N, 1))
            y[0:N] = np.dot(X[:, 0:N].conj().T, theta)
            y = y + noise
            w = np.zeros((L, 1))

            for i in range(0, N):
                if i > q[j]:
                    qq = range(i, i - q[j], -1)

                    yvec = y[qq]
                    Xq = inputvec(qq)
                    Xq = np.reshape(Xq, newshape=(Xq.shape[0], Xq.shape[1]))
                    e = yvec - np.dot(Xq, w)
                    eins = y[i] - np.dot(w.conj().T, inputvec(i))
                    w = w + mu[k] * np.dot(np.dot(Xq.conj().T, np.linalg.inv(delta*np.eye(q[j])+np.dot(Xq, Xq.c
                    MSE1[i, t] = eins ** 2

            # RLS recursion
            w = np.zeros((L,1))
            P = (1/delta) * np.eye(L)
            for i in range(0, N):

                gamma = 1/(1+np.dot(inputvec(i).conj().T, np.dot(P, inputvec(i))))
                gi = np.dot(P, inputvec(i)) * gamma
                e = y[i] - np.dot(w.conj().T, inputvec(i))
                w = w + gi * e
                P = P - np.dot(gi, gi.conj().T)/gamma
                MSE2[i, t] = e ** 2

            # RLS recursion
            w = np.zeros((L, 1))

            for i in range(0, N):
                # mu[t]=1;%/(i^0.5);
                e = y[i] - np.dot(w.conj().T, inputvec(i))
                muu = mu[k] / (delta+np.dot(inputvec(i).conj().T, inputvec(i)))
                w = w + muu * e * inputvec(i)
                MSE3[i, t] = e ** 2

        MSEav1 = sum(MSE1.conj().T) / Iter_n
        MSEav2 = sum(MSE2.conj().T) / Iter_n
        MSEav3 = sum(MSE3.conj().T) / Iter_n

        plt.subplot(len(q),len(mu),g)
        g +=1
        plt.plot(10 * np.log10(MSEav1), 'r', lw=0.5)
        plt.plot(10 * np.log10(MSEav2), 'b', lw=0.5)
        plt.plot(10 * np.log10(MSEav3), 'g', lw=0.5)
        plt.title('Average error per iteration  q =' + str(q[j]) + '  mu = ' + str(mu[k]), fontsize=8)
        plt.ylabel('dB', fontsize=15)
        plt.legend(('APA', 'RLS', 'NLMS'), loc='upper right', shadow=True)
    plt.show()
```

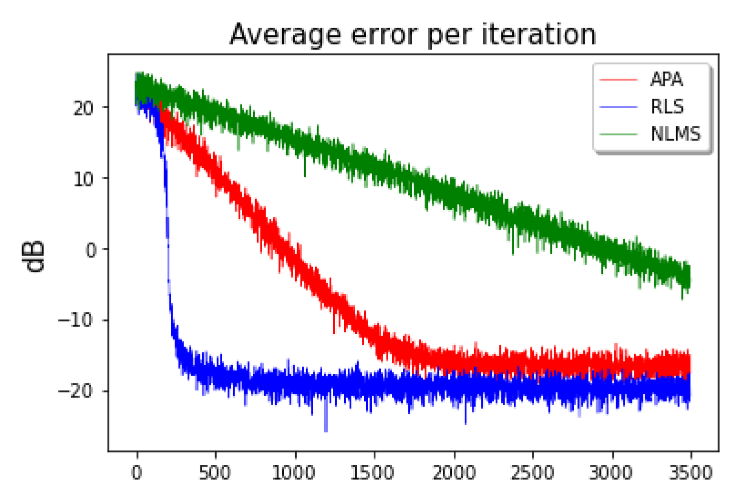
```
<Figure size 1008x864 with 0 Axes>
```

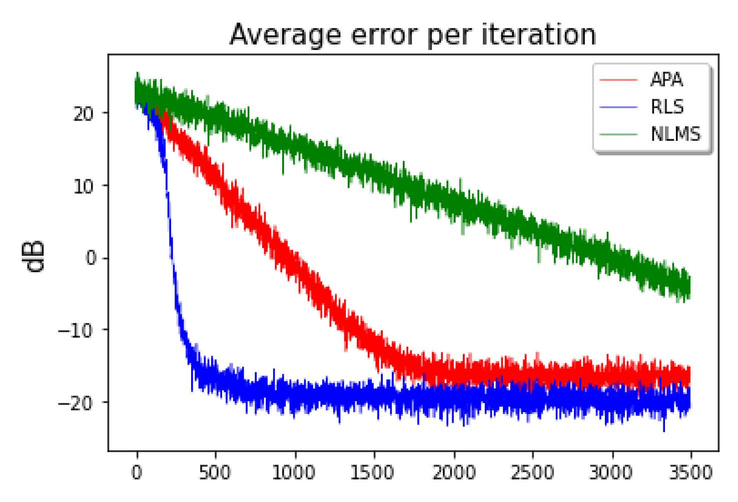


In the graphs above the parameter q is changed 3 times (10,50,75),change q increase window and that increases the data used to predict,in this case the only recursion method changing is APA. Also, the value of mu changes with 3 different values (0.005, 0.5, 1), this shows how m is related to learning rate, the bigger it is the fastets it update. It can be seen that the most effective recursion method is RLS and the slowest one is NLMS.

In [4]:
```python
#changing delta

