DM1 Logbook

26/10/2021

under stand this experiment

Model:

$$e(k) = d(k) - \underbrace{w^T(k)\, x(k)}_{\hat{y}(k)}$$

$$[\, w_1(k) \quad w_2(k) \cdots w_L(k) \,] \qquad \begin{bmatrix} x(k) \\ x(k-1) \\ \vdots \\ x(k-L+1) \end{bmatrix}$$

initially $W(0) = 0$

$$W(k+1) = W(k) + 2\mu X(k)\, e(k)$$

scalars for the adaptation gain

27/10/2021

① LMS

In order to implement LMS algorithm

```
e, w = LMS(x, dn, mu, L_f)
```

$N\times 1$ $N\times L$ $N\times 1$ $N\times 1$ degree of filter

$$0 < \mu < 1/LP_x$$

x is the input signal after passing the adaptive filter

$W_{adapt}^T \cdot X$

actual output

It is assumed that the desired impulse response (the unknown pulse) is $\frac{1}{k}exp(-(k-4)^2/4)$ with length $L=9$ FIR filter

$\therefore d = X * W$

the difference between them is the error

1/11/2021 1.1

My LMS algorithm:

```
# LMS algorithm
def LMS(x_in, d_in, m, L):  # x_in is the adaptive filter input, d_in is the desired response signal
    S = len(x_in)           # m is the adaptation gain, L is the length of the adaptive filter
    e = np.zeros(S)  # error array
    w = [[0] * L for _ in range(S+1)]  # weights, should be N*L matrix, the additional row is for the initial w
    zs = np.zeros(L-1)
    x_add0 = np.concatenate((zs, x_in))  # add L-1 zeros to the beginning of input, start with [0 0 0 0 0 0 0 0 x1]

    for k in range(S):
        x_k = x_add0[k-len(x_add0)-1+L:k-len(x_add0)-1:-1]  # take k to k+L-1 x_add0, and reverse their order
        e[k] = d_in[k] - x_k.dot(w[k])  # calculate the error between response signal and desired response signal
        w[k+1] = np.add(w[k], 2 * m * x_k * e[k])  # update the weight

    return e, w[1:S+1]  # return N*L weights matrix
```

In the requirement, w is a $N\times L$ matrix, but I set it to be $(N+1)\times L$ in the function, and return its second to last row. Since if we start to run the algorithm from

$x[8]$, that means we have 9 values $x[0], x[1], \cdots x[8]$, which fit the length of the filter. But this is not the practical case. In practice, we still have input before $x[0]$. Here, I add 8 zeros before $x[0]$, and the first x_k becomes

$$\left([\underbrace{0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0}_{8 \text{ zeros}} \quad x[0]] \right) \text{ in reverse order.}$$

In this case, I need $(N+1)$ rows of W to store the updated weight inside the function.

According to the requirement, I changed the length of filter, and the result is:

It is clear that the result of filter length $L=9$ is much better than those of $L=5$ and $L=11$.

$L=5$ means undermodelling, $L=11$ means overfitting. Both cases lead to a worse result comparing with the suitable length $L=9$.

2/11/2021 1.2

```
Loop = 200
all_error = np.zeros([Loop, 200])
all_weight_error = np.zeros([Loop, 200])

for q in range(Loop):
    L_filter = 9  # length of the adaptive filter
    all_error[q, :], all_weight_error[q, :] = main(L_filter)

mean_error = np.mean(all_error**2, 0)
mean_learning_rate = np.mean(all_weight_error, 0)
```

Two Loop × N matrixs are used to store the output of the algorithm, then take the average of these two matrixs (average of their rows), the result is shown:

These are the results of 200 loops, due to curiosity, I tried 20000 loops.

It is clear that the error and weight error decrease "linearly" in the logarithmic axis.

1.3 By changing the value of μ to be $\begin{cases} 0.5 \\ 0.05 \\ 0.001 \end{cases}$

The result is:

It is clear that a big μ ($\mu = 0.5 \gg \frac{1}{3} \times \frac{1}{L}$) makes the system becomes really unstable.

I change 0.5 to be $0.037 \approx \frac{1}{3} \times \frac{1}{L} = \frac{1}{27}$, in order to observe how $\mu = 0.01$ and $\mu = 0.001$ change the output

The result shows that a smaller will not make the result becomes unstable, but will make smaller changes to the result. An extremely small $\mu = 0.001$ almost makes no changes to the result.

3/11/2021

1.4 By adding a noise, which reach a SNR of 100

∴ noise variance is $\frac{1}{100}$

∴

```
noise_1 = 1 / 10 * np.random.randn(sample + L_filter_un - 1)
```

The result is:

The result shows that the performance of LMS dealing with noise is not so good.

② LMS & NLMS + real world speech signal

Since the input signal becomes a real world speech signal, μ should be changed

μ, the adaptation gain, is a tradeoff between convergence speed and misadjustment.

high (low) μ implies fast (slow) convergence and small (large) misadjustment.

By changing the value of μ, I find a μ which gives almost the best result. I don't know how this value corresbonds to the length of filter or the power of input signal. I'm going to think further in the formal lab report.

$$\mu = \frac{1}{0.5592 \times L}$$

NLMS: this is the normalized LMS

Because the power of real world speech does not have a limitation, it can be quite large or small, which makes it hard for LMS to process. By normalizing the input speech signal, the performance will be better.

I found a small problem in the assignment.

$$\boldsymbol{w}(k+1) = \boldsymbol{w}(k) + \frac{2\mu}{1+||\boldsymbol{x}(k)||^2}\boldsymbol{x}(k)e(k)$$

here has a 1

After talking with GTA, we agree that theoretically, this 1 should be removed. This 1 is used to avoid 0 $||x(k)||^2$, but due to $||x(k)||$ is a quite small value, adding 1 has a great influence to it.

Therefore, I set up a if to avoid 0 $||x(k)||^2$:

```
temp = np.linalg.norm(x_k)
if temp != 0:
    x_normalized = x_k / temp ** 2
else:
    x_normalized = x_k

e[k] = d_in[k] - np.dot(w[k], x_k)  # calculate the error between desired response signal and response signal
w[k + 1] = np.add(w[k], 2 * m * x_normalized * e[k])  # update the weight
```

The result is:

It is clear that the performance of NLMS is much better than LMS.

