Assignment D: 4.1

In [ ]:

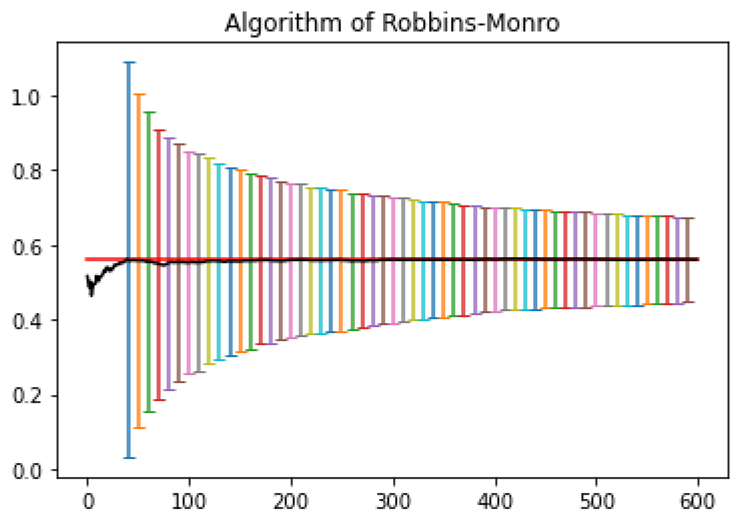
```
import numpy as np
from matplotlib import pyplot as plt
```

Part i: Applying Robbins-Monro algorithm

Part ii: Plotting the standard deviation

In [2]:

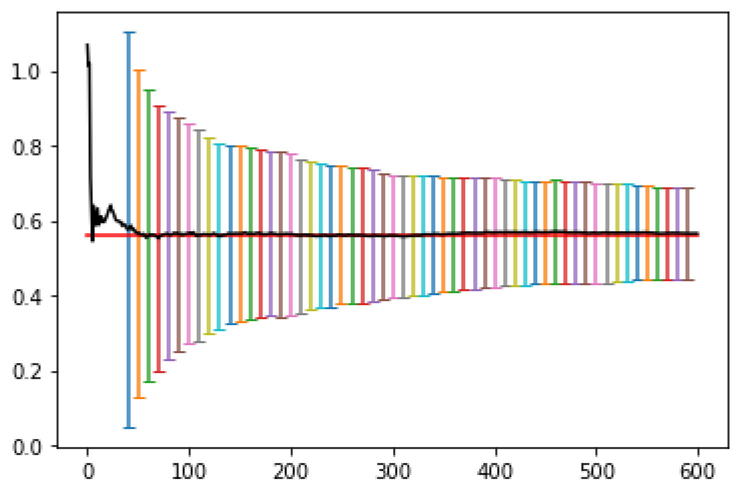
```
N = 600  # The total number of data points
L = 2  # The unknown vector's dimension
theta = np.random.randn(L, 1)  # Parameter undefined
w = np.zeros((L, 1))  # A Initial Estimate

Iter_number = 1000 #the total number of iterations
w_tot = np.zeros((N, Iter_number))
noise_var = 0.1
input_vec = lambda n: X[:, n].copy()

for It in range(0, Iter_number):  # It=1:Iter_number
    X = np.random.randn(L, N)
    noise = np.random.randn(N, 1) * np.sqrt(noise_var)
    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):
        myu = 1 / (i+1)  # Length of the step
        e = y[i] - np.dot(w.conj().T, input_vec(i))  # Calculation of errors
        w = w + myu * e * input_vec(i)
        w_tot[i][It] = w[0][0]

theta_1 = theta[0] * np.ones((N, 1))
plt. plot(theta_1, color='red')
plt.title("Algorithm of Robbins-Monro")
mean_w = np.mean(w_tot.conj().T, axis=0)
plt.plot(mean_w, color='k', linestyle='solid')


# "Part2"

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

plt.show()
```



Part 3: (myu/2)

In [7]:

