Bachelor of Technology (B. Tech.)

Computer and Communication Engineering (CCE)

Amrita School of Engineering

Coimbatore Campus (India)

Academic Year – 2022 - 23

S. No	Name	Roll Number
1	B Ambareesh	CB.EN.U4CCE20006
2	Manoj Parthiban	CB.EN.U4CCE20032
3	V Parthiv	CB.EN.U4CCE20041
4	Santosh	CB.EN.U4CCE20053

Semester VI – Third Year

19CCE311 Wireless Communication

Assignment – LMS & RLS Algorithms

(This assignment aims to implement and compare the performance of two adaptive filtering algorithms, namely the Least Mean Square (LMS) and Recursive Least Squares (RLS) algorithms, for the task of noise cancellation. The LMS algorithm uses an adaptive filter to estimate and subtract the noise from a given input signal, while the RLS algorithm uses a recursive estimation approach to achieve the same objective.)

Faculty In-charge – Ms Prabha G.

LEAST MEAN SQUARE (LMS) FILTER

1. Introduction:

The Least Mean Square (LMS) filter is a widely used adaptive filter algorithm in the field of signal processing and machine learning. It is a type of linear filter that adjusts its coefficients iteratively to minimize the mean square error (MSE) between the desired output and the filter's output. The LMS filter is particularly useful for applications such as noise cancellation, echo cancellation, and equalization.

The main advantage of the LMS filter is its simplicity and computational efficiency. It is relatively easy to implement and does not require complex calculations or large amounts of memory.

However, it has some limitations. The convergence speed of the LMS algorithm depends on the choice of the adaptation parameter μ, and selecting an inappropriate value can result in slow convergence or instability. Additionally, the LMS algorithm is sensitive to the statistical properties of the input signal, and its performance may degrade in non-stationary or highly correlated environments.

Despite its limitations, the LMS filter remains a popular choice for various applications due to its simplicity, adaptability, and real-time processing capabilities. It has been widely used in areas such as communication systems, audio and speech processing, adaptive beamforming, and active noise control.

2. Algorithm:

The LMS algorithm is based on the concept of stochastic gradient descent, where the filter coefficients are updated incrementally in the direction of the negative gradient of the MSE. The update equation for the LMS filter can be expressed as:

$$w(n+1) = w(n) + \mu * e(n) * x(n)$$

In the above equation, $w(n)$ represents the filter coefficients at iteration n, μ is the step size or adaptation parameter that controls the convergence speed, $e(n)$ is the error signal (the difference between the desired output and the filter output), and $x(n)$ is the input signal or observation.

The LMS algorithm starts with an initial set of filter coefficients and then iteratively adjusts them based on the current input and error signal. By continuously updating the coefficients, the LMS filter adapts to the changing characteristics of the input signal and tries to minimize the error between the desired and actual outputs.

3. Pseudo Code:

i. Initialize the filter coefficients: Start by assigning initial values to the filter coefficients, denoted as $w(0)$.

ii. Receive an input signal: Obtain the next input signal or observation, denoted as $x(n)$.

iii. Compute the filter output: Multiply the filter coefficients with the input signal and calculate the filter output, denoted as y(n).

iv. Calculate the error: Determine the error signal e(n) by subtracting the desired output or target signal from the filter output, i.e., e(n) = d(n) - y(n), where d(n) is the desired output.

v. Update the filter coefficients: Adjust the filter coefficients using the LMS update equation:

$$w(n+1) = w(n) + \mu * e(n) * x(n)$$

Here, μ is the step size or adaptation parameter that controls the convergence speed. The smaller the value of μ, the slower the convergence.

vi. Repeat steps 2 to 5: Iterate through steps 2 to 5, processing each input signal and updating the filter coefficients.

vii. Repeat until convergence: Continue the iterations until the filter converges, i.e., the error signal becomes sufficiently small or reaches a predefined threshold.

The LMS algorithm adapts the filter coefficients iteratively, gradually minimizing the mean square error between the desired output and the filter output. By adjusting the coefficients based on the gradient of the error signal, the LMS filter can adapt to changing signal characteristics and optimize its performance over time.

