Signals and Systems

Computer Assignment 4 Report

University of Tehran

School of Electrical and Computer Engineering

Student Name:	Parsa Bukani
Student Number:	810102501
Course:	Signals and Systems
Instructor:	Dr. Akhavan
Semester:	Fall 1403

November 30, 2025

Contents

1 Introduction

This report presents the complete solution, MATLAB codes, figures, and analysis for Computer Assignment 4 of the Signals and Systems course. The assignment consists of two major parts: **Amplitude Coding Simulation** and **Classification using MATLAB's Classification Learner**. Each subsection includes explanations, figures, and source code when required.

2 Part 1: Amplitude Coding

2.1 Exercise 1-1: Mapset Creation

We construct a 2×32 cell array named `Mapset`. Row 1 contains the 32 allowed characters, and row 2 contains their assigned 5-bit binary codes.

```
Nch = 32;
mapset = cell(2, Nch);
Alphabet = 'abcdefghijklmnopqrstuvwxyz␣.,!";';
for i = 1:Nch
    mapset{1,i} = Alphabet(i);
    mapset{2,i} = dec2bin(i-1, 5);
end
```

2.2 Exercise 1-2: Amplitude Coding

The message is first converted to binary using `message2binary`. If the binary length is not divisible by the bitrate, the message is padded with ";" until it becomes divisible. Each group of `bitrate` bits is then mapped to an amplitude level $\alpha = \frac{\text{decimal(bits)}}{2^{\text{bitrate}}-1}$ and transmitted as a 1-second sinusoid.

```
function codedMessage = amp_coding(message, bitrate, mapset)
    binMessage = message2binary(message, mapset);
    while mod(length(binMessage), bitrate) ~= 0
        message = [message, ';'];
        binMessage = message2binary(message, mapset);
    end
    fs = 100; t = 0:1/fs:1-1/fs; codedMessage = [];
    for i = 1:bitrate:length(binMessage)
        bits = binMessage(i:i+bitrate-1);
        alpha = bin2dec(bits)/(2^bitrate - 1);
        codedMessage = [codedMessage, alpha*sin(2*pi*t)];
    end
end
```

2.3 Exercise 1-3: Plot of Coded Signal for "signal"

The function `amp_coding` was applied to the message "signal" for bitrates 1, 2, and 3 bit/s. Each symbol corresponds to a 1-second sinusoid with amplitude determined by the encoded bits.

