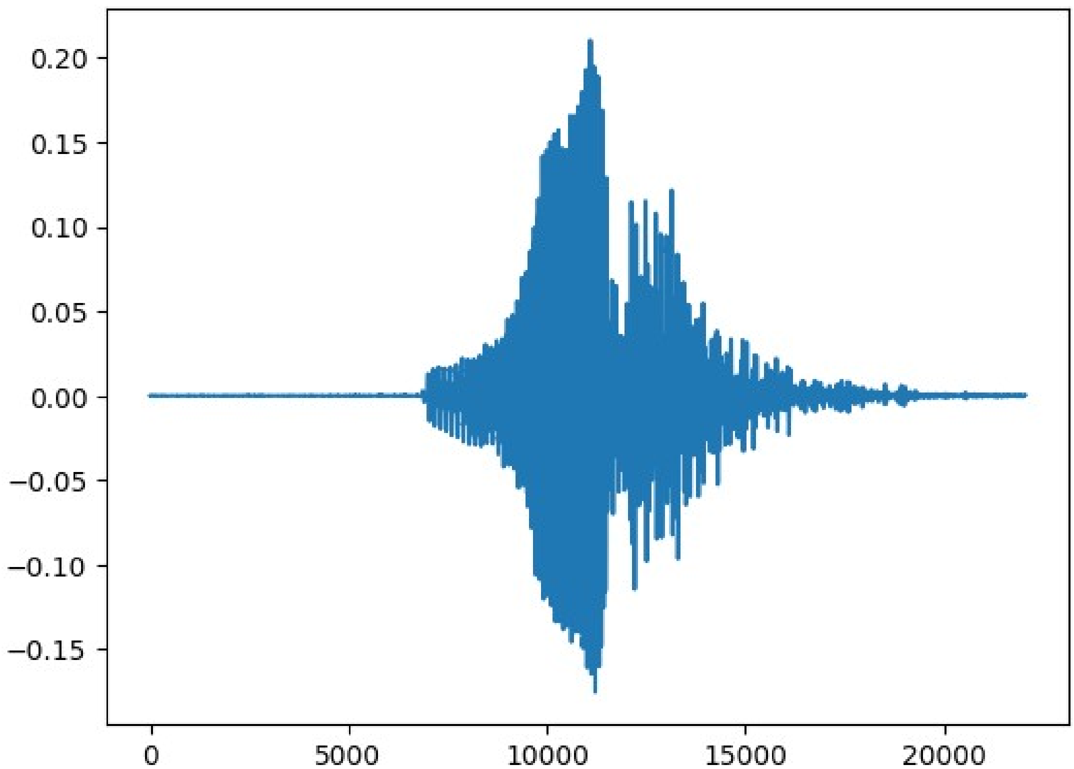
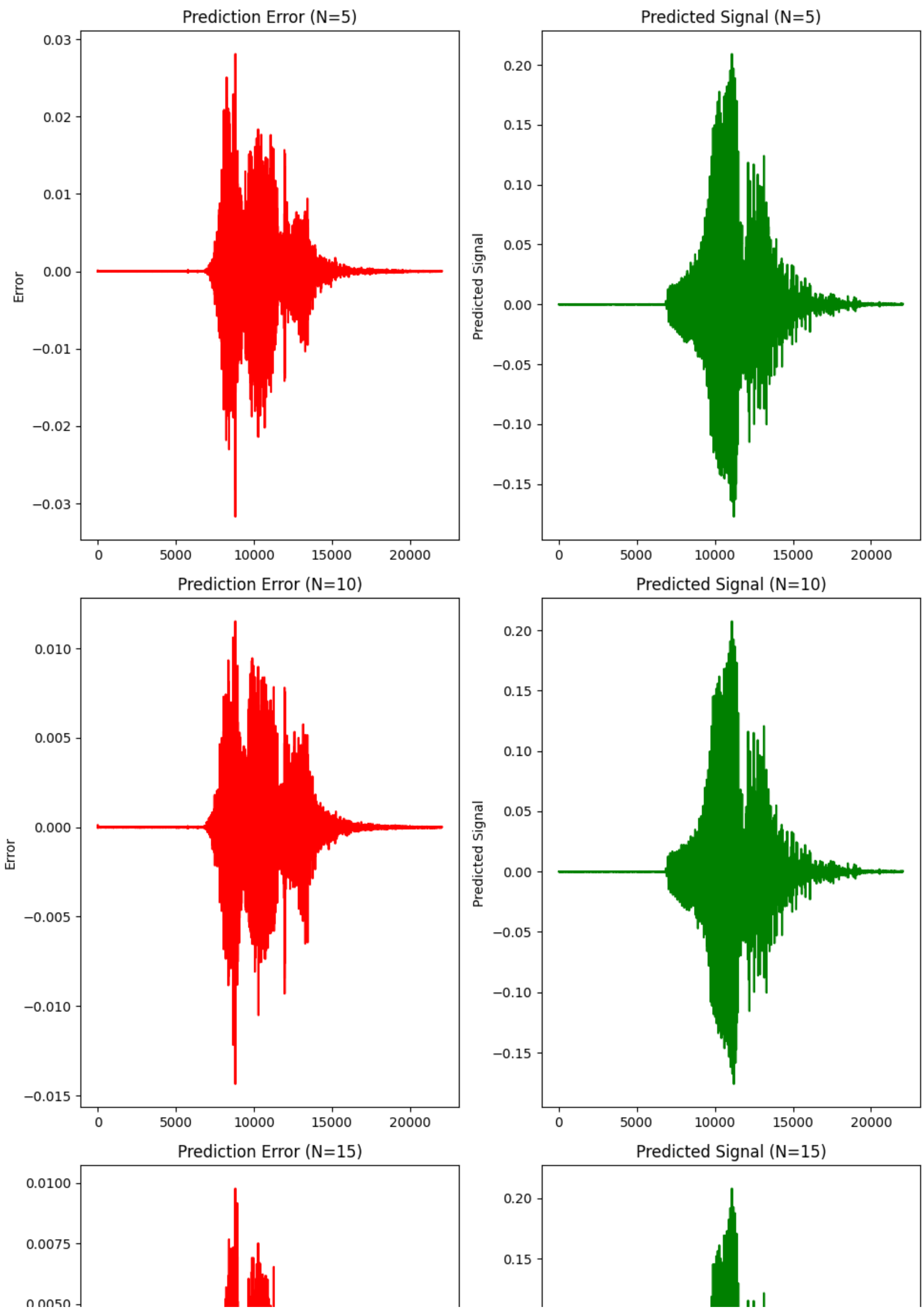
```python
import librosa
import numpy as np
import matplotlib.pyplot as plt
audio_path = 'Train_0_Example_1.wav'
audio_data, sample_rate = librosa.load(audio_path)
sliced =  audio_data[:22050]

t = np.linspace(0, len(sliced), len(sliced))
plt.plot(t, sliced)
```

```
[<matplotlib.lines.Line2D at 0x7b06485cac90>]
```




```python
import librosa
import numpy as np
import matplotlib.pyplot as plt

audio_path = 'Train_0_Example_1.wav'
audio_data, sample_rate = librosa.load(audio_path)
sliced = audio_data[:22050]

def autocorr(x, N):
    result = np.correlate(x, x, mode='same')
    result = result[len(result) // 2:]
    auto_vec = result[1:N + 1]
    auto_corr_matrix = np.zeros((N, N))
```

```python
    for i in range(N):
        for j in range(N):
            auto_corr_matrix[i, j] = result[abs(i - j)]

    return auto_corr_matrix, auto_vec

def autocorr_vec_func(x, N):
    result = np.correlate(x, x, mode='same')
    result = result[len(result) // 2:]
    auto_vec = result[1:N + 1]
    return auto_vec

def plot_error_and_prediction_for_N_values(N_values, sliced):
    plt.figure(figsize=(10, len(N_values) * 6))

    for i, N in enumerate(N_values):
        autocorr_mat, autocorr_vec = autocorr(sliced, N)
        autocorr_vec_r_1 = autocorr_vec_func(sliced, N)

        a = -np.linalg.inv(autocorr_mat) @ autocorr_vec_r_1

        x_predicted = np.zeros_like(sliced)

        for n in range(N, len(sliced)):
            x_predicted[n] = -np.sum(a * sliced[n - N:n][::-1])

        err = sliced - x_predicted

        plt.subplot(len(N_values), 2, 2 * i + 1)
        plt.plot(err, color='r')
        plt.title(f"Prediction Error (N={N})")
        plt.ylabel("Error")

        plt.subplot(len(N_values), 2, 2 * i + 2)
        plt.plot(x_predicted, color='g')
        plt.title(f"Predicted Signal (N={N})")
        plt.ylabel("Predicted Signal")

    plt.tight_layout()
    plt.show()

N_values = [5, 10, 15]
plot_error_and_prediction_for_N_values(N_values, sliced)
```