4. Python Code:

```python
class LMSFilter:

    def __init__(self, num_taps, step_size):
        self.num_taps = num_taps
        self.step_size = step_size
        self.weights = np.random.rand(num_taps)

    def update(self, signal, target):
        prediction = np.dot(self.weights.T, signal)
        error = target - prediction
        self.weights += self.step_size * error * signal
        return prediction, error
```

This code defines a Python class named `LMSFilter` which implements an adaptive filter using the Least Mean Squares (LMS) algorithm. The `__init__` function initializes the properties of the filter object such as the number of weights or taps, and the step size to update these weights. In addition, it creates an array of random weights.

The `update` method takes two inputs, the input signal and the desired output signal (or target). It computes the predicted output by multiplying the input signal with the current set of weights. It then computes the error between the predicted output and the desired output.

Finally, it updates the weights to minimize the error between the predicted output and the desired output using the LMS algorithm, by scaling the error by the input signal and learning rate (step_size) and adding to the weights.

The method returns the prediction and error values. This class can be used in various applications such as signal processing, noise cancellation, and system identification.

RECURSIVE LEAST SQUARES (RLS) FILTER

1. Introduction:

The Recursive Least Squares (RLS) filter is an adaptive filter algorithm used in the field of signal processing and machine learning. It is a type of linear filter that estimates the coefficients iteratively by minimizing the weighted sum of squared errors. The RLS filter is particularly useful for applications such as system identification, channel equalization, and adaptive noise cancellation.

The RLS filter has several advantages, including fast convergence, high accuracy, and good tracking capabilities. It is suitable for applications where accurate estimation of the filter coefficients is crucial. However, the RLS algorithm is computationally more complex and requires more memory compared to the LMS algorithm. It may also be sensitive to numerical stability issues, especially when dealing with ill-conditioned matrices.

Despite these considerations, the RLS filter is widely used in various applications, such as adaptive equalization in communication systems, adaptive noise cancellation in audio processing, and system identification in control systems. Its ability to provide accurate estimation and adaptability make it a valuable tool in signal processing and adaptive filtering.

2. Algorithm:

The RLS algorithm is based on the method of recursive estimation and is known for its ability to provide fast convergence and accurate parameter estimation. Unlike the LMS algorithm, which updates the filter coefficients incrementally, the RLS algorithm computes the filter coefficients recursively using a matrix inversion technique.

The RLS algorithm maintains two main matrices: the inverse of the autocorrelation matrix and the cross-correlation matrix. These matrices store the statistical properties of the input signal and the desired output. At each iteration, the RLS algorithm updates these matrices based on the current input and output signals. The filter coefficients are then calculated using the updated matrices.

The update equation for the RLS filter can be expressed as:

$$w(n) = w(n-1) + K(n) * e(n)$$

$$P(n) = (1/\lambda) * [P(n-1) - K(n) * P(n-1) * x(n)]$$

Here, w(n) represents the filter coefficients at iteration n, e(n) is the error signal (the difference between the desired output and the filter output), P(n) is the inverse of the autocorrelation matrix, K(n) is the Kalman gain, λ is the forgetting factor that controls the influence of past data, and x(n) is the input signal or observation.

The RLS algorithm updates the filter coefficients and the inverse of the autocorrelation matrix recursively, allowing for efficient and real-time adaptation. The use of matrix inversion in the RLS algorithm provides accurate parameter estimation, especially in situations with correlated or non-stationary input signals.