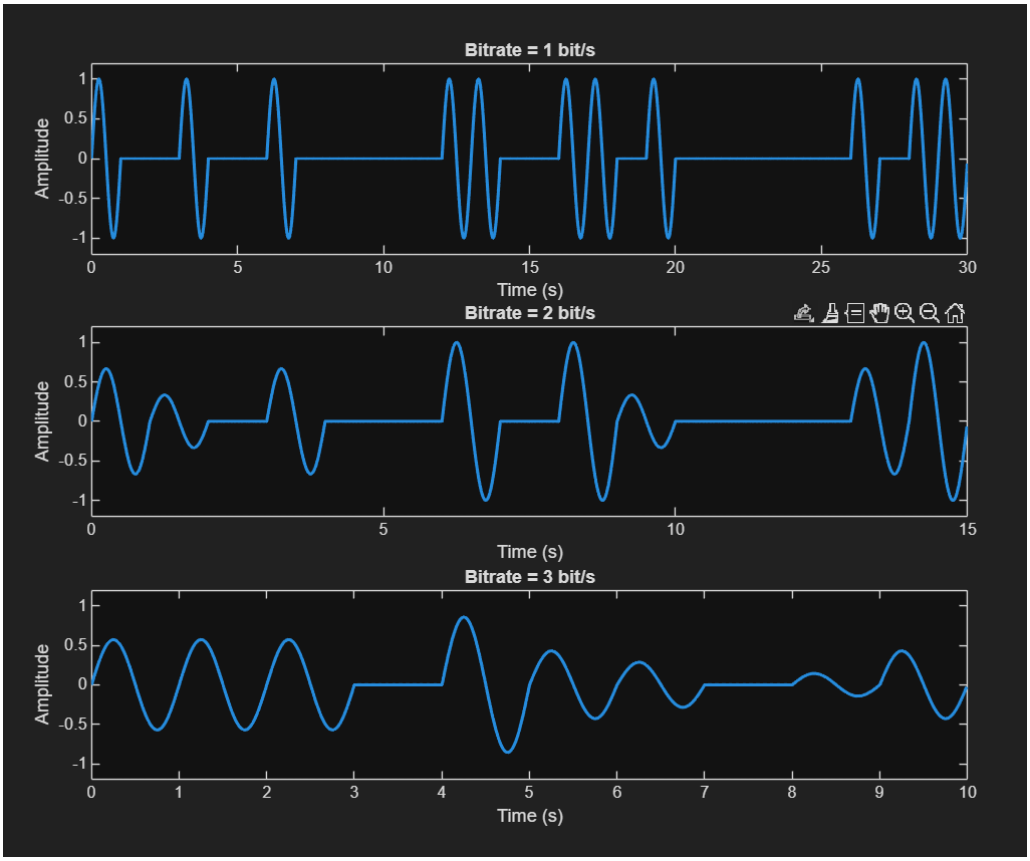


Figure 1: Transmitted signal for the message "signal" at bitrates 1, 2, and 3.

2.4 Exercise 1-4: Amplitude Decoding

To decode the transmitted signal, each 1-second block (100 samples) is correlated with the reference signal $2\sin(2\pi t)$. The discrete correlation is computed as:

$$\text{corr}(a, b) = 0.01 \cdot (a \cdot b)$$

This value lies between 0 and 1 and corresponds to one of the amplitude levels used during encoding. The identified level is converted back to its binary representation, and every 5 bits are mapped to a character using the `Mapset`.

MATLAB Code

Correlation function:

```
function correlation = corr(a, b)
    correlation = 0.01 * dot(a, b);
    correlation = max(0, correlation);
    correlation = min(1, correlation);
end
```

Binary-to-character function:

```
function character = getCharacter(binary, mapset)
    for j = 1:size(mapset,2)
        if strcmp(mapset{2,j}, binary)
            character = mapset{1,j};
            return;
        end
    end
    character = '?'; % fallback
end
```

Decoding function:

```
function decodedMessage = amp_decoding(codedMessage, bitrate,
   mapset)

fs = 100;
t = 0:1/fs:1-1/fs;
template = 2 * sin(2*pi*t);

binarizedMessage = '';
decodedMessage = '';

numSymbols = length(codedMessage) / fs;

for i = 1:numSymbols
    segment = codedMessage((i-1)*fs + 1 : i*fs);
    c = corr(segment, template);

    for j = 0:(2^bitrate - 1)
        center = j / (2^bitrate - 1);
        tol = 0.5 / (2^bitrate - 1);

        if center - tol <= c && c <= center + tol
            binarizedMessage = [binarizedMessage, dec2bin(j,
               bitrate)];
            break;
        end
    end
end

for k = 1:5:length(binarizedMessage)
```

```
    character = getCharacter(binarizedMessage(k:k+4), mapset);
    decodedMessage = [decodedMessage, character];
end

end
```

Testing on the results of Exercise 1-3:

```
for bitrate = 1:3
    decodedMessage = amp_decoding(coded{bitrate}, bitrate, mapset
       );
    fprintf('Bitrate␣%d␣   ␣Decoded:␣%s\n', bitrate,
       decodedMessage);
end
```

Decoding Results

Running the decoding function on the three coded signals produced in Exercise 1-3 yields the following results:

```
Bitrate 1     Decoded Message: signal
Bitrate 2     Decoded Message: signal
Bitrate 3     Decoded Message: signal
```

As expected, the decoded message matches the original message “signal“ for all three bitrates, confirming that the implementation of the encoding and decoding functions is correct in the noise-free scenario.

2.5 Exercise 1-5: Properties of Gaussian Noise

In this part, we generate 3000 samples using `randn(1,3000)` and verify that the produced noise is Gaussian with zero mean and unit variance.