As for the complexity,

LMS: $e(k) = \underbrace{d(k) - \underbrace{w(k) \cdot x(k)}_{L}}_{L}$

length: L

$w(k+1) = \underbrace{w(k) + \underbrace{2\mu\, x(k)\, e(k)}_{L+1}}_{L}$ $\Rightarrow 4L+1 \Rightarrow O(L)$

Same

NLMS: $e(k) = \underbrace{d(k) - \underbrace{w(k) \cdot x(k)}_{L}}_{L}$

$w(k+1) = \underbrace{w(k) + \frac{\overbrace{2\mu\, x(k)\, e(k)}^{L+1}}{\underbrace{\|x(k)\|_2^2}_{L+1}}}_{L}$ $\Rightarrow 5L+2 \Rightarrow O(L)$

For a fixed point precision LMS filter, the temporal and hardware complexities are significantly less than that of a floating point filter, while the performance can match that of the floating point filter.

4/11/2021

3.1 RLS

```
def compute_error(x, d, w):
    return (d - np.dot(w, x))

def compute_kalman_gain(x, last_rinv, forget_factor):
    x = x[:, np.newaxis]
    a = np.matmul(last_rinv, x)

    return a/(forget_factor + np.matmul(x.T, a))

def compute_rinv(x, last_rinv, kalman_gain, forget_factor):
    x = x[:, np.newaxis]
    a = np.matmul(x.T, last_rinv)

    b = np.matmul(kalman_gain, a)
    return (1/forget_factor)*(last_rinv - b)

def rls(x, d, X, L, forget_factor):
    weights = [np.zeros(L)]
    z = np.zeros(L)
    rinv = np.eye(L)

    errors = []
    N = len(x)
    for k in range(0, N - L):
        x_k = x[k:k+L]
        d_k = x[k]
        w_k = weights[k]

        e_k = compute_error(x_k, d_k, w_k)

        kalman_gain = compute_kalman_gain(x_k, rinv, forget_factor)

        new_weights = w_k + kalman_gain.flatten()*e_k

        rinv = compute_rinv(x_k, rinv, kalman_gain, forget_factor)

        weights.append(new_weights)
        errors.append(e_k)

    return np.array(weights), np.array(errors)
```

$$\boldsymbol{g}(k) = \frac{\boldsymbol{R}^{-1}(k-1)\boldsymbol{x}(k)}{\lambda + \boldsymbol{x}^T(k)\boldsymbol{R}^{-1}(k-1)\boldsymbol{x}(k)}$$

$$\boldsymbol{R}^{-1}(k) = \frac{1}{\lambda}[\boldsymbol{R}^{-1}(k-1) - \underbrace{\boldsymbol{g}(k)\underbrace{\boldsymbol{x}^T(k)\boldsymbol{R}^{-1}(k-1)}_{a}}_{b}]$$

Initialization : W is a $L \times 1$ zero matrix $\overbrace{[0\ 0\ 0\ \ldots\ 0\ 0\ 0]}^{L}$

R^{-1} is initialized as a $L \times L$ I matrix

$\rightarrow e(k) = d(k) - w(k) \cdot x(k)$

$$\boldsymbol{w}(k+1) = \boldsymbol{w}(k) + \boldsymbol{g}(k)e(k)$$

Complexity : e_k : $O(L)$ new_weights : $O(L)$

kalman_gain : $O(L^2)$ rinv : $O(L^2)$

$\therefore \Downarrow$

$O(L^2)$

3.2 Repeat 1.1

```python
def RLS(x_in, d_in, lamb, L):
    S = len(x_in)
    e = np.zeros(S)  # error array
    w = [[0] * L for _ in range(S + 1)]  # weights, should be N*L matrix, the additional row is for the initial w
    zs = np.zeros(L - 1)
    x_add0 = np.concatenate((zs, x_in))
    rinv = np.eye(L)
    for k in range(S):
        x_k = x_add0[k - len(x_add0) - 1 + L:k - len(x_add0) - 1:-1]  # take k to k+L-1 x_add0, and reverse their order
        e[k] = compute_error(x_k, d_in[k], w[k])
        kalman_gain = compute_kalman_gain(x_k, rinv, lamb)
        w[k+1] = w[k] + kalman_gain.flatten() * e[k]
        rinv = compute_rinv(x_k, rinv, kalman_gain, lamb)
    return e, w[1:S + 1]
```

I found another problem in the assignment sheet, the provided algorithm does not reverse the order of X_k.

After fixing this problem, I get the following result:

It seems that overfitting problem is not obvious in RLS

Repeat 1.2 run the code 200 times:

Therefore, overfitting is not a problem in RLS, while undermodelling is still a problem.

Repeat 1.4

It is clear that noise also has significant influence to the system.