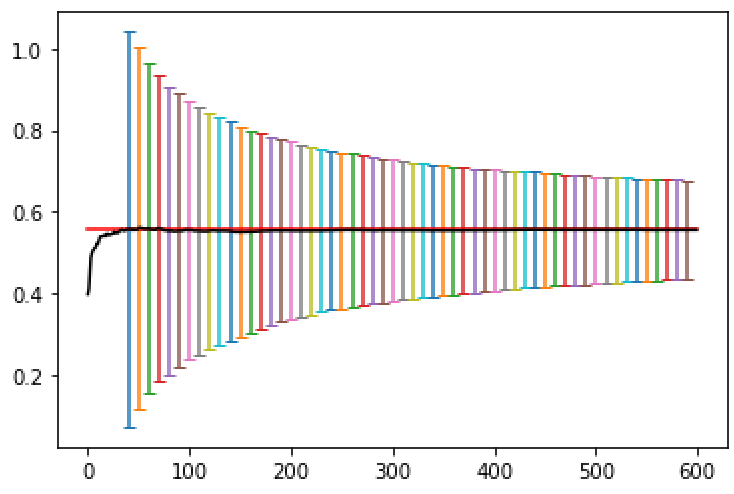
```
for It in range(0, Iter_number):  # It=1:Iter_number
    X = np.random.randn(L, N)
    noise = np.random.randn(N, 1) * np.sqrt(noise_var)
    y = np.zeros((N, 1))
    y[0:N] = np.dot(X[:, 0:N].conj().T, theta)
    y = y + noise
    w = np.zeros((L, 1))
    for i in range(0, N):
        myu = 1 / (i+1)*2  # Length of the step
        e = y[i] - np.dot(w.conj().T, input_vec(i))  # Calculation of errors
        w = w + myu * e * input_vec(i)
        w_tot[i][It] = w[0][0]

theta1 = theta[0] * np.ones((N, 1))
plt. plot(theta1, color='red')
meanw = np.mean(w_tot.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(w_tot[i, :], axis=0), capsize=3)

plt.show()
```



Part 3: (myu/0.7)

In [4]:

```
for It in range(0, Iter_number):  # It=1:Iter_number
    X = np.random.randn(L, N)
    noise = np.random.randn(N, 1) * np.sqrt(noise_var)
    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):
        myu = 1 / (i+1)*0.7  # Length of the step
        e = y[i] - np.dot(w.conj().T, input_vec(i))  # Calculation of errors
        w = w + myu * e * input_vec(i)
        w_tot[i][It] = w[0][0]

theta1 = theta[0] * np.ones((N, 1))
plt. plot(theta1, color='red')
meanw = np.mean(w_tot.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(w_tot[i, :], axis=0), capsize=3)

plt.show()
```



Comments

* According to the graph, we can see that if we divided the value of myu by 0.7 then it gives better error as compared to divided by 2.

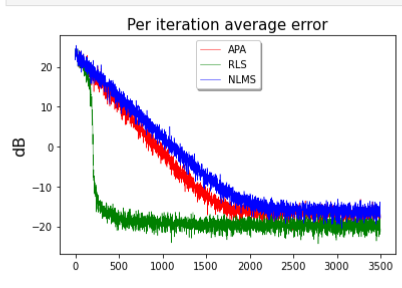
*Small learning rates cause the Robbins–Monro iterates to converge slowly, whereas large learning rates cause the iterates to numerically diverge.