MATLAB Code

```
noise = randn(1,3000);

noise_mean = mean(noise);
noise_var  = var(noise);

fprintf("Mean  = %.4f\n", noise_mean);
fprintf("Var   = %.4f\n", noise_var);
```

```
figure;
histogram(noise, 40, 'Normalization', 'pdf');
hold on;
x = linspace(-4,4,200);
plot(x, normpdf(x, 0, 1), 'r', 'LineWidth', 2);
```

Results

The numerical results confirm:

- Mean = 0.0142
- Variance = 1.025
- Histogram closely matches the PDF of $\mathcal{N}(0, 1)$

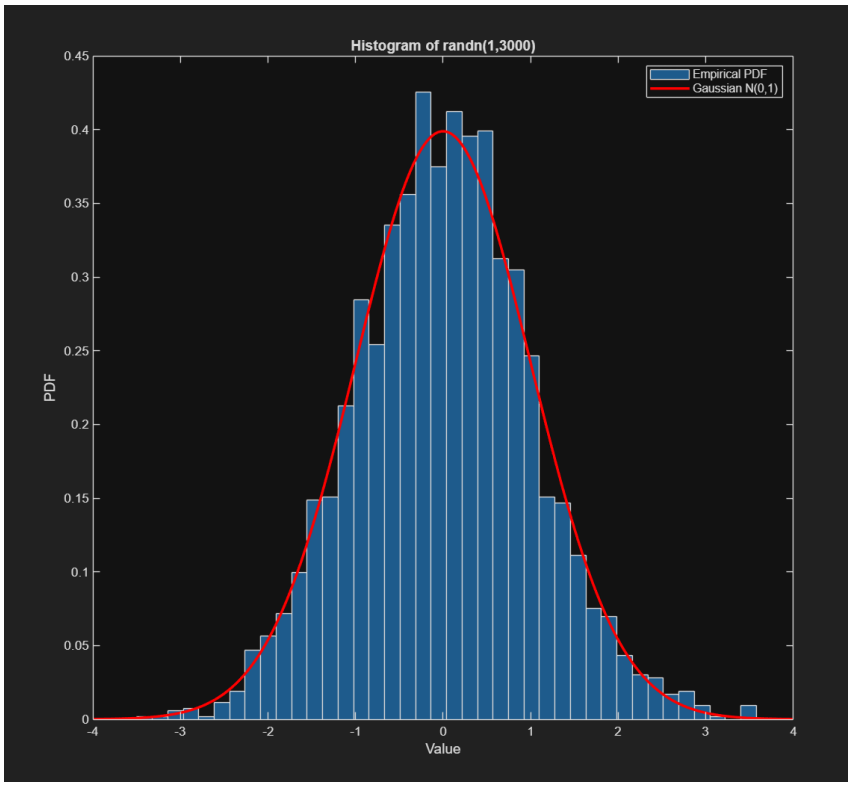


Figure 2: Histogram of 3000 samples from `randn`, overlaid with the theoretical Gaussian PDF.

2.6 Exercise 1-6: Decoding in the Presence of Noise

A Gaussian noise sequence with variance 0.0001 (standard deviation 0.01) was added to the encoded signal. Each noisy signal was then decoded using the same correlation-based method described in Exercise 1-4.

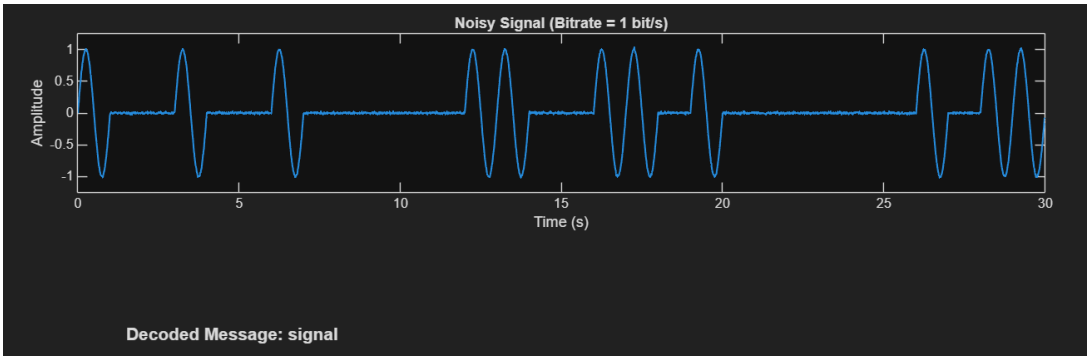
MATLAB Code

```
std_noise = 0.01;
coded{bitrate} = amp_coding(message, bitrate, mapset);
noise = std_noise * randn(1, length(coded{bitrate}));
noisy{bitrate} = coded{bitrate} + noise;
decodedMessage = amp_decoding(noisy{bitrate}, bitrate, mapset);
```