3.3

It is clear that as $\lambda \downarrow$, the performance becomes better

1.1

```
# -*- coding: utf-8 -*-
"""

"""

import numpy as np
import math
import matplotlib.pyplot as plt


"_____________________________________________________Sub
Functions_____________________________________________________"


def compute_error(w_k, w_o):  # Normalised weight error vector norm
    numerator = np.sum((w_k - w_o) ** 2)
    denominator = np.sum(w_o ** 2)
    return numerator / denominator


# LMS algorithm
def LMS(x_in, d_in, m, L):  # x_in is the adaptive filter input, d_in
is the desired response signal
    S = len(x_in)           # m is the adaptation gain, L is the
length of the adaptive filter
    e = np.zeros(S)  # error array
    w = [[0] * L for _ in range(S+1)]  # weights, should be N*L
matrix, the additional row is for the initial w
    zs = np.zeros(L-1)
    x_add0 = np.concatenate((zs, x_in))  # add L-1 zeros to the
beginning of input, start with [0 0 0 0 0 0 0 0 x1]

    for k in range(S):
        x_k = x_add0[k-len(x_add0)-1+L:k-len(x_add0)-1:-1]  # take k
to k+L-1 x_add0, and reverse their order
        e[k] = d_in[k] - x_k.dot(w[k])  # calculate the error between
response signal and desired response signal
        w[k+1] = np.add(w[k], 2 * m * x_k * e[k])  # update the
weight

    return e, w[1:S+1]  # return N*L weights matrix


"_____________________________________________________Main
function_____________________________________________________"


def main(L_f):
    size = 200  # size of data array
    L_f_un = 9  # length of the unknown filter
    mu = 1 / (3 * L_f)  # adaptation gain, input signal has a unit
power

    x = np.random.randn(size)  # generate the input signal with unit
power and zero mean

    # This is the theoretical optimized w, use it to generate desired
response signal
    opt_w = np.zeros(L_f_un)
```

```
    for i in range(L_f_un):
        opt_w[i] = 1/(i+1) * math.exp(-((i-4)**2)/4)
    dn = np.convolve(x, opt_w, mode="full")  # desired response
signal

    # Calculate
    e, w = LMS(x, dn, mu, L_f)

    # Compute the learning rate
    w_e = np.zeros(size)
    opt_w_fit = np.zeros(L_f)  # If the length of unknown filter and
adaptive filter are not the same, they cannot
    for i in range(L_f):       # be used in 'compute_error', hence
fit the length of opt_w with adaptive filter
        opt_w_fit[i] = 1 / (i + 1) * math.exp(-((i - 4) ** 2) / 4)
    for i in range(size):
        w_e[i] = compute_error(w[i], opt_w_fit)

    return e, w_e  # return N*L weights matrix


"_____________________________________________________Plot
Diagrams_____________________________________________________"


if __name__ == "__main__":
    L_filter = 9  # length of the adaptive filter
    error_equal, weight_error_equal = main(L_filter)

    L_filter = 5  # length of the adaptive filter
    error_low, weight_error_low = main(L_filter)

    L_filter = 11  # length of the adaptive filter
    error_high, weight_error_high = main(L_filter)

    plt.figure(1)
    plt.semilogy(error_equal**2, label="Filter Length = 9")
    plt.semilogy(error_low**2, label="Filter Length = 5")
    plt.semilogy(error_high**2, label="Filter Length = 11")
    plt.xlabel("Sample Number")
    plt.ylabel("Power of error")
    plt.title("Error between response signal and desired response
signal")
    plt.legend()

    plt.figure(2)
    plt.semilogy(weight_error_equal, label="Filter Length = 9")
    plt.semilogy(weight_error_low, label="Filter Length = 5")
    plt.semilogy(weight_error_high, label="Filter Length = 11")
    plt.xlabel("Sample Number")
    plt.ylabel("Error Rate")
    plt.title("Normalised weight error vector norm")
    plt.legend()

    plt.show()
```

1.2

```python
# -*- coding: utf-8 -*-
"""

"""

import numpy as np
import math
import matplotlib.pyplot as plt


"_____________________________________________________Sub
Functions_____________________________________________________"


def compute_error(w_k, w_o):  # Normalised weight error vector norm
    numerator = np.sum((w_k - w_o) ** 2)
    denominator = np.sum(w_o ** 2)
    return numerator / denominator


# LMS algorithm
def LMS(x_in, d_in, m, L):  # x_in is the adaptive filter input, d_in
is the desired response signal
    S = len(x_in)           # m is the adaptation gain, L is the
length of the adaptive filter
    e = np.zeros(S)  # error array
    w = [[0] * L for _ in range(S+1)]  # weights, should be N*L
matrix, the additional row is for the initial w
    zs = np.zeros(L-1)
    x_add0 = np.concatenate((zs, x_in))  # add L-1 zeros to the
beginning of input, start with [0 0 0 0 0 0 0 0 x1]

    for k in range(S):
        x_k = x_add0[k-len(x_add0)-1+L:k-len(x_add0)-1:-1]  # take k
to k+L-1 x_add0, and reverse their order
        e[k] = d_in[k] - x_k.dot(w[k])  # calculate the error between
response signal and desired response signal
        w[k+1] = np.add(w[k], 2 * m * x_k * e[k])  # update the
weight

    return e, w[1:S+1]  # return N*L weights matrix


"_____________________________________________________Main
function_____________________________________________________"


def main(L_f):
    size = 200  # size of data array
    L_f_un = 9  # length of the unknown filter
    mu = 1 / (3 * L_f)  # adaptation gain, input signal has a unit
power

    x = np.random.randn(size)  # generate the input signal with unit
power and zero mean

    # This is the theoretical optimized w, use it to generate desired
response signal
    opt_w = np.zeros(L_f_un)
    for i in range(L_f_un):
        opt_w[i] = 1/(i+1) * math.exp(-((i-4)**2)/4)
```

```
    dn = np.convolve(x, opt_w, mode="full")  # desired response
signal

    # Calculate
    e, w = LMS(x, dn, mu, L_f)

    # Compute the learning rate
    w_e = np.zeros(size)
    opt_w_fit = np.zeros(L_f)  # If the length of unknown filter and
adaptive filter are not the same, they cannot
    for i in range(L_f):       # be used in 'compute_error', hence
fit the length of opt_w with adaptive filter
        opt_w_fit[i] = 1 / (i + 1) * math.exp(-((i - 4) ** 2) / 4)
    for i in range(size):
        w_e[i] = compute_error(w[i], opt_w_fit)

    return e, w_e  # return N*L weights matrix


"_____________________________________________________Plot
Diagrams_____________________________________________________"


if __name__ == "__main__":
    Loop = 200
    all_error = np.zeros([Loop, 200])
    all_weight_error = np.zeros([Loop, 200])

    for q in range(Loop):
        L_filter = 9  # length of the adaptive filter
        all_error[q, :], all_weight_error[q, :] = main(L_filter)

    mean_error = np.mean(all_error**2, 0)
    mean_learning_rate = np.mean(all_weight_error, 0)

    plt.figure(1)
    plt.semilogy(mean_error, label="Error")
    plt.xlabel("Sample Number")
    plt.ylabel("Power of error")
    plt.title("Error between response signal and desired response
signal")
    plt.legend()

    plt.figure(2)
    plt.semilogy(mean_learning_rate, label="Weight Error")
    plt.xlabel("Sample Number")
    plt.ylabel("Error Rate")
    plt.title("Normalised weight error vector norm")
    plt.legend()

    plt.show()
```