In [ ]:

Assignment D: 4.2

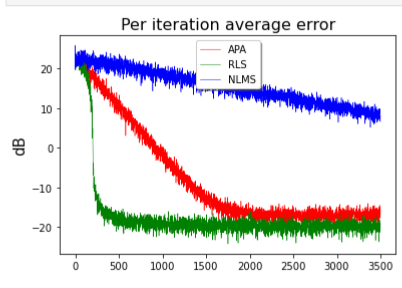
```python
import os
import sys
import numpy as np
from matplotlib import pyplot as plt
sys.path.append(os.getcwd())
sys.path.append('../')
import warnings
warnings.filterwarnings("ignore", category=RuntimeWarning)
```

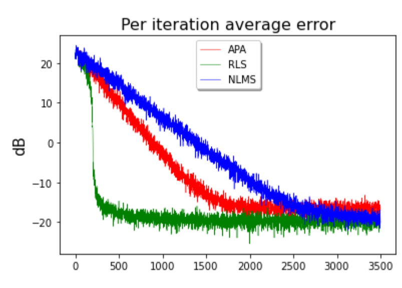
```python
L = 200   # The unknown vector's dimension
N = 3500   # Data Quantity
theta = np.random.randn(L, 1)   # Undefined parameter

Iter_number = 30
mse_1 = np.zeros((N, Iter_number))
mse_2 = np.zeros((N, Iter_number))
mse_3 = np.zeros((N, Iter_number))

noise_var = 0.01
epsilon_e = np.sqrt(2) * noise_var

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

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

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

for It in range(0, Iter_number):   # =1:Iter_number

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


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

    y = np.zeros((N, 1))
    y[0:N] = np.dot(X[:, 0:N].conj().T, theta)
    y = y + noise
    w = np.zeros((L, 1))
    myu = 0.2
    delta = 0.001
    q = 30   # The number of APA windows that were used
    for i in range(0, N):
        if i > q:
            qq = range(i, i - q, -1)   # qq=i:-1:i-q+1;

            y_vec = y[qq]
            Xq = input_vec(qq)
            Xq = np.reshape(Xq, newshape=(Xq.shape[0], Xq.shape[1]))
            e = y_vec - np.dot(Xq, w)
            e_ins = y[i] - np.dot(w.conj().T, input_vec(i))
            w = w + myu * np.dot(np.dot(Xq.conj().T, np.linalg.inv(delta*np.eye(q)+np.dot(Xq, Xq.conj().T))), e
            mse_1[i, It] = e_ins ** 2

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

        gamma = 1/(1+np.dot(input_vec(i).conj().T, np.dot(P, input_vec(i))))
        gi = np.dot(P, input_vec(i)) * gamma
        e = y[i] - np.dot(w.conj().T, input_vec(i))
        w = w + gi * e
        P = P - np.dot(gi, gi.conj().T)/gamma
        mse_2[i, It] = e ** 2

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

    for i in range(0, N):
        e = y[i] - np.dot(w.conj().T, input_vec(i))
        myu_n = myu / (delta+np.dot(input_vec(i).conj().T, input_vec(i)))
        w = w + myu_n * e * input_vec(i)
        mse_3[i, It] = e ** 2

mse_av1 = sum(mse_1.conj().T) / Iter_number
mse_av2 = sum(mse_2.conj().T) / Iter_number
mse_av3 = sum(mse_3.conj().T) / Iter_number

plt.plot(10 * np.log10(mse_av1), 'r', lw=0.5)
plt.plot(10 * np.log10(mse_av2), 'g', lw=0.5)
plt.plot(10 * np.log10(mse_av3), 'b', lw=0.5)
plt.title("Per iteration average error", fontsize=15)
plt.ylabel('dB', fontsize=16)
plt.legend(('APA', 'RLS', 'NLMS'),
           loc='upper center', shadow=True)
plt.show()
```