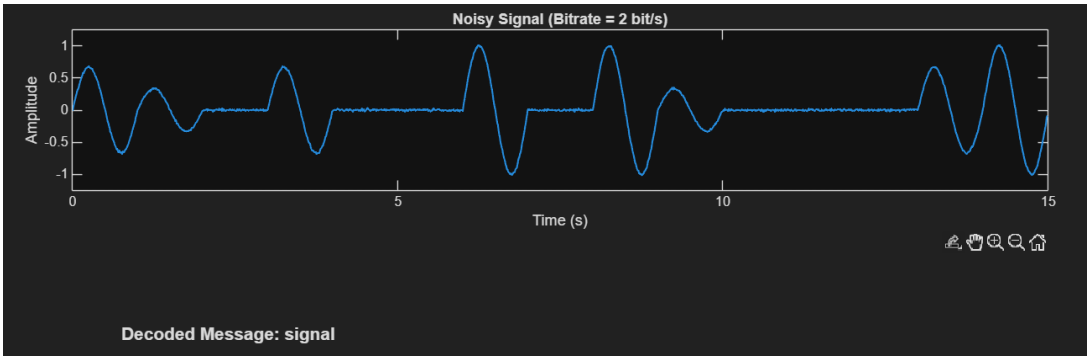
Decoding Results

```
Bitrate 1     signal
Bitrate 2     signal
Bitrate 3     signal
```

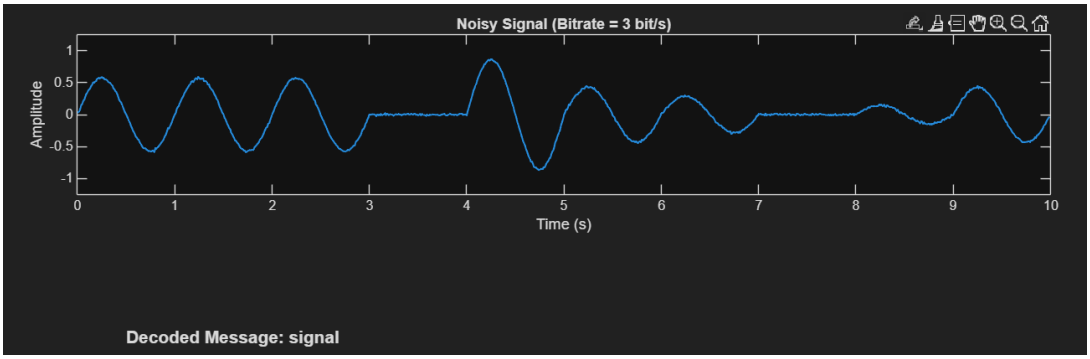
Despite the added noise, all three bitrates successfully recover the original message "signal". As expected, bitrate 3 exhibits slightly more distortion, but the decoding remains correct due to the small noise variance.



(a) Noisy encoded signal — Bitrate 1



(b) Noisy encoded signal — Bitrate 2



(c) Noisy encoded signal — Bitrate 3

Figure 3: Noisy encoded signals (variance = 0.0001) for the three bitrates.

2.7 Exercise 1-7: Effect of Increasing Noise Power

To study the robustness of different bitrates, Gaussian noise with three standard deviations ($\sigma = 0.1,\ 0.5,\ 1$) was added to the encoded signal. Each noisy signal was decoded, and the results were compared.

Observations

- For $\sigma = 0.1$, all bitrates decode the message correctly.
- For $\sigma = 0.5$, bitrate 3 shows noticeable distortion and is less reliable.

- For $\sigma = 1$, decoding becomes highly unreliable for bitrates 3 and 2.
- Bitrate 1 is the most noise-resistant, matching the theoretical expectation that larger amplitude spacing improves robustness.