L = 200  # Dimension of the unknown vector
N = 3500  # Number of Data
theta = np.random.randn(L, 1)  # Unknown parameter

Iter_n = 30
MSE1 = np.zeros((N, Iter_n))
MSE2 = np.zeros((N, Iter_n))
MSE3 = np.zeros((N, Iter_n))

noisevar = 0.01
epsilon = np.sqrt(2) * noisevar

delta = np.array([0.001,0.1,1])
mu = 0.2
for j in range(len(delta)):
    for t in range(0, Iter_n):

        X = np.random.randn(L, N)
        inputvec = lambda n: np.array([X[:, n].copy()]).conj().T


        noise = np.random.randn(N, 1) * np.sqrt(noisevar)

        y = np.zeros((N, 1))
        y[0:N] = np.dot(X[:, 0:N].conj().T, theta)
        y = y + noise
        w = np.zeros((L, 1))
        q = 30
        for i in range(0, N):
            if i > q:
                qq = range(i, i - q, -1)

                yvec = y[qq]
                Xq = inputvec(qq)
                Xq = np.reshape(Xq, newshape=(Xq.shape[0], Xq.shape[1]))
                e = yvec - np.dot(Xq, w)
                eins = y[i] - np.dot(w.conj().T, inputvec(i))
                w = w + mu * np.dot(np.dot(Xq.conj().T, np.linalg.inv(delta[j]*np.eye(q)+np.dot(Xq, Xq.conj().T
                MSE1[i, t] = eins ** 2

        # RLS recursion
        w = np.zeros((L,1))
        P = (1/delta[j]) * np.eye(L)
        for i in range(0, N):

            gamma = 1/(1+np.dot(inputvec(i).conj().T, np.dot(P, inputvec(i))))
            gi = np.dot(P, inputvec(i)) * gamma
            e = y[i] - np.dot(w.conj().T, inputvec(i))
            w = w + gi * e
            P = P - np.dot(gi, gi.conj().T)/gamma
            MSE2[i, t] = e ** 2

        # NLMS Recursion
        w = np.zeros((L, 1))

        for i in range(0, N):

            e = y[i] - np.dot(w.conj().T, inputvec(i))
            muu = mu / (delta[j]+np.dot(inputvec(i).conj().T, inputvec(i)))
            w = w + muu * e * inputvec(i)
            MSE3[i, t] = e ** 2

    MSEav1 = sum(MSE1.conj().T) / Iter_n
    MSEav2 = sum(MSE2.conj().T) / Iter_n
    MSEav3 = sum(MSE3.conj().T) / Iter_n