Prediction Error (N=5)
Error
0.03
0.02
0.01
0.00
−0.01
−0.02
−0.03
0
5000
10000
15000
20000
Predicted Signal (N=5)
Predicted Signal
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
Prediction Error (N=10)
Error
0.010
0.005
0.000
−0.005
−0.010
−0.015
Predicted Signal (N=10)
Predicted Signal
Prediction Error (N=15)
0.0100
0.0075
0.0050
Predicted Signal (N=15)
0.20
0.15

from the above plots we can say as the value of N increases the error decreases.

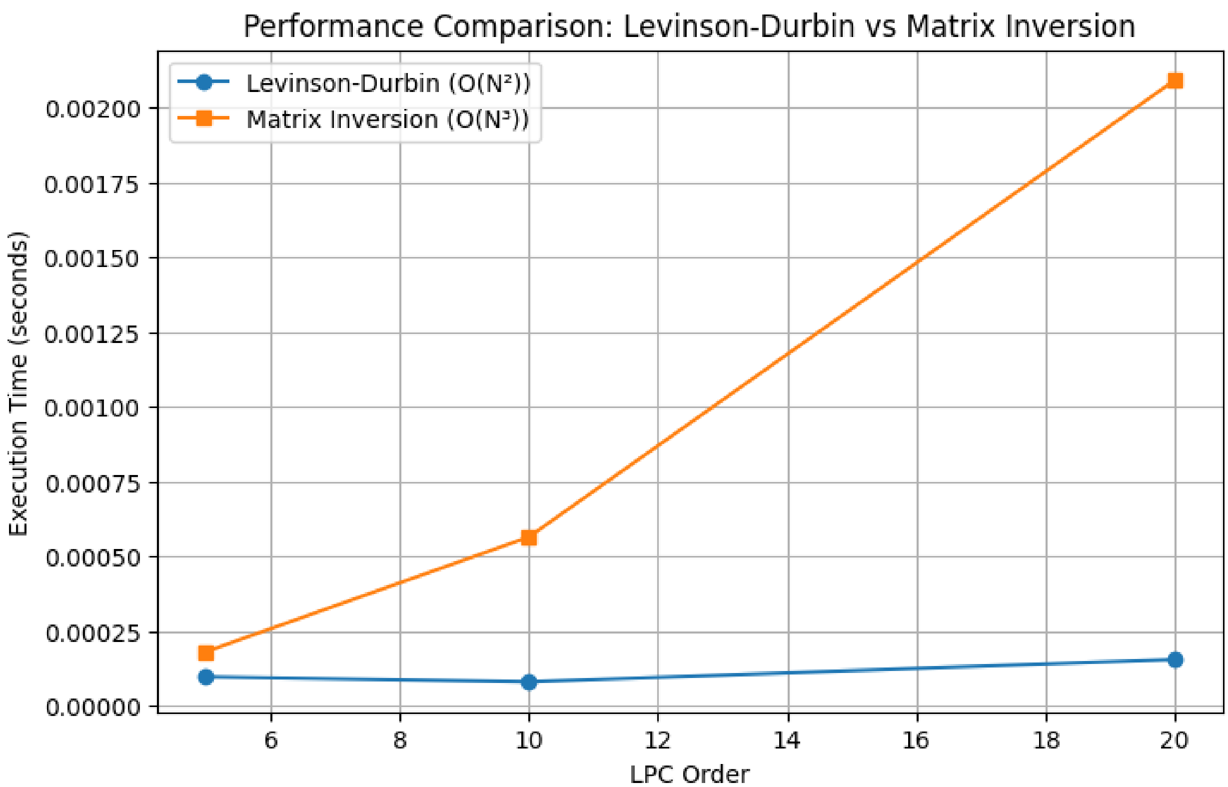
Question 2 assignment 5

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

def autocorr_sequence(x, p_order):
    return np.array([np.dot(x[:len(x)-lag], x[lag:]) for lag in
range(p_order + 1)])