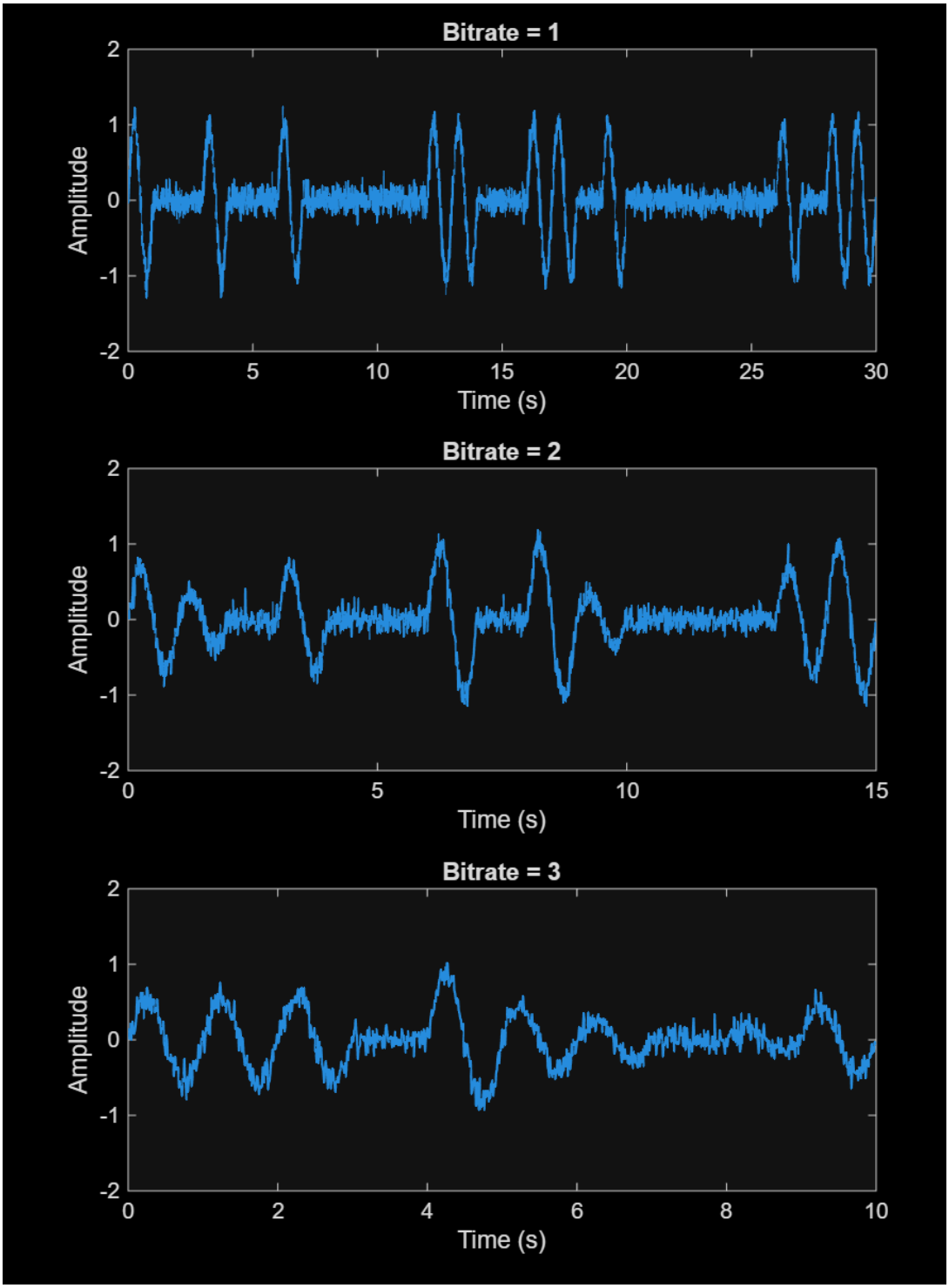


Figure 4: Decoded results for noise standard deviation $\sigma = 0.1$.