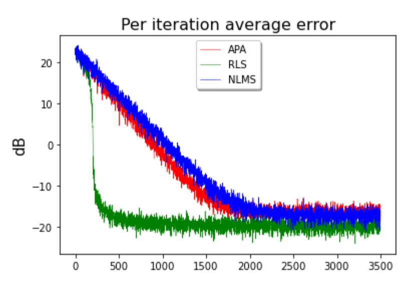


The finest performance comes from RLS

```python
#Chanhing the value of myu

L = 200   # The unknown vector's dimension
N = 3500   # Data Quantity
theta = np.random.randn(L, 1)   # Undefined parameter

Iter_number = 30
mse_1 = np.zeros((N, Iter_number))
mse_2 = np.zeros((N, Iter_number))
mse_3 = np.zeros((N, Iter_number))

noise_var = 0.01
epsilon_e = np.sqrt(2) * noise_var

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

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

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

myu = np.array([0.1,0.5,1])
delta = 0.001
for j in range(len(myu)):
    for It in range(0, Iter_number):   # =1:Iter_number

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


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

        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   # The number of APA windows that were used
        for i in range(0, N):
            if i > q:
                qq = range(i, i - q, -1)   # qq=i:-1:i-q+1;
                y_vec = y[qq]
                Xq = input_vec(qq)
                Xq = np.reshape(Xq, newshape=(Xq.shape[0], Xq.shape[1]))
                e = y_vec - np.dot(Xq, w)
                e_ins = y[i] - np.dot(w.conj().T, input_vec(i))
                w = w + myu[j] * np.dot(np.dot(Xq.conj().T, np.linalg.inv(delta*np.eye(q)+np.dot(Xq, Xq.conj().
                mse_1[i, It] = e_ins ** 2

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

            gamma = 1/(1+np.dot(input_vec(i).conj().T, np.dot(P, input_vec(i))))
            gi = np.dot(P, input_vec(i)) * gamma
            e = y[i] - np.dot(w.conj().T, input_vec(i))
            w = w + gi * e
            P = P - np.dot(gi, gi.conj().T)/gamma
            mse_2[i, It] = e ** 2

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

        for i in range(0, N):
            e = y[i] - np.dot(w.conj().T, input_vec(i))
            myu_n = myu[j] / (delta+np.dot(input_vec(i).conj().T, input_vec(i)))
            w = w + myu_n * e * input_vec(i)
            mse_3[i, It] = e ** 2

    mse_av1 = sum(mse_1.conj().T) / Iter_number
    mse_av2 = sum(mse_2.conj().T) / Iter_number
    mse_av3 = sum(mse_3.conj().T) / Iter_number

    plt.plot(10 * np.log10(mse_av1), 'r', lw=0.5)
    plt.plot(10 * np.log10(mse_av2), 'g', lw=0.5)
    plt.plot(10 * np.log10(mse_av3), 'b', lw=0.5)
    plt.title("Per iteration average error", fontsize=16)
    plt.ylabel('dB', fontsize=15)
    plt.legend(('APA', 'RLS', 'NLMS'),
               loc='upper center', shadow=True)
    plt.show()
```



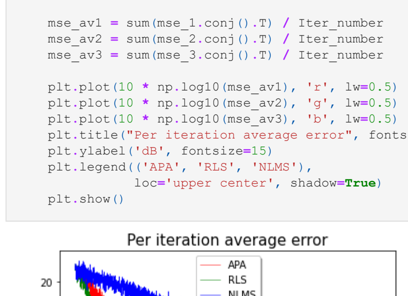


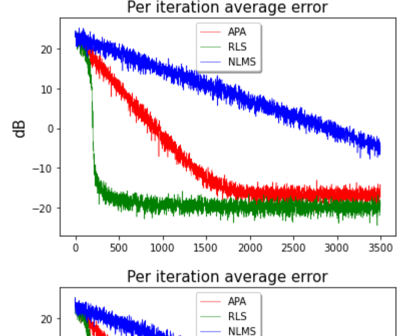


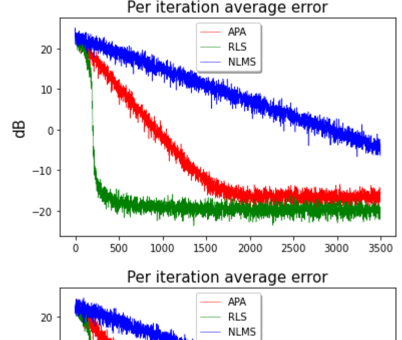
***The value of m changes with three different values(0.1,0.5,1) in the graphs above, demonstrating how m is related to learning rate; the larger it is, the faster it updates.**