def levinson_durbin_vp(r_seq, p_order):
    # Implements Vaidyanathan's formulation from the Linear Prediction
theory:
    #   λₙ = -[r(n) + Σₖ₌₁ⁿ⁻¹ aₙ₋₁(k) r(n-k)] / Eₙ₋₁
    #   aₙ(k) = aₙ₋₁(k) + λₙ aₙ₋₁(n-k), k = 1,...,n-1 and aₙ(n) = λₙ
    #   Eₙ = (1 - λₙ²) Eₙ₋₁
    coeffs = np.zeros(p_order + 1)
    coeffs[0] = 1.0
    err_energy = r_seq[0]

    if err_energy == 0:
        return np.zeros(p_order), err_energy

    for n in range(1, p_order + 1):
        lam = -np.dot(coeffs[:n], r_seq[n:0:-1]) / err_energy
        new_coeffs = coeffs[:n+1] + lam * np.flip(coeffs[:n+1])
        coeffs[:n+1] = new_coeffs
        err_energy *= (1 - lam**2)

    return -coeffs[1:], err_energy

def lpc_matrix_method(x, p_order):
    R_mat = np.zeros((p_order, p_order))
    r_vec = np.zeros((p_order, 1))

    for i in range(p_order):
        for j in range(p_order):
            seg1 = x[p_order - 1 - i : len(x) - i]
            seg2 = x[p_order - 1 - j : len(x) - j]
            L = min(len(seg1), len(seg2))
            R_mat[i, j] = np.dot(seg1[:L], seg2[:L])
        seg1 = x[p_order - 1 - i : len(x) - 1]
        seg2 = x[p_order:]
        L = min(len(seg1), len(seg2))
        r_vec[i] = np.dot(seg1[:L], seg2[:L])

    try:
        sol = np.linalg.solve(R_mat, r_vec)
    except np.linalg.LinAlgError:
```

```
        sol = np.zeros((p_order, 1))

    return sol.flatten()

inp_signal = sliced
pred_order = 10

r_seq = autocorr_sequence(inp_signal, pred_order)

start_time = time.time()
ld_coeffs, ld_err = levinson_durbin_vp(r_seq, pred_order)
ld_exec_time = time.time() - start_time

start_time = time.time()
mat_coeffs = lpc_matrix_method(inp_signal, pred_order)
mat_exec_time = time.time() - start_time

mse_val = np.mean((ld_coeffs - mat_coeffs)**2)

print(f"LPC Coefficients for Order {pred_order}")
print("Levinson-Durbin Method:", ld_coeffs)
print("Matrix Inversion Method:", mat_coeffs)
print("\nMean Squared Error (MSE):", mse_val)
print(f"Execution Time (Levinson-Durbin): {ld_exec_time:.6f} sec")
print(f"Execution Time (Matrix Inversion): {mat_exec_time:.6f} sec")

orders_to_check = [5, 10, 20]
ld_times = []
mat_times = []

for ord_val in orders_to_check:
    r_seq_temp = autocorr_sequence(inp_signal, ord_val)

    start_time = time.time()
    levinson_durbin_vp(r_seq_temp, ord_val)
    ld_times.append(time.time() - start_time)

    start_time = time.time()
    lpc_matrix_method(inp_signal, ord_val)
    mat_times.append(time.time() - start_time)

plt.figure(figsize=(8, 5))
plt.plot(orders_to_check, ld_times, marker='o', label="Levinson-Durbin
(O(N²))")
plt.plot(orders_to_check, mat_times, marker='s', label="Matrix
Inversion (O(N³))")
plt.xlabel("LPC Order")
plt.ylabel("Execution Time (seconds)")
plt.title("Performance Comparison: Levinson-Durbin vs Matrix
Inversion")
```