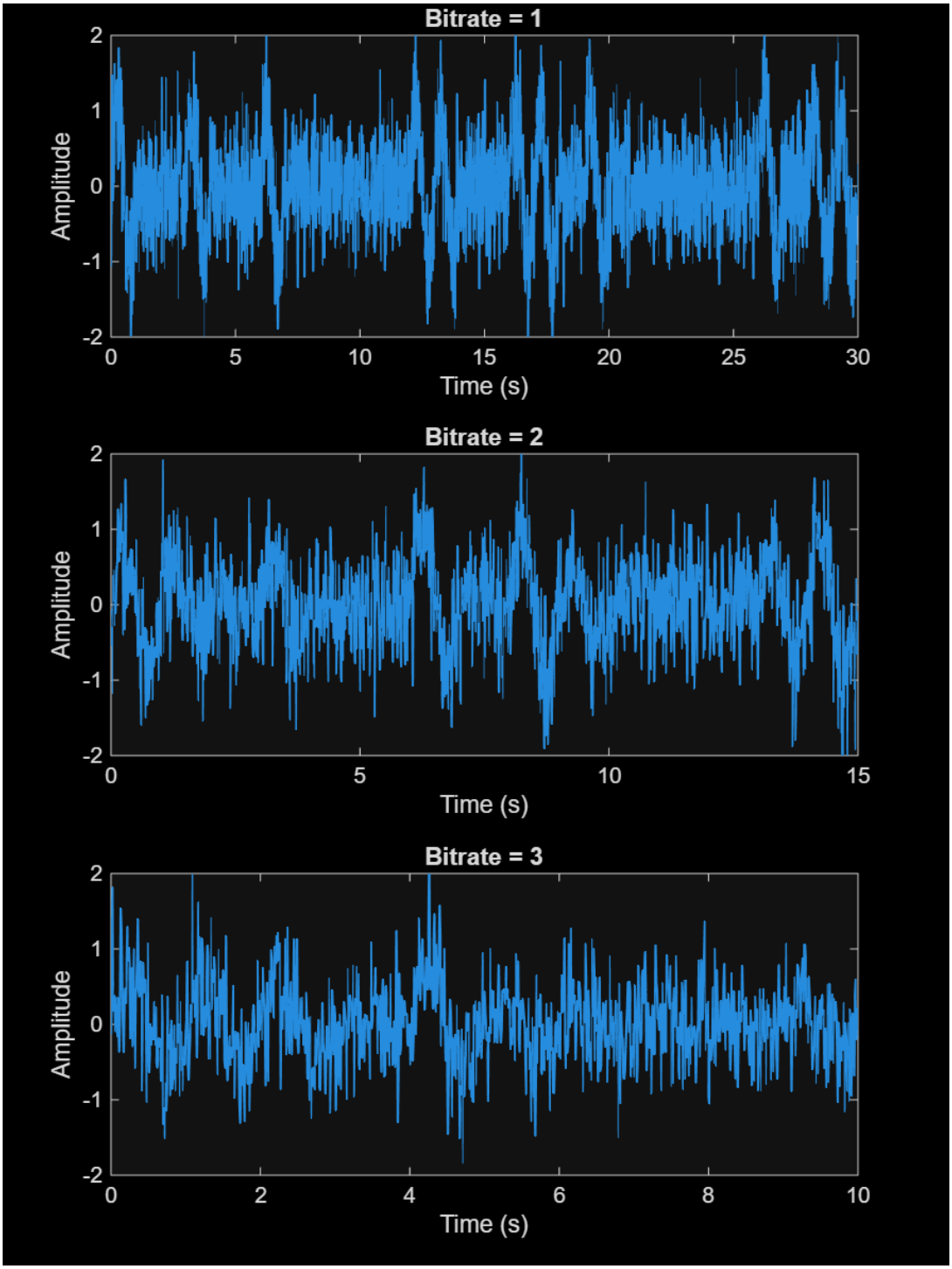


Figure 5: Decoded results for noise standard deviation $\sigma = 0.5$.