    plt.plot(10 * np.log10(MSEav1), 'r', lw=0.5)
    plt.plot(10 * np.log10(MSEav2), 'b', lw=0.5)
    plt.plot(10 * np.log10(MSEav3), 'g', lw=0.5)
    plt.title("Average error per iteration", fontsize=15)
    plt.ylabel('dB', fontsize=15)
    plt.legend(('APA', 'RLS', 'NLMS'),
               loc='upper right', shadow=True)
    plt.show()
```







In the last graphs, the value of delta is changed 3 times(0.001,0.1,1), it is observable that there are not drastic changes in the different methods.

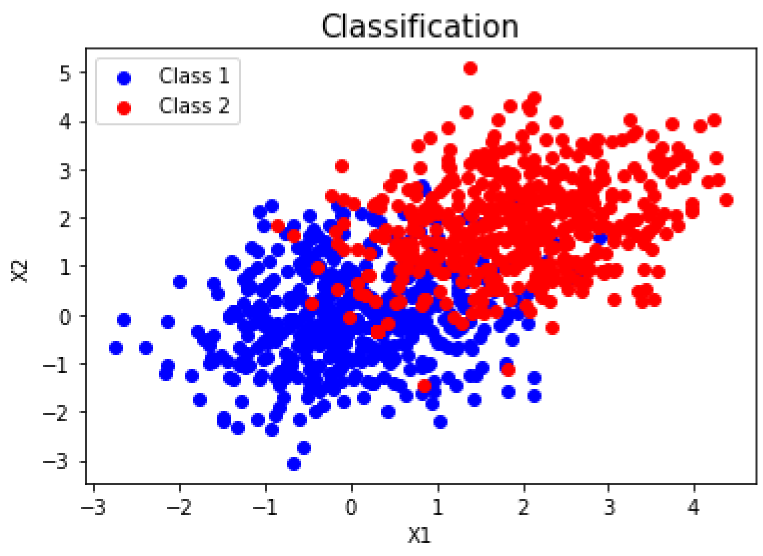
Question 4.3

In [1]:
```
import numpy as np
import math
from functools import reduce
from matplotlib import pyplot as plt
```

In [2]:
```
def multivariate_normal_pdf(x, mean, sigma):
    l = x.shape[0]
    det_S = np.linalg.det(sigma)
    norm_const = 1.0/((2.0*np.pi)**(l/2.0)*np.sqrt(det_S))
    inv_S = np.linalg.inv(sigma)
    a1 = np.dot(np.dot((x-mean), inv_S), (x-mean))

    return norm_const*np.exp(-(1.0/2.0)*a1)

def multivariate_normal_pdf_v2(x, mean, sigma):
    l = x.shape[1]
    det_S = np.linalg.det(sigma)
    norm_const = 1.0/((2.0*np.pi)**(l/2.0)*np.sqrt(det_S))
    inv_S = np.linalg.inv(sigma)
    a1    = np.sum(np.dot(x-mean, inv_S)*(x-mean), axis = 1)

    return norm_const*np.exp(-0.5*a1)
```

i.

In [3]:
```
N = 500 # Number of data points per class
S = np.array([[1, .25], [.25, 1]]) #covariance matrix
L = np.array([[0, 1], [0.005, 0]])
m1 = np.array([0, 0]) #mean for first class
m2 = np.array([2, 2]) #mean for second class

xtr1 = np.random.multivariate_normal(m1,S,N)
xtr2 = np.random.multivariate_normal(m2,S,N)


X = np.concatenate((xtr1, xtr2), axis = 0) #data_set
Y = np.concatenate((0*np.ones((N, 1)), 1*np.ones((N, 1))), axis = 0)

plt.figure(1)
plt.scatter(X[np.nonzero(Y == 0),0], X[np.nonzero(Y == 0),1], color = "b",label = "Class 1")
plt.scatter(X[np.nonzero(Y == 1),0], X[np.nonzero(Y == 1),1], color = "r",label = "Class 2")
plt.title("Classification", fontsize=15)
plt.legend(loc = 0)
plt.xlabel("X1");
plt.ylabel("X2");
```

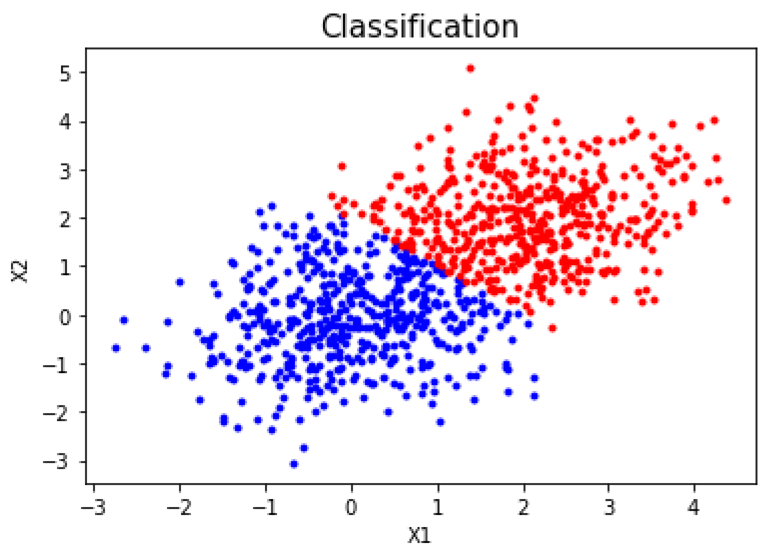


ii. & iii.

In [4]:
```
# Bayes classification of X
# Estimation of priori probabilities
p = 2*N
P2 = P1 = 0.5
p1 = np.zeros(p)
p2 = np.zeros(p)