***It can be observed that RLS is the most efficient recursion method, whereas NLMS is the slowest.¶**

```python
#changing delta

L = 200   # The unknown vector's dimension
N = 3500   # Data Quantity
theta = np.random.randn(L, 1)   # Undefined parameter

Iter_number = 30
mse_1 = np.zeros((N, Iter_number))
mse_2 = np.zeros((N, Iter_number))
mse_3 = np.zeros((N, Iter_number))

noise_var = 0.01
epsilon_e = np.sqrt(2) * noise_var

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

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

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

delta = np.array([0.001,0.1,1])
myu = 0.2
for j in range(len(delta)):
    for It in range(0, Iter_number):   # =1:Iter_number

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


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

        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   # The number of APA windows that were used
        for i in range(0, N):
            if i > q:
                qq = range(i, i - q, -1)   # qq=i:-1:i-q+1;
                y_vec = y[qq]
                Xq = input_vec(qq)
                Xq = np.reshape(Xq, newshape=(Xq.shape[0], Xq.shape[1]))
                e = y_vec - np.dot(Xq, w)
                e_ins = y[i] - np.dot(w.conj().T, input_vec(i))
                w = w + myu * np.dot(np.dot(Xq.conj().T, np.linalg.inv(delta[j]*np.eye(q)+np.dot(Xq, Xq.conj().
                mse_1[i, It] = e_ins ** 2

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

            gamma = 1/(1+np.dot(input_vec(i).conj().T, np.dot(P, input_vec(i))))
            gi = np.dot(P, input_vec(i)) * gamma
            e = y[i] - np.dot(w.conj().T, input_vec(i))
            w = w + gi * e
            P = P - np.dot(gi, gi.conj().T)/gamma
            mse_2[i, It] = e ** 2

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

        for i in range(0, N):
            e = y[i] - np.dot(w.conj().T, input_vec(i))
            myu_n = myu / (delta[j]+np.dot(input_vec(i).conj().T, input_vec(i)))
            w = w + myu_n * e * input_vec(i)
            mse_3[i, It] = e ** 2

    mse_av1 = sum(mse_1.conj().T) / Iter_number
    mse_av2 = sum(mse_2.conj().T) / Iter_number
    mse_av3 = sum(mse_3.conj().T) / Iter_number

    plt.plot(10 * np.log10(mse_av1), 'r', lw=0.5)
    plt.plot(10 * np.log10(mse_av2), 'g', lw=0.5)
    plt.plot(10 * np.log10(mse_av3), 'b', lw=0.5)
    plt.title("Per iteration average error", fontsize=15)
    plt.ylabel('dB', fontsize=15)
    plt.legend(('APA', 'RLS', 'NLMS'),
               loc='upper center', shadow=True)
    plt.show()
```