3. Pseudo Code:

i. Initialize the filter coefficients and matrices: Start by assigning initial values to the filter coefficients, denoted as w(0). Initialize the inverse of the autocorrelation matrix, denoted as P(0).

ii. Receive an input signal: Obtain the next input signal or observation, denoted as x(n).

iii. Compute the filter output: Multiply the filter coefficients with the input signal and calculate the filter output, denoted as y(n).

iv. Calculate the error: Determine the error signal e(n) by subtracting the desired output or target signal from the filter output, i.e., e(n) = d(n) - y(n), where d(n) is the desired output.

v. Update the matrices: Update the inverse of the autocorrelation matrix P(n) and the cross-correlation matrix R(n) using recursive formulas.

vi. Calculate the Kalman gain: Compute the Kalman gain K(n) using the updated matrices.

vii. Update the filter coefficients: Adjust the filter coefficients using the RLS update equation:

$$w(n) = w(n-1) + K(n) * e(n)$$

viii. Repeat steps 2 to 7: Iterate through steps 2 to 7, processing each input signal and updating the filter coefficients and matrices recursively.

ix. Repeat until convergence: Continue the iterations until the filter converges, i.e., the error signal becomes sufficiently small or reaches a predefined threshold.

The RLS algorithm estimates the filter coefficients by recursively updating the matrices and calculating the Kalman gain. By using matrix inversion techniques, it provides accurate parameter estimation and fast convergence compared to other adaptive filter algorithms.

4. Python Code:

```
class RLSFilter:

    def __init__(self, num_taps, forgetting_factor):
        self.num_taps = num_taps
        self.forgetting_factor = forgetting_factor
        self.weights = np.random.rand(num_taps)
        self.P = 1e3 * np.eye(num_taps)

    def update(self, signal, target):
        prediction = np.dot(self.weights.T, signal)
        error = target - prediction
        K = np.dot(self.P, signal) / (self.forgetting_factor +
np.dot(signal.T, np.dot(self.P, signal)))
        self.weights += K * error
        self.P = (self.P - np.outer(K, np.dot(signal.T, self.P))) /
self.forgetting_factor
        return prediction, error
```

This code defines a Python class named `RLSFilter` that implements an adaptive filter using the Recursive Least Squares (RLS) algorithm. The `__init__` function initializes the properties of the filter object such as the number of weights or taps, and the forgetting factor which controls the trade-off between tracking speed and steady-state performance. Additionally, it creates an array of random weights and initializes a matrix P with dimensions num_taps x num_taps, which is used to update the weights.

The `update` method takes two inputs, the input signal and the desired output signal (or target). It computes the predicted output by multiplying the input signal with the current set of weights. It then computes the error between the predicted output and the desired output.

The RLS algorithm updates the weight vector based on K, which is the Kalman gain matrix. The Kalman gain matrix is computed using the inverse of the correlation matrix P, which is also updated after every iteration. The weights are updated by adding the product of the Kalman gain and the error vector.

Finally, the method returns the prediction and error values. This class can be used in various applications such as signal processing, noise cancellation, and system identification.

APPLICATION & PERFORMANCE ANALYSIS

This is a Python code that applies LMS and RLS filters to an audio signal with the aim of denoising it. The steps in this code are as follows:

1. **Import necessary libraries for reading audio files, working with arrays, and plotting graphs: scipy.io.wavfile, numpy, and matplotlib.pyplot.**

```
import scipy.io.wavfile as wav
import numpy as np
import matplotlib.pyplot as plt
```

2. **Define constants for the maximum amplitude of the input signal (MAX_AMPLITUDE), number of filter taps (NUM_TAPS), step size of the LMS filter (STEP_SIZE), and forgetting factor of the RLS filter (FORGETTING_FACTOR).**

```
NOISY_SIGNAL_FILE = 'noisy_speech.wav'
CLEAN_SIGNAL_FILE = 'clean_speech.wav'

MAX_AMPLITUDE = 32768.0
NUM_TAPS = 256
STEP_SIZE = 0.01
FORGETTING_FACTOR = 0.99
```

3. **Define two classes: LMSFilter and RLSFilter.**

LMSFilter has a constructor that takes in the number of taps and step size of the filter. It initializes weights randomly and is an update method that takes in a signal and its target computes a prediction and error, and updates the weights. This method returns the prediction and error.

```
class LMSFilter:
    def __init__(self, num_taps, step_size):
        self.num_taps = num_taps
        self.step_size = step_size
        self.weights = np.random.rand(num_taps)

    def update(self, signal, target):
        prediction = np.dot(self.weights.T, signal)
        error = target - prediction
        self.weights += self.step_size * error * signal
        return prediction, error
```