# Estimation of pdf's for each data point
p1=multivariate_normal_pdf_v2(X,m1 ,S); # prior propability Gaussian_PDF
p2=multivariate_normal_pdf_v2(X,m2 ,S);

classf = np.zeros(p)

for i in range(0, p):
    if P1*p1[i] > P2*p2[i]:
        classf[i] = 0
    else:
        classf[i] = 1

# Error probability estimation
Pe = 0  # Probability of error
for i in range(0, p):
    if classf[i] != Y[i][0]:
        Pe += 1

Pe /= p
print('Pe: %f' % Pe)
```

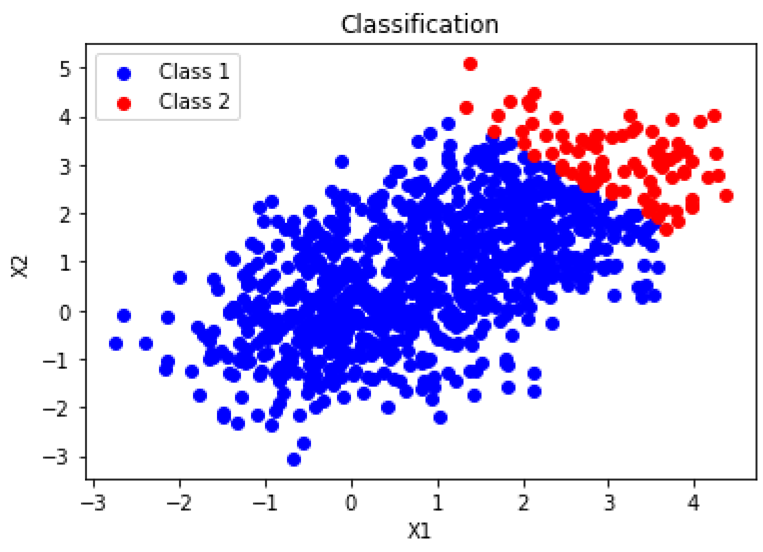
```
Pe: 0.106000
```

In [5]:
```
plt.figure(1)
plt.plot(X[np.nonzero(classf == 0),0], X[np.nonzero(classf == 0),1], '.b')
plt.plot(X[np.nonzero(classf == 1),0], X[np.nonzero(classf == 1),1], '.r')
plt.title("Classification", fontsize=15)

plt.xlabel("X1");
plt.ylabel("X2");
```



iv.

In [6]:
```
p1=multivariate_normal_pdf_v2(X,m1 ,S);
p2=multivariate_normal_pdf_v2(X,m2 ,S);

# Classification of the data points
class_loss = np.zeros(p)
for i in range(0, p):
    if L[0][1] * P1 * p1[i] > L[1][0] * P2 * p2[i]:
        class_loss[i] = 0
    else:
        class_loss[i] = 1

plt.figure(1)
plt.scatter(X[np.nonzero(class_loss == 0),0], X[np.nonzero(class_loss == 0),1], color = "b",label = "Class 1")
plt.scatter(X[np.nonzero(class_loss == 1),0], X[np.nonzero(class_loss == 1),1], color = "r",label = "Class 2")
plt.title(r"Classification", fontsize=12)
plt.legend(loc = 0)

plt.xlabel("X1");
plt.ylabel("X2");
```



v.

In [7]:
```
# Error probability estimation
Avgrisk = 0  # Average risk
for i in range(0, p):
    if class_loss[i] != Y[i][0]:
        if Y[i][0] == 0:
            Avgrisk = Avgrisk + L[0, 1]
        else:
            Avgrisk = Avgrisk + L[1, 0]
Avgrisk /= p
print('Avgrisk: %f' % Avgrisk)
```

```
Avgrisk: 0.002070
```

vi

By seen the graphs it can be said that Bayesian classification achieves results nearly to 10% error Compared to the classicBayes classification rule, the average risk minimization rule gives smaller values of average risk as well as very low daya points are categorized in class 2 this second case is due to the loss of class 2 of 0.005, so the points from this class give lower risk when missclassified. In the first scenario, the classification rule dictates for almost all the overlapping regions among both classes. This is because classification error on data steming from omega 2 is 'lesser' compared with another classification error on data derived from omega 1.