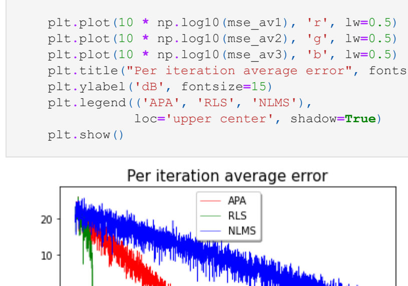


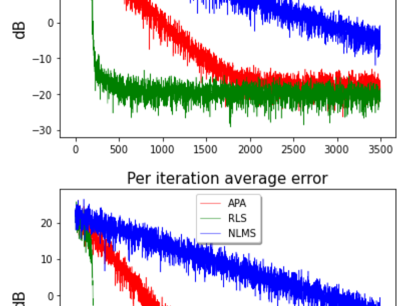


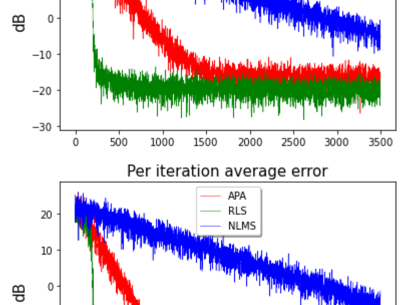


By changing the value of delta from [0.001,0.1,1], we can not observe much changes on the above three graphs

```python
# Changing the value of q

L = 200   # # The unknown vector's dimension
N = 3500   # Data Quantity
theta = np.random.randn(L, 1)   # Undefined parameter

Iter_number = 10
mse_1 = np.zeros((N, Iter_number))
mse_2 = np.zeros((N, Iter_number))
mse_3 = np.zeros((N, Iter_number))

noise_var = 0.01
epsilon_e = np.sqrt(2) * noise_var

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

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

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

q = np.array([10,50,100])
myu = 0.2
delta = 0.001
for j in range(len(q)):
    for It in range(0, Iter_number):   # =1:Iter_number

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


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

        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)   # qq=i:-1:i-q+1;

                y_vec = y[qq]
                Xq = input_vec(qq)
                Xq = np.reshape(Xq, newshape=(Xq.shape[0], Xq.shape[1]))
                e = y_vec - np.dot(Xq, w)
                e_ins = y[i] - np.dot(w.conj().T, input_vec(i))
                w = w + myu * np.dot(np.dot(Xq.conj().T, np.linalg.inv(delta*np.eye(q[j])+np.dot(Xq, Xq.conj().
                mse_1[i, It] = e_ins ** 2

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

            gamma = 1/(1+np.dot(input_vec(i).conj().T, np.dot(P, input_vec(i))))
            gi = np.dot(P, input_vec(i)) * gamma
            e = y[i] - np.dot(w.conj().T, input_vec(i))
            w = w + gi * e
            P = P - np.dot(gi, gi.conj().T)/gamma
            mse_2[i, It] = e ** 2

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

        for i in range(0, N):
            e = y[i] - np.dot(w.conj().T, input_vec(i))
            myu_n = myu / (delta+np.dot(input_vec(i).conj().T, input_vec(i)))
            w = w + myu_n * e * input_vec(i)
            mse_3[i, It] = e ** 2

    mse_av1 = sum(mse_1.conj().T) / Iter_number
    mse_av2 = sum(mse_2.conj().T) / Iter_number
    mse_av3 = sum(mse_3.conj().T) / Iter_number

    plt.plot(10 * np.log10(mse_av1), 'r', lw=0.5)
    plt.plot(10 * np.log10(mse_av2), 'g', lw=0.5)
    plt.plot(10 * np.log10(mse_av3), 'b', lw=0.5)
    plt.title("Per iteration average error", fontsize=15)
    plt.ylabel('dB', fontsize=15)
    plt.legend(('APA', 'RLS', 'NLMS'),
               loc='upper center', shadow=True)
    plt.show()
```







The parameter q is adjusted three times in the graphs above(10,50,100). Altering q expands the window, which increases the data utilised to predict. In this case, the only recursion method changing is APA.

Assignment D: 4.3

In [ ]:

```
import numpy as np
import math
from functools import reduce
from matplotlib import pyplot as plt
```

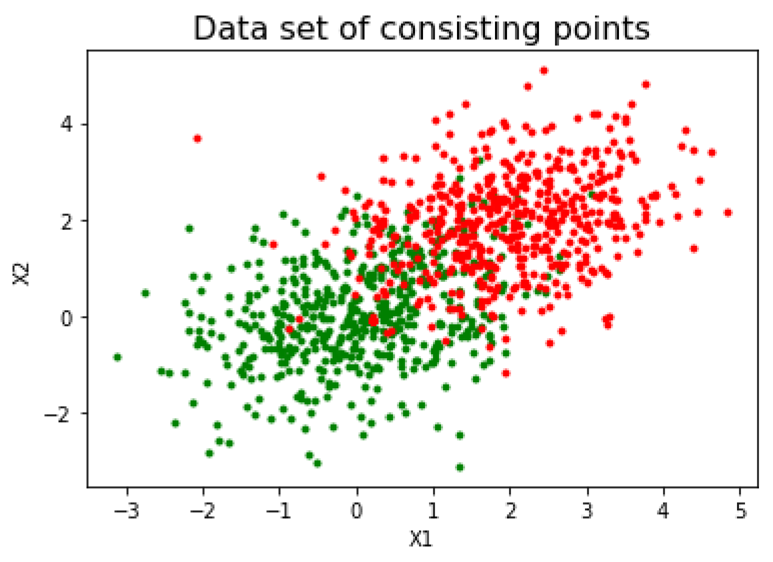
In [2]:

```
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)
```