```
plt.legend()
plt.grid(True)
plt.show()
```

```
LPC Coefficients for Order 10
Levinson-Durbin Method: [ 2.88395599 -4.75415728  6.65924554 -
8.01590357  8.2720335  -7.46790108
  5.81774498 -3.78533529  1.96202566 -0.60582756]
Matrix Inversion Method: [ 2.89952655 -4.805869    6.74868026 -
8.13789627  8.41487217 -7.61007908
  5.93811526 -3.87266241  2.0117271  -0.62051302]

Mean Squared Error (MSE): 0.009121572827519842
Execution Time (Levinson-Durbin): 0.000426 sec
Execution Time (Matrix Inversion): 0.001027 sec
```



Question 3 Assignment 5

```
import numpy as np

def toeplitz_mat(x, M):
    c_seq = np.array([np.sum(x[:len(x)-k] * x[k:]) for k in
range(M+1)]) / len(x)
    T_mat = np.array([[c_seq[abs(i - j)] for j in range(M)] for i in
range(M)])
```

```python
    return T_mat, c_seq[1:M+1]

def covariance_mat(x, M):
    L = len(x) - M
    A = np.array([x[i:L+i] for i in range(M)])
    C_mat = A @ A.T / L
    b_vec = A @ x[M:L+M] / L
    return C_mat, b_vec

inp = sliced
if len(inp) < 22050:
    inp = np.pad(inp, (0, (22050 + 1 - len(inp))))
M = 10
T_auto, v_auto = toeplitz_mat(inp, M)
C_cov, v_cov = covariance_mat(inp, M)
sol_auto = np.linalg.solve(T_auto, v_auto)
sol_cov = np.linalg.solve(C_cov, v_cov)
c_full = np.array([np.sum(inp[:len(inp)-k] * inp[k:]) for k in
range(M+1)]) / len(inp)
sol_ld, _ = levinson_durbin_algorithm(c_full, M)
err_auto = np.mean((sol_ld - sol_auto) ** 2)
err_cov = np.mean((sol_ld - sol_cov) ** 2)
print("Autocorrelation Method:", sol_auto)
print("Covariance Method:", sol_cov)
print("Levinson-Durbin:", sol_ld)
print("LD matches Autocorrelation Method:", np.allclose(sol_auto,
sol_ld))
print("LD matches Covariance Method:", np.allclose(sol_cov, sol_ld))
if err_auto < err_cov:
    print("\nThe Autocorrelation Method is closer to the LD
recursion.")
else:
    print("\nThe Covariance Method is closer to the LD recursion.")
```

```
Autocorrelation Method: [ 2.9162736  -4.8617277   6.8455887  -8.27033
8.570287   -7.7652454
  6.0699315  -3.9687324   2.066714   -0.63684595]
Covariance Method: [-0.60748875  1.9654255  -3.788144    5.8191524  -
7.4672003   8.268566
 -8.010928    6.6542745  -4.750216    2.8823647 ]
Levinson-Durbin: [ 2.91627357 -4.86172784  6.84558888 -8.27033048
8.57028706 -7.76524533
  6.06993134 -3.96873241  2.066714   -0.63684594]
LD matches Autocorrelation Method: True
LD matches Covariance Method: False

The Autocorrelation Method is closer to the LD recursion.
```

Question 1

$$e[n] = x[n] + \sum_{i=1}^{N} a^*_{N,i}\, x[n-i]$$

$$E[e[n]\, x[n-j]] = 0 \quad \text{--- (1)}$$

$$E\left[\left(x[n] + \sum_{i=1}^{N} a^*_{N,i}\, x[n-i]\right)\left(x[n-i]\right)\right] = 0$$

$$E[x[n]\ x[n-i]] + \sum_{i=1}^{N} a^*_{N,i}\, E\left[x[n-i]\, x[n-j]\right] = 0$$

$$R(i) + \sum_{i=1}^{N} a^*_{N,i}\, R(j-i) = 0 \quad \text{--- (2)}$$

$$i \in \{1, N\}$$

And

$$R(0) + \sum_{i=1}^{N} a^{*}_{N,i} R(i) = \epsilon_N^{f} \quad \text{--- (3)}$$

We have $(N+1)$ equation and $(N+1)$ unknowns. Hence we can find $R(0), R(1), \ldots, R(N)$