RLSFilter has a constructor too that takes in the number of taps and forgetting factor, initializes weights and P randomly, and an update method that first predicts the output, computes the error, and then calculates the K matrix which is multiplied by the error to update the weights. Finally, the method updates the P matrix and returns the prediction and error.

```
class RLSFilter:
    def __init__(self, num_taps, forgetting_factor):
        self.num_taps = num_taps
        self.forgetting_factor = forgetting_factor
        self.weights = np.random.rand(num_taps)
        self.P = 1e3 * np.eye(num_taps)

    def update(self, signal, target):
        prediction = np.dot(self.weights.T, signal)
        error = target - prediction
        K = np.dot(self.P, signal) / (self.forgetting_factor +
np.dot(signal.T, np.dot(self.P, signal)))
        self.weights += K * error
        self.P = (self.P - np.outer(K, np.dot(signal.T, self.P))) /
self.forgetting_factor
        return prediction, error
```

4. **Define functions for reading the noisy and clean audio file, getting the magnitude spectrum of an input signal, getting the frequency and spectrum of an input signal, plotting the signal and its frequency spectrum, plotting the filter response and the filtered signal, plotting histogram representation of an input signal, padding an audio signal with zeros to match a target length.**

```
def read_audio_file(file_name):
    """Reads an audio file and returns the sampling rate and signal
data."""
    with open(file_name, 'rb') as file:
        rate, signal = wav.read(file)
    signal = signal.astype(np.float32) / MAX_AMPLITUDE
    return rate, signal

def get_magnitude_spectrum(signal, rate):
    """Returns the magnitude spectrum of a signal"""
    freqs, spectrum = signal_spectrum(signal, rate)
    magnitudes = np.abs(spectrum)
    return freqs, magnitudes
```

5. **The `signal_spectrum` function takes in an audio signal and its sample rate as input computes the frequency spectrum of the signal using the Fast Fourier Transform (FFT) and returns the frequency range and its corresponding spectrum. The function pads the input signal with zeros to make its length equal to a power of 2 before applying FFT.**

```
def signal_spectrum(signal, rate):
    """Returns the frequency and spectrum of a signal"""
    n = len(signal)
    k = np.arange(n)
    t = n/rate
    freqs = k/t # two sides frequency range
    freqs = freqs[:n//2] # one side frequency range
    sp = np.fft.fft(signal)/n # fft computing and normalization
    sp = sp[:n//2]
    return freqs, sp
```

6. **The `plot_signal_and_spectrum` function takes in an audio signal and its title as input, calls the `signal_spectrum` function to compute its frequency spectrum, and plots both the original signal and its magnitude spectrum on separate subplots.**

```
def plot_signal_and_spectrum(signal, rate, title):
    """Plots the signal and its frequency spectrum"""

    fig, axs = plt.subplots(2, 1, figsize=(12, 8))
    axs[0].plot(signal)
    axs[0].set_xlabel('Time (s)')
    axs[0].set_ylabel('Amplitude')
    axs[0].set_title(title + ' Audio Signal')

    freqs, magnitudes = get_magnitude_spectrum(signal, rate)
    axs[1].plot(freqs, magnitudes)
    axs[1].set_xlabel('Frequency (Hz)')
    axs[1].set_ylabel('Magnitude')
    axs[1].set_title(title + ' Magnitude Spectrum')

    plt.tight_layout()
    plt.show()
```