1.3

```
# -*- coding: utf-8 -*-
"""

"""
```

```
import numpy as np
import math
import matplotlib.pyplot as plt


"_____________________________________________________Sub
Functions_____________________________________________________"


def compute_error(w_k, w_o):  # Normalised weight error vector norm
    numerator = np.sum((w_k - w_o) ** 2)
    denominator = np.sum(w_o ** 2)
    return numerator / denominator


# LMS algorithm
def LMS(x_in, d_in, m, L):  # x_in is the adaptive filter input, d_in
is the desired response signal
    S = len(x_in)           # m is the adaptation gain, L is the
length of the adaptive filter
    e = np.zeros(S)  # error array
    w = [[0] * L for _ in range(S+1)]  # weights, should be N*L
matrix, the additional row is for the initial w
    zs = np.zeros(L-1)
    x_add0 = np.concatenate((zs, x_in))  # add L-1 zeros to the
beginning of input, start with [0 0 0 0 0 0 0 0 x1]

    for k in range(S):
        x_k = x_add0[k-len(x_add0)-1+L:k-len(x_add0)-1:-1]  # take k
to k+L-1 x_add0, and reverse their order
        e[k] = d_in[k] - x_k.dot(w[k])  # calculate the error between
response signal and desired response signal
        w[k+1] = np.add(w[k], 2 * m * x_k * e[k])  # update the
weight

    return e, w[1:S+1]  # return N*L weights matrix


"_____________________________________________________Main
function_____________________________________________________"


def main(L_f, mu):
    size = 200  # size of data array
    L_f_un = 9  # length of the unknown filter

    x = np.random.randn(size)  # generate the input signal with unit
power and zero mean

    # This is the theoretical optimized w, use it to generate desired
response signal
    opt_w = np.zeros(L_f_un)
    for i in range(L_f_un):
        opt_w[i] = 1/(i+1) * math.exp(-((i-4)**2)/4)
    dn = np.convolve(x, opt_w, mode="full")  # desired response
signal

    # Calculate
    e, w = LMS(x, dn, mu, L_f)

    # Compute the learning rate
```

```
    lr = np.zeros(size)
    opt_w_fit = np.zeros(L_f)  # If the length of unknown filter and
adaptive filter are not the same, they cannot
    for i in range(L_f):       # be used in 'compute_error', hence
fit the length of opt_w with adaptive filter
        opt_w_fit[i] = 1 / (i + 1) * math.exp(-((i - 4) ** 2) / 4)
    for i in range(size):
        lr[i] = compute_error(w[i], opt_w_fit)

    return e, lr  # return N*L weights matrix


"_____________________________________________________Plot
Diagrams_____________________________________________________"


if __name__ == "__main__":
    Loop = 200

    all_error_m1 = np.zeros([Loop, 200])
    all_error_m2 = np.zeros([Loop, 200])
    all_error_m3 = np.zeros([Loop, 200])
    all_learning_rate_m1 = np.zeros([Loop, 200])
    all_learning_rate_m2 = np.zeros([Loop, 200])
    all_learning_rate_m3 = np.zeros([Loop, 200])

    for q in range(Loop):
        L_filter = 9  # length of the adaptive filter
        mu1 = 0.037  # adaptation gain, input signal has a unit power
        all_error_m1[q, :], all_learning_rate_m1[q, :] =
main(L_filter, mu1)
        mu2 = 0.01  # adaptation gain, input signal has a unit power
        all_error_m2[q, :], all_learning_rate_m2[q, :] =
main(L_filter, mu2)
        mu3 = 0.001  # adaptation gain, input signal has a unit power
        all_error_m3[q, :], all_learning_rate_m3[q, :] =
main(L_filter, mu3)

    mean_error_m1 = np.mean(all_error_m1**2, 0)
    mean_error_m2 = np.mean(all_error_m2**2, 0)
    mean_error_m3 = np.mean(all_error_m3**2, 0)
    mean_learning_rate_m1 = np.mean(all_learning_rate_m1, 0)
    mean_learning_rate_m2 = np.mean(all_learning_rate_m2, 0)
    mean_learning_rate_m3 = np.mean(all_learning_rate_m3, 0)

    plt.figure(1)
    plt.semilogy(mean_error_m1, label="adaptation gain = 0.037")
    plt.semilogy(mean_error_m2, label="adaptation gain = 0.01")
    plt.semilogy(mean_error_m3, label="adaptation gain = 0.001")
    plt.xlabel("Sample Number")
    plt.ylabel("Power of error")
    plt.title("Error between response signal and desired response
signal")
    plt.legend()

    plt.figure(2)
    plt.semilogy(mean_learning_rate_m1, label="adaptation gain =
0.037")
    plt.semilogy(mean_learning_rate_m2, label="adaptation gain =
0.01")
    plt.semilogy(mean_learning_rate_m3, label="adaptation gain =
```

```
0.001")
    plt.xlabel("Sample Number")
    plt.ylabel("Error Rate")
    plt.title("Normalised weight error vector norm")
    plt.legend()

    plt.show()
```

1.4

```
# -*- coding: utf-8 -*-
"""

"""

import numpy as np
import math
import matplotlib.pyplot as plt


"_____________________________________________________Sub
Functions_____________________________________________________"


def compute_error(w_k, w_o):  # Normalised weight error vector norm
    numerator = np.sum((w_k - w_o) ** 2)
    denominator = np.sum(w_o ** 2)
    return numerator / denominator


# LMS algorithm
def LMS(x_in, d_in, m, L):  # x_in is the adaptive filter input, d_in
is the desired response signal
    S = len(x_in)           # m is the adaptation gain, L is the
length of the adaptive filter
    e = np.zeros(S)  # error array
    w = [[0] * L for _ in range(S+1)]  # weights, should be N*L
matrix, the additional row is for the initial w
    zs = np.zeros(L-1)
    x_add0 = np.concatenate((zs, x_in))  # add L-1 zeros to the
beginning of input, start with [0 0 0 0 0 0 0 0 x1]

    for k in range(S):
        x_k = x_add0[k-len(x_add0)-1+L:k-len(x_add0)-1:-1]  # take k
to k+L-1 x_add0, and reverse their order
        e[k] = d_in[k] - np.dot(w[k], x_k)  # calculate the error
between response signal and desired response signal
        w[k+1] = np.add(w[k], 2 * m * x_k * e[k])  # update the
weight

    return e, w[1:S+1]  # return N*L weights matrix


"_____________________________________________________Main
function_____________________________________________________"


def main(size, L_f, L_f_un, N):
    mu = 1 / (3 * L_f)  # adaptation gain, input signal has a unit
```

```
power
    x = np.random.randn(size)  # generate the input signal with unit
power and zero mean

    # This is the theoretical optimized w, use it to generate desired
response signal
    opt_w = np.zeros(L_f_un)
    for i in range(L_f_un):
        opt_w[i] = 1/(i+1) * math.exp(-((i-4)**2)/4)
    dn = np.add(np.convolve(x, opt_w, mode="full"), N)  # desired
response signal

    # Calculate
    e, w = LMS(x, dn, mu, L_f)

    # Compute the learning rate
    lr = np.zeros(size)
    for i in range(size):
        lr[i] = compute_error(w[i], opt_w)

    return e, lr  # return N*L weights matrix


"_____________________________________________________Plot
Diagrams_____________________________________________________"


if __name__ == "__main__":
    Loop = 300
    sample = 200  # sample number

    all_error = np.zeros([Loop, sample])
    all_learning_rate = np.zeros([Loop, sample])
    all_error_noise = np.zeros([Loop, sample])
    all_learning_rate_noise = np.zeros([Loop, sample])

    for q in range(Loop):
        L_filter = 9  # length of the adaptive filter
        L_filter_un = 9  # length of the unknown filter

        # generate noise that leads to a 20dB SNR, size of dn is
size+L_f_un-1
        noise_0 = np.zeros(200 + L_filter_un - 1)
        noise_1 = 1 / 10 * np.random.randn(sample + L_filter_un - 1)

        all_error[q, :], all_learning_rate[q, :] = main(sample,
L_filter, L_filter_un, noise_0)
        all_error_noise[q, :], all_learning_rate_noise[q, :] =
main(sample, L_filter, L_filter_un, noise_1)

    mean_error = np.mean(all_error**2, 0)
    mean_learning_rate = np.mean(all_learning_rate, 0)
    mean_error_noise = np.mean(all_error_noise**2, 0)
    mean_learning_rate_noise = np.mean(all_learning_rate_noise, 0)

    plt.figure(1)
    plt.semilogy(mean_error, label="Without Noise")
    plt.semilogy(mean_error_noise, label="With Noise")
    plt.xlabel("Sample Number")
    plt.ylabel("Power of error")
    plt.title("Error between response signal and desired response
```

```
signal")
    plt.legend()

    plt.figure(2)
    plt.semilogy(mean_learning_rate, label="Without Noise")
    plt.semilogy(mean_learning_rate_noise, label="With Noise")
    plt.xlabel("Sample Number")
    plt.ylabel("Error Rate")
    plt.title("Normalised weight error vector norm")
    plt.legend()

    plt.show()
```

2.0

```
# -*- coding: utf-8 -*-
"""

"""

import numpy as np
import math
import matplotlib.pyplot as plt
import scipy.io

"_____________________________________________________Sub
Functions_____________________________________________________"


def compute_error(w_k, w_o):  # Normalised weight error vector norm
    numerator = np.sum((w_k - w_o) ** 2)
    denominator = np.sum(w_o ** 2)
    return numerator / denominator


# LMS algorithm
def LMS(x_in, d_in, m, L):  # x_in is the adaptive filter input, d_in
is the desired response signal
    S = len(x_in)  # m is the adaptation gain, L is the length of the
adaptive filter
    e = np.zeros(S)  # error array
    w = [[0] * L for _ in range(S + 1)]  # weights, should be N*L
matrix, the additional row is for the initial w
    zs = np.zeros(L - 1)
    x_add0 = np.concatenate((zs, x_in))  # add L-1 zeros to the
beginning of input, start with [0 0 0 0 0 0 0 0 x1]

    for k in range(S):
        x_k = x_add0[k - len(x_add0) - 1 + L:k - len(x_add0) - 1:-1]
# take k to k+L-1 x_add0, and reverse their order
        e[k] = d_in[k] - np.dot(w[k], x_k)  # calculate the error
between response signal and desired response signal
        w[k + 1] = np.add(w[k], 2 * m * x_k * e[k])  # update the
weight

    return e, w[1:S + 1]  # return N*L weights matrix


"_____________________________________________________Main
```

```
function_____________________________________________________"


def main(x, mu):
    size = len(x)
    L_f = 9
    L_f_un = 9

    P_x = np.sum(x ** 2) / size

    # This is the theoretical optimized w, use it to generate desired
response signal
    opt_w = np.zeros(L_f_un)
    for i in range(L_f_un):
        opt_w[i] = 1 / (i + 1) * math.exp(-((i - 4) ** 2) / 4)
    dn = np.convolve(x, opt_w, mode="full")  # desired response
signal

    # Calculate
    e_LMS, w_LMS = LMS(x, dn, mu, L_f)

    # Compute the normalised weight error vector norm
    lr_LMS = np.zeros(size)
    lr_NLMS = np.zeros(size)
    for i in range(size):
        lr_LMS[i] = compute_error(w_LMS[i], opt_w)

    return e_LMS, lr_LMS  # return N*L weights matrix


"_____________________________________________________Plot
Diagrams_____________________________________________________"

if __name__ == "__main__":
    s1 = scipy.io.loadmat('s1.mat')
    s1 = s1['s1']
    s1 = np.array(s1).flatten()
    L_f = 9

    P_x = np.sum(s1 ** 2)
    print(P_x)
    print(P_x / len(s1))


    m2 = 1 / (0.5592 * L_f)  # adaptation gain, input signal has a
unit power
    all_error_LMS2, all_learning_rate_LMS2 = main(s1, m2)


    plt.figure(1)

    plt.semilogy(all_error_LMS2**2, label="LMS")

    plt.xlabel("Sample Number")
    plt.ylabel("Power of error")
    plt.title("Error between response signal and desired response
signal")
    plt.legend()

    plt.figure(2)
```

```
    plt.semilogy(all_learning_rate_LMS2, label="LMS")

    plt.xlabel("Sample Number")
    plt.ylabel("Error Rate")
    plt.title("Normalised weight error vector norm")
    plt.legend()

    plt.show()
```