Part i: Creating a data set and plotting it

In [3]:

```
m1 = np.array([0, 0]) #mean for the first-class
m2 = np.array([2, 2]) #mean for the second-class
S = np.array([[1, .25], [.25, 1]]) #covariance_matrix
L = np.array([[0, 1], [0.005, 0]])
N = 500 # Per class, the number of data points

xtr_1 = np.random.multivariate_normal(m1,S,N)
xtr_2 = np.random.multivariate_normal(m2,S,N)
ytr_1 = 0*np.ones((N,1))
ytr_2 = 1*np.ones((N,1))
X = np.concatenate((xtr_1, xtr_2), axis = 0) #data_set
Y = np.concatenate((ytr_1,ytr_2), axis = 0)

plt.figure(1)
plt.plot(X[np.nonzero(Y == 0),0], X[np.nonzero(Y == 0),1], '.g')
plt.plot(X[np.nonzero(Y == 1),0], X[np.nonzero(Y == 1),1], '.r')
plt.title(r'Data set of consisting points', fontsize=16)

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



Part ii: Bayes classification rule

Part iii: Estimating the probability of an error

Part iv: Plotted in multiple colors according to average risk minimization rule

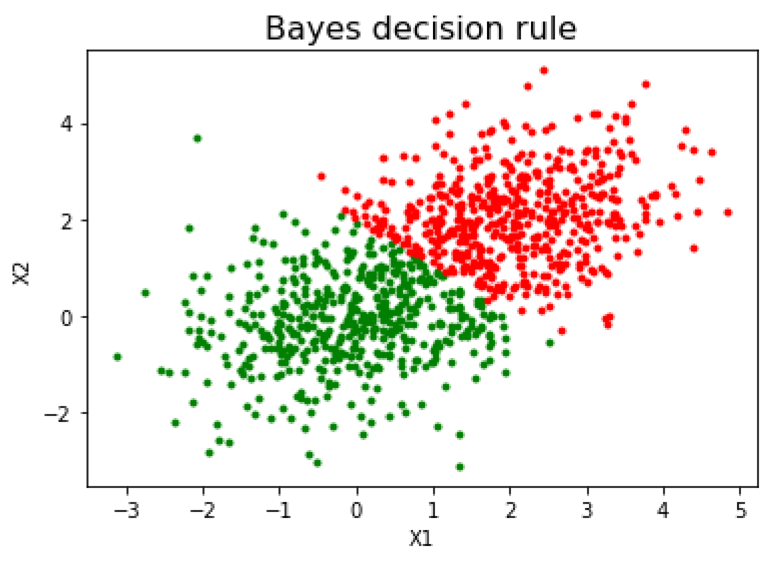
In [4]:

```
# (ii) X's Bayes classification
# Probability estimation based on prior probabilities
p = 2*N
P1 = 0.5
P2 = P1
p1 = np.zeros(p)
p2 = np.zeros(p)

# Computation of each data point's pdf

p1=multivariate_normal_pdf_v2(X,m1,S); #prior_propability*Gaussian_PDF
p2=multivariate_normal_pdf_v2(X,m2,S);
classification = np.zeros(p)
classes = np.zeros(p)

for i in range(0, p):
    if P1*p1[i] > P2*p2[i]:
        classification[i] = 0
    else:
        classification[i] = 1
# (iii) Estimating the probability of an error
Pe = 0  # Error probability
for i in range(0, p):   # =1:p
    if classes[i] != Y[i][0]:
        Pe += 1