7. **`plot_filter_responses` function takes in a filtered signal, original signal, sample rate and title as input, computes the magnitude spectrum of both signals using the `get_magnitude_spectrum` function (not shown here), and plots the comparison of the two spectra as well as the comparison of the two time-domain signals on separate subplots.**

```python
def plot_filter_responses(filtered_signal, signal, rate, title):
    """Plots the filter response and the filtered signal"""

    fig, axs = plt.subplots(2, 1, figsize=(12, 8))
    axs[0].plot(signal, label='Original Signal', alpha=0.5)
    axs[0].plot(filtered_signal, label='Filtered Signal')
    axs[0].set_xlabel('Time (s)')
    axs[0].set_ylabel('Amplitude')
    axs[0].set_title(title + ' Filtered Audio Signal')
    axs[0].legend()

    freqs, magnitudes = get_magnitude_spectrum(signal, rate)
    axs[1].plot(freqs, magnitudes, label='Original Spectrum', alpha=0.5)

    freqs, magnitudes = get_magnitude_spectrum(filtered_signal, rate)
    axs[1].plot(freqs, magnitudes, label='Filtered Spectrum')
    axs[1].set_xlabel('Frequency (Hz)')
    axs[1].set_ylabel('Magnitude')
    axs[1].set_title(title + ' Magnitude Spectrum')
    axs[1].legend()

    plt.tight_layout()
    plt.show()
```

8. **The `plot_histogram` function takes in an audio signal and its title as input creates a histogram of its amplitudes and displays it.**

```python
def plot_histogram(signal, title):
    """Plots the histogram representation of a signal"""
    fig, ax = plt.subplots(figsize=(12, 8))
    ax.hist(signal, bins=100)
    ax.set_xlabel('Amplitude')
    ax.set_ylabel('Count')
    ax.set_title('Histogram of ' + title + ' Signal')
    plt.tight_layout()
    plt.show()
```

9. **The `pad_audio_signal` function takes in an audio signal and a target length as input, pads the signal with zeros if it's shorter than the target length and returns the resulting signal trimmed to the desired length.**

```
def pad_audio_signal(signal, target_len):
    """Pads an audio signal with zeros to match a target length."""
    if len(signal) < target_len:
        padding = target_len - len(signal)
        signal = np.pad(signal, (0, padding), mode='constant')
    return signal[:target_len]
```

10. **This is the `main` function of a program that applies two different adaptive filters, the Least Mean Square (LMS) filter and Recursive Least Squares (RLS) filter, to eliminate noise from an audio signal.**

The function first loads two audio signals, a noisy signal and a clean signal, from two separate audio files using the `read_audio_file` function (not shown here).

```
def main():
    # Load audio signals
    rate, noisy_signal = read_audio_file(NOISY_SIGNAL_FILE)
    rate, clean_signal = read_audio_file(CLEAN_SIGNAL_FILE)
```

Then, the signals are normalized by dividing each sample with `MAX_AMPLITUDE`, which is the maximum possible amplitude of the signal to represent the signal as values between -1 and 1.

```
    # Normalize signals to have maximum amplitude of 1
    noisy_signal = noisy_signal.astype(np.float32) / MAX_AMPLITUDE
    clean_signal = clean_signal.astype(np.float32) / MAX_AMPLITUDE
```

Next, the signals are padded with zeros to make their lengths equal to the length of the longer signal using the `pad_audio_signal` function.

```
    # Pad signals with zeros to match length
    noisy_signal = pad_audio_signal(noisy_signal, len(clean_signal))
    clean_signal = pad_audio_signal(clean_signal, len(noisy_signal))
```

Two adaptive filters, LMS and RLS filter, are initialized with their respective hyperparameters, such as the number of filter taps, learning rates and forgetting factor.

```
    # Initialize filters
    lms_filter = LMSFilter(NUM_TAPS, STEP_SIZE)
    rls_filter = RLSFilter(NUM_TAPS, FORGETTING_FACTOR)
```

Finally, the filters are applied to the noisy signal using a sliding window technique for each instant of time. In each iteration, a window of length `NUM_TAPS` is selected starting from the current sample, and the filter coefficients are updated using this window and the corresponding target value from the clean signal. The filtered output of both filters is computed and stored for each sample.

```
# Apply filters to noisy signal
lms_output = np.zeros_like(noisy_signal)
rls_output = np.zeros_like(noisy_signal)
for i in range(NUM_TAPS, len(noisy_signal)):
    signal_window = noisy_signal[i-NUM_TAPS:i]
    target = clean_signal[i]
    y_lms, e_lms = lms_filter.update(signal_window, target)
    y_rls, e_rls = rls_filter.update(signal_window, target)
    lms_output[i] = y_lms
    rls_output[i] = y_rls
```

11. **This code block is responsible for evaluating the performance of the two filters in removing noise from the original audio signal and visualizing the results.**

First, the `min_len` variable is used to ensure that both `lms_output` and `clean_signal` arrays have the same length. Any extra samples after the minimum length are discarded using a slicing operation.

```
# Ensure both arrays have the same length
min_len = min(len(lms_output), len(clean_signal))
lms_output = lms_output[:min_len]
clean_signal = clean_signal[:min_len]
```

Then, the mean squared error (MSE) between the filtered output and the clean signal is calculated for both the LMS filter and RLS filter using the formula `(filtered_output - clean_signal)^2`. The MSE values between the two filters and the clean signal are printed.

```
# Calculate the LMS and RLS errors
lms_error = np.mean((lms_output - clean_signal)**2)
rls_error = np.mean((rls_output - clean_signal)**2)
print('LMS error: %.2f' % lms_error)
print('RLS error: %.2f' % rls_error)
```

Next, several functions are called to plot various results:

- **`plot_signal_and_spectrum`: plots the time-domain and frequency-domain spectrum of the signals.**
- **`plot_filter_responses`: plots the impulse response and frequency response of the two filters.**
- **`plot_histogram`: plots histograms of the clean and noisy signals.**

```
    # Plot results
    plot_signal_and_spectrum(clean_signal, rate, 'Clean')
    plot_signal_and_spectrum(noisy_signal, rate, 'Noisy')

    plot_signal_and_spectrum(lms_output, rate, 'LMS')
    plot_signal_and_spectrum(rls_output, rate, 'RLS')

    plot_filter_responses(lms_output, noisy_signal, rate, 'LMS')
    plot_filter_responses(rls_output, noisy_signal, rate, 'RLS')

    plot_histogram(clean_signal, 'Clean')
    plot_histogram(noisy_signal, 'Noisy')
```

Finally, a comparison plot is created that shows the clean signal, the noisy signal, and the filtered outputs of both LMS and RLS filters. The plot title is "Comparison of Original and Filtered Signals" and it includes a legend indicating which line represents each signal.

```
    # Plot comparison of original and filtered signals
    plt.plot(clean_signal, label='Clean')
    plt.plot(noisy_signal, alpha=0.5, label='Noisy')
    plt.plot(lms_output, label='LMS')
    plt.plot(rls_output, label='RLS')
    plt.title('Comparison of Original and Filtered Signals')
    plt.legend()
    plt.show()
```

The visualization helps to evaluate the effectiveness of the filters in removing the noise from the original signal and to compare the performance of the two filters against each other.

12. If the Python file is executed, the main function is run.

```
if __name__ == '__main__':
    main()
```

PERFORMANCE ANALYSIS

After implementing and analyzing the LMS and RLS algorithms for noise cancellation, the following inferences can be drawn:

1. Noise cancellation: Both the LMS and RLS algorithms are effective in reducing or cancelling out noise from a given input signal. The algorithms adaptively adjust their filter coefficients to minimize the difference between the filtered output and the desired signal.

2. Convergence speed: The LMS algorithm typically has a faster convergence speed compared to the RLS algorithm. This is because the LMS algorithm updates the filter coefficients based on instantaneous error values, while the RLS algorithm uses a recursive estimation approach that considers past and present error values.

3. Computational complexity: The RLS algorithm generally has a higher computational complexity compared to the LMS algorithm. This is due to the matrix inversions and multiplications involved in the RLS algorithm's calculation of the filter coefficients.

4. Performance trade-off: The RLS algorithm provides better noise cancellation performance compared to the LMS algorithm in terms of mean square error (MSE) or signal-to-noise ratio (SNR). However, this performance improvement comes at the cost of increased computational complexity.

CONCLUSION

Overall, the choice between the LMS and RLS algorithms depends on the specific application requirements and the trade-off between performance and computational complexity. By analyzing the results and comparing the performance of the LMS and RLS algorithms, we can gain insights into their effectiveness and make informed decisions regarding their suitability for noise cancellation applications in different scenarios.