2.1

```
# -*- coding: utf-8 -*-
"""

"""

import numpy as np
import math
import matplotlib.pyplot as plt
import scipy.io

"_____________________________________________________Sub
Functions_____________________________________________________"


def compute_error(w_k, w_o):  # Normalised weight error vector norm
    numerator = np.sum((w_k - w_o) ** 2)
    denominator = np.sum(w_o ** 2)
    return numerator / denominator


# LMS algorithm
def LMS(x_in, d_in, m, L):  # x_in is the adaptive filter input, d_in
is the desired response signal
    S = len(x_in)  # m is the adaptation gain, L is the length of the
adaptive filter
    e = np.zeros(S)  # error array
    w = [[0] * L for _ in range(S + 1)]  # weights, should be N*L
matrix, the additional row is for the initial w
    zs = np.zeros(L - 1)
    x_add0 = np.concatenate((zs, x_in))  # add L-1 zeros to the
beginning of input, start with [0 0 0 0 0 0 0 0 x1]

    for k in range(S):
        x_k = x_add0[k - len(x_add0) - 1 + L:k - len(x_add0) - 1:-1]
# take k to k+L-1 x_add0, and reverse their order
        e[k] = d_in[k] - np.dot(w[k], x_k)  # calculate the error
between response signal and desired response signal
        w[k + 1] = np.add(w[k], 2 * m * x_k * e[k])  # update the
weight

    return e, w[1:S + 1]  # return N*L weights matrix


def NLMS(x_in, d_in, m, L):  # x_in is the adaptive filter input,
d_in is the desired response signal
    S = len(x_in)  # m is the adaptation gain, L is the length of the
adaptive filter
    e = np.zeros(S)  # error array
```

```
    w = [[0] * L for _ in range(S + 1)]  # weights, should be N*L
matrix, the additional row is for the initial w
    zs = np.zeros(L - 1)
    x_add0 = np.concatenate((zs, x_in))  # add L-1 zeros to the
beginning of input, start with [0 0 0 0 0 0 0 0 x1]

    for k in range(S):
        x_k = x_add0[k - len(x_add0) - 1 + L:k - len(x_add0) - 1:-1]
# take k to k+L-1 x_add0, and reverse their order

        temp = np.linalg.norm(x_k)
        if temp != 0:
            x_normalized = x_k / temp ** 2
        else:
            x_normalized = x_k

        e[k] = d_in[k] - np.dot(w[k], x_k)  # calculate the error
between desired response signal and response signal
        w[k + 1] = np.add(w[k], 2 * m * x_normalized * e[k])  #
update the weight

    return e, w[1:S + 1]  # return N*L weights matrix


"_____________________________________________________Main
function_____________________________________________________"


def main(x):
    size = len(x)
    L_f = 9
    L_f_un = 9

    P_x = np.sum(x ** 2) / size

    mu = 1 / (0.5592 * L_f)  # adaptation gain, input signal has a
unit power

    # This is the theoretical optimized w, use it to generate desired
response signal
    opt_w = np.zeros(L_f_un)
    for i in range(L_f_un):
        opt_w[i] = 1 / (i + 1) * math.exp(-((i - 4) ** 2) / 4)
    dn = np.convolve(x, opt_w, mode="full")  # desired response
signal

    # Calculate
    e_LMS, w_LMS = LMS(x, dn, mu, L_f)
    e_NLMS, w_NLMS = NLMS(x, dn, mu, L_f)

    # Compute the normalised weight error vector norm
    lr_LMS = np.zeros(size)
    lr_NLMS = np.zeros(size)
    for i in range(size):
        lr_LMS[i] = compute_error(w_LMS[i], opt_w)
        lr_NLMS[i] = compute_error(w_NLMS[i], opt_w)

    return e_LMS, lr_LMS, e_NLMS, lr_NLMS  # return N*L weights
matrix
```

```
"_____________________________________________________Plot
Diagrams_____________________________________________________"

if __name__ == "__main__":
    s1 = scipy.io.loadmat('s1.mat')
    s1 = s1['s1']
    s1 = np.array(s1).flatten()

    all_error_LMS, all_learning_rate_LMS, all_error_NLMS,
all_learning_rate_NLMS = main(s1)

    plt.figure(1)
    plt.semilogy(all_error_LMS**2, label="LMS")
    plt.semilogy(all_error_NLMS**2, label="NLMS")
    plt.xlabel("Sample Number")
    plt.ylabel("Power of error")
    plt.title("Error between response signal and desired response
signal")
    plt.legend()

    plt.figure(2)
    plt.semilogy(all_learning_rate_LMS, label="LMS")
    plt.semilogy(all_learning_rate_NLMS, label="NLMS")
    plt.xlabel("Sample Number")
    plt.ylabel("Error Rate")
    plt.title("Normalised weight error vector norm")
    plt.legend()

    plt.show()
```

3.1

```
# -*- coding: utf-8 -*-
"""

"""

import numpy as np
import math
import matplotlib.pyplot as plt

"_____________________________________________________Sub
Functions_____________________________________________________"


def compute_error(x_in, d_in, w):
    return d_in - np.dot(w, x_in)


def compute_sse(w_k, w_o):  # Normalised weight error vector norm
    numerator = np.sum((w_k - w_o) ** 2)
    denominator = np.sum(w_o ** 2)
    return numerator / denominator


def compute_kalman_gain(x_in, last_rinv, ff):
    x_in = x_in[:, np.newaxis]
    a = np.matmul(last_rinv, x_in)
    return a / (ff + np.matmul(x_in.T, a))
```

```
def compute_rinv(x_in, last_rinv, kalman_gain, ff):
    x_in = x_in[:, np.newaxis]
    a = np.matmul(x_in.T, last_rinv)
    b = np.matmul(kalman_gain, a)
    return (1 / ff) * (last_rinv - b)


def RLS(x_in, d_in, lamb, L):
    S = len(x_in)
    e = np.zeros(S)  # error array
    w = [[0] * L for _ in range(S + 1)]  # weights, should be N*L
matrix, the additional row is for the initial w
    zs = np.zeros(L - 1)
    x_add0 = np.concatenate((zs, x_in))
    rinv = np.eye(L)
    for k in range(S):
        x_k = x_add0[k - len(x_add0) - 1 + L:k - len(x_add0) - 1:-1]
# take k to k+L-1 x_add0, and reverse their order
        e[k] = compute_error(x_k, d_in[k], w[k])
        kalman_gain = compute_kalman_gain(x_k, rinv, lamb)
        w[k+1] = w[k] + kalman_gain.flatten() * e[k]
        rinv = compute_rinv(x_k, rinv, kalman_gain, lamb)
    return e, w[1:S + 1]


"_____________________________________________________Main
function_____________________________________________________"


if __name__ == "__main__":
    L_f = 9
    L_f_un = 9
    forget_factor = 1
    Size = 200

    Loop = 1
    all_error_equal = np.zeros([Loop, Size])
    all_misad_equal = np.zeros([Loop, Size])
    all_error_low = np.zeros([Loop, Size])
    all_misad_low = np.zeros([Loop, Size])
    all_error_high = np.zeros([Loop, Size])
    all_misad_high = np.zeros([Loop, Size])

    for q in range(Loop):
        x = np.random.randn(Size)  # generate the input signal with
unit power and zero mean

        # This is the theoretical optimized w, use it to generate
desired response signal
        opt_w = np.zeros(L_f_un)
        for i in range(L_f_un):
            opt_w[i] = 1 / (i + 1) * math.exp(-((i - 4) ** 2) / 4)
        dn = np.convolve(x, opt_w, mode="full")  # desired response
signal

        L_f_equal = 9  # length of the adaptive filter
        all_error_equal[q, :], w_RLS_equal = RLS(x, dn,
forget_factor, L_f_equal)
```

```
        L_f_low = 5  # length of the adaptive filter
        all_error_low[q, :], w_RLS_low = RLS(x, dn, forget_factor,
L_f_low)

        L_f_high = 11  # length of the adaptive filter
        all_error_high[q, :], w_RLS_high = RLS(x, dn, forget_factor,
L_f_high)

        # Compute the misadjustment (normalised weight error vector
norm)
        opt_w_low = np.zeros(L_f_low)
        for i in range(L_f_low):
            opt_w_low[i] = 1 / (i + 1) * math.exp(-((i - 4) ** 2) /
4)
        dn = np.convolve(x, opt_w_low, mode="full")  # desired
response signal

        opt_w_high = np.zeros(L_f_high)
        for i in range(L_f_high):
            opt_w_high[i] = 1 / (i + 1) * math.exp(-((i - 4) ** 2) /
4)
        dn = np.convolve(x, opt_w_high, mode="full")  # desired
response signal

        for i in range(Size):
            all_misad_equal[q, i] = compute_sse(w_RLS_equal[i],
opt_w)
            all_misad_low[q, i] = compute_sse(w_RLS_low[i],
opt_w_low)
            all_misad_high[q, i] = compute_sse(w_RLS_high[i],
opt_w_high)

    mean_error_equal = np.mean(all_error_equal**2, 0)
    mean_error_low = np.mean(all_error_low**2, 0)
    mean_error_high = np.mean(all_error_high**2, 0)

    mean_misad_equal = np.mean(all_misad_equal, 0)
    mean_misad_low = np.mean(all_misad_low, 0)
    mean_misad_high = np.mean(all_misad_high, 0)

    plt.figure(1)
    plt.semilogy(mean_error_equal, label="L_f = 9")
    plt.semilogy(mean_error_low, label="L_f = 5")
    plt.semilogy(mean_error_high, label="L_f = 11")
    plt.xlabel("Sample Number")
    plt.ylabel("Power of error")
    plt.title("Error between response signal and desired response
signal")
    plt.legend()

    plt.figure(2)
    plt.semilogy(mean_misad_equal, label="L_f = 9")
    plt.semilogy(mean_misad_low, label="L_f = 5")
    plt.semilogy(mean_misad_high, label="L_f = 11")
    plt.xlabel("Sample Number")
    plt.ylabel("Error Rate")
    plt.title("Normalised weight error vector norm")
    plt.legend()

    plt.show()
```

3.2

```
# -*- coding: utf-8 -*-
"""

"""

import numpy as np
import math
import matplotlib.pyplot as plt

"_____________________________________________________Sub
Functions_____________________________________________________"


def compute_error(x_in, d_in, w):
    return d_in - np.dot(w, x_in)


def compute_sse(w_k, w_o):  # Normalised weight error vector norm
    numerator = np.sum((w_k - w_o) ** 2)
    denominator = np.sum(w_o ** 2)
    return numerator / denominator


def compute_kalman_gain(x_in, last_rinv, ff):
    x_in = x_in[:, np.newaxis]
    a = np.matmul(last_rinv, x_in)
    return a / (ff + np.matmul(x_in.T, a))


def compute_rinv(x_in, last_rinv, kalman_gain, ff):
    x_in = x_in[:, np.newaxis]
    a = np.matmul(x_in.T, last_rinv)
    b = np.matmul(kalman_gain, a)
    return (1 / ff) * (last_rinv - b)


def RLS(x_in, d_in, lamb, L):
    S = len(x_in)
    e = np.zeros(S)  # error array
    w = [[0] * L for _ in range(S + 1)]  # weights, should be N*L
matrix, the additional row is for the initial w
    zs = np.zeros(L - 1)
    x_add0 = np.concatenate((zs, x_in))
    rinv = np.eye(L)
    for k in range(S):
        x_k = x_add0[k - len(x_add0) - 1 + L:k - len(x_add0) - 1:-1]
# take k to k+L-1 x_add0, and reverse their order
        e[k] = compute_error(x_k, d_in[k], w[k])
        kalman_gain = compute_kalman_gain(x_k, rinv, lamb)
        w[k+1] = w[k] + kalman_gain.flatten() * e[k]
        rinv = compute_rinv(x_k, rinv, kalman_gain, lamb)
    return e, w[1:S + 1]


"_____________________________________________________Main
function_____________________________________________________"
```

```
if __name__ == "__main__":
    L_f = 9
    L_f_un = 9
    forget_factor = 0.8
    Size = 200

    Loop = 200
    all_error_equal = np.zeros([Loop, Size])
    all_misad_equal = np.zeros([Loop, Size])
    all_error_low = np.zeros([Loop, Size])
    all_misad_low = np.zeros([Loop, Size])
    all_error_high = np.zeros([Loop, Size])
    all_misad_high = np.zeros([Loop, Size])

    for q in range(Loop):
        x = np.random.randn(Size)  # generate the input signal with
unit power and zero mean

        # This is the theoretical optimized w, use it to generate
desired response signal
        opt_w = np.zeros(L_f_un)
        for i in range(L_f_un):
            opt_w[i] = 1 / (i + 1) * math.exp(-((i - 4) ** 2) / 4)
        dn = np.convolve(x, opt_w, mode="full")  # desired response
signal

        L_f_equal = 9  # length of the adaptive filter
        all_error_equal[q, :], w_RLS_equal = RLS(x, dn,
forget_factor, L_f_equal)

        L_f_low = 5  # length of the adaptive filter
        all_error_low[q, :], w_RLS_low = RLS(x, dn, forget_factor,
L_f_low)

        L_f_high = 11  # length of the adaptive filter
        all_error_high[q, :], w_RLS_high = RLS(x, dn, forget_factor,
L_f_high)

        # Compute the misadjustment (normalised weight error vector
norm)
        opt_w_low = np.zeros(L_f_low)
        for i in range(L_f_low):
            opt_w_low[i] = 1 / (i + 1) * math.exp(-((i - 4) ** 2) /
4)
        dn = np.convolve(x, opt_w_low, mode="full")  # desired
response signal

        opt_w_high = np.zeros(L_f_high)
        for i in range(L_f_high):
            opt_w_high[i] = 1 / (i + 1) * math.exp(-((i - 4) ** 2) /
4)
        dn = np.convolve(x, opt_w_high, mode="full")  # desired
response signal

        for i in range(Size):
            all_misad_equal[q, i] = compute_sse(w_RLS_equal[i],
opt_w)
            all_misad_low[q, i] = compute_sse(w_RLS_low[i],
opt_w_low)
```

```
            all_misad_high[q, i] = compute_sse(w_RLS_high[i],
opt_w_high)

    mean_error_equal = np.mean(all_error_equal**2, 0)
    mean_error_low = np.mean(all_error_low**2, 0)
    mean_error_high = np.mean(all_error_high**2, 0)

    mean_misad_equal = np.mean(all_misad_equal, 0)
    mean_misad_low = np.mean(all_misad_low, 0)
    mean_misad_high = np.mean(all_misad_high, 0)

    plt.figure(1)
    plt.semilogy(mean_error_equal, label="L_f = 9")
    plt.semilogy(mean_error_low, label="L_f = 5")
    plt.semilogy(mean_error_high, label="L_f = 11")
    plt.xlabel("Sample Number")
    plt.ylabel("Power of error")
    plt.title("Error between response signal and desired response
signal")
    plt.legend()

    plt.figure(2)
    plt.semilogy(mean_misad_equal, label="L_f = 9")
    plt.semilogy(mean_misad_low, label="L_f = 5")
    plt.semilogy(mean_misad_high, label="L_f = 11")
    plt.xlabel("Sample Number")
    plt.ylabel("Error Rate")
    plt.title("Normalised weight error vector norm")
    plt.legend()

    plt.show()
```

3.4

```
# -*- coding: utf-8 -*-
"""

"""

import numpy as np
import math
import matplotlib.pyplot as plt

"_____________________________________________________Sub
Functions_____________________________________________________"


def compute_error(x_in, d_in, w):
    return d_in - np.dot(w, x_in)


def compute_sse(w_k, w_o):  # Normalised weight error vector norm
    numerator = np.sum((w_k - w_o) ** 2)
    denominator = np.sum(w_o ** 2)
    return numerator / denominator
```

```
def compute_kalman_gain(x_in, last_rinv, ff):
    x_in = x_in[:, np.newaxis]
    a = np.matmul(last_rinv, x_in)
    return a / (ff + np.matmul(x_in.T, a))


def compute_rinv(x_in, last_rinv, kalman_gain, ff):
    x_in = x_in[:, np.newaxis]
    a = np.matmul(x_in.T, last_rinv)
    b = np.matmul(kalman_gain, a)
    return (1 / ff) * (last_rinv - b)


def RLS(x_in, d_in, lamb, L):
    S = len(x_in)
    e = np.zeros(S)  # error array
    w = [[0] * L for _ in range(S + 1)]  # weights, should be N*L
matrix, the additional row is for the initial w
    zs = np.zeros(L - 1)
    x_add0 = np.concatenate((zs, x_in))
    rinv = np.eye(L)
    for k in range(S):
        x_k = x_add0[k - len(x_add0) - 1 + L:k - len(x_add0) - 1:-1]
# take k to k+L-1 x_add0, and reverse their order
        e[k] = compute_error(x_k, d_in[k], w[k])
        kalman_gain = compute_kalman_gain(x_k, rinv, lamb)
        w[k+1] = w[k] + kalman_gain.flatten() * e[k]
        rinv = compute_rinv(x_k, rinv, kalman_gain, lamb)
    return e, w[1:S + 1]


"_____________________________________________________Main
function_____________________________________________________"


if __name__ == "__main__":
    L_f = 9
    L_f_un = 9
    forget_factor = 1
    Size = 200

    Loop = 200
    all_error = np.zeros([Loop, Size])
    all_misad = np.zeros([Loop, Size])
    all_error_noise = np.zeros([Loop, Size])
    all_misad_noise = np.zeros([Loop, Size])

    for q in range(Loop):
        x = np.random.randn(Size)  # generate the input signal with
unit power and zero mean
        noise = 1 / 10 * np.random.randn(Size + L_f_un - 1)

        # This is the theoretical optimized w, use it to generate
desired response signal
        opt_w = np.zeros(L_f_un)
        for i in range(L_f_un):
            opt_w[i] = 1 / (i + 1) * math.exp(-((i - 4) ** 2) / 4)
        dn = np.convolve(x, opt_w, mode="full")  # desired response
signal
        dn_noise = np.add(np.convolve(x, opt_w, mode="full"), noise)
# desired response signal
```

```
        all_error[q, :], w_RLS = RLS(x, dn, forget_factor, L_f)
        all_error_noise[q, :], w_RLS_noise = RLS(x, dn_noise,
forget_factor, L_f)

        # Compute the misadjustment (normalised weight error vector
norm)
        for i in range(Size):
            all_misad[q, i] = compute_sse(w_RLS[i], opt_w)
            all_misad_noise[q, i] = compute_sse(w_RLS_noise[i],
opt_w)

    mean_error = np.mean(all_error**2, 0)
    mean_error_noise = np.mean(all_error_noise**2, 0)
    mean_misad = np.mean(all_misad, 0)
    mean_misad_noise = np.mean(all_misad_noise, 0)

    plt.figure(1)
    plt.semilogy(mean_error, label="Without noise")
    plt.semilogy(mean_error_noise, label="With noise")
    plt.xlabel("Sample Number")
    plt.ylabel("Power of error")
    plt.title("Error between response signal and desired response
signal")
    plt.legend()

    plt.figure(2)
    plt.semilogy(mean_misad, label="Without noise")
    plt.semilogy(mean_misad_noise, label="With noise")
    plt.xlabel("Sample Number")
    plt.ylabel("Error Rate")
    plt.title("Normalised weight error vector norm")
    plt.legend()

    plt.show()
```