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

```
Pe: 0.500000
```

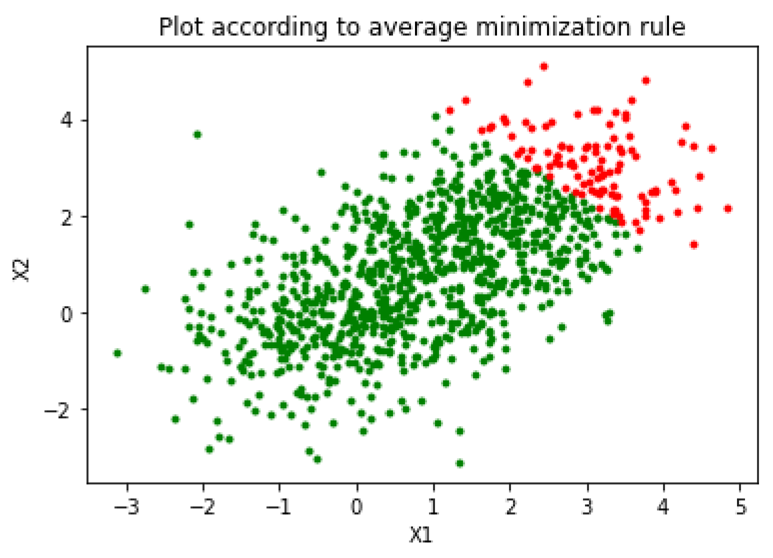
In [5]:

```
plt.figure(1)
plt.plot(X[np.nonzero(classification == 0),0], X[np.nonzero(classification == 0),1], '.g')
plt.plot(X[np.nonzero(classification == 1),0], X[np.nonzero(classification == 1),1], '.r')
plt.title(r'Bayes decision rule', fontsize=16)

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



In [6]:

```
p1=multivariate_normal_pdf_v2(X,m1 ,S); # Prior propability * Gaussian_PDF
p2=multivariate_normal_pdf_v2(X,m2 ,S);

# The data points are categorised.
classification_loss = np.zeros(p)
for i in range(0, p):
    if L[0][1] * P1 * p1[i] > L[1][0] * P2 * p2[i]:
        classification_loss[i] = 0
    else:
        classification_loss[i] = 1
#iv: Average risk minimization rule
plt.figure(1)
plt.plot(X[np.nonzero(classification_loss == 0),0], X[np.nonzero(classification_loss == 0),1], '.g')
plt.plot(X[np.nonzero(classification_loss == 1),0], X[np.nonzero(classification_loss == 1),1], '.r')
plt.title(r'Plot according to average minimization rule', fontsize=12)

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



Part v: Estimation of average risk for the loss matrix

In [7]:

```
Average_risk = 0
for i in range(0, p):
    if classification_loss[i] != Y[i][0]:
        if Y[i][0] == 0:
            Average_risk = Average_risk + L[0, 1]
        else:
            Average_risk = Average_risk + L[1, 0]
Average_risk/= p
print('Average_risk: %f' % Average_risk)
```

```
Average_risk: 0.002975
```

Part vi: Comments

The average risk minimization criterion reduces the average risk value in comparison to the maximum risk.

The probability of error obtained by the classical rules of Bayesian classification.

In the preceding case, the classification rules determine almost all

The area of overlap between the two classes favors w1. This is because the misclassification of data derived from w2 is cheap compared to the opposite misclassification.

The Bayesian classification has achieved a result of approximately 10% error, which can also be seen in the figure.

When data point is assigned by the average risk minimization rule, we see that very low data points have been classified as category 2. The reason is that the loss for category 2 is only 0.005, so Category 2 data points will provide low risk in the following situations of being misclassified compared to Category 1 with a loss value of 1. Therefore, based on the given data point mapping, the total risk is under.

In [ ]: