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

```
# -*- 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()
```