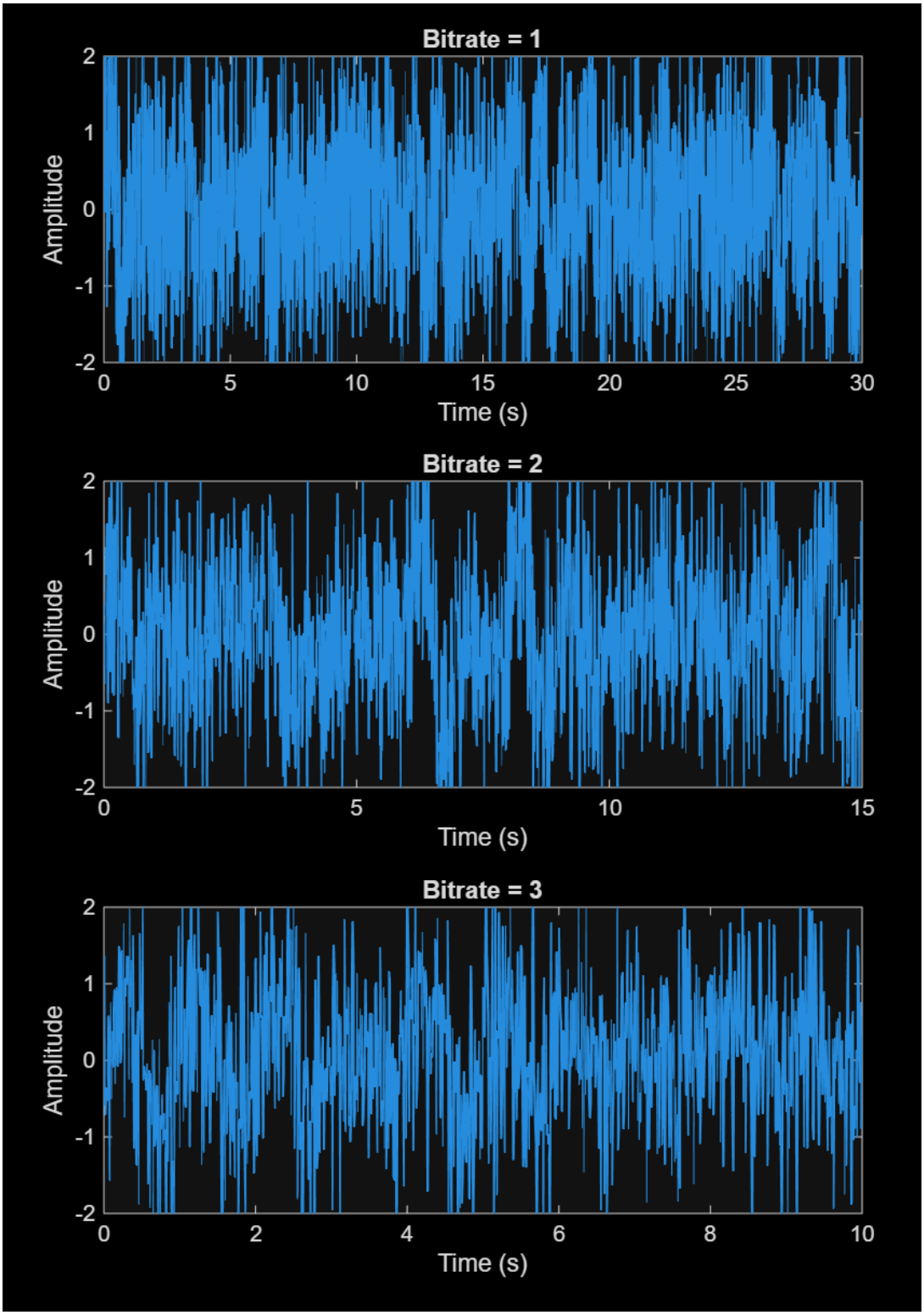


Figure 6: Decoded results for noise standard deviation $\sigma = 1$.

2.8 Exercise 1-8: Maximum Tolerable Noise

Using repeated simulations with increasing noise power, the maximum standard deviation σ for which the decoded message remains correct (or nearly correct) was estimated for

each bitrate.

Approximate Results

- **Bitrate 1:** $\sigma \approx 1.8$–1.9

$$\text{variance} = \sigma^2 \approx 3.24\text{–}3.61$$

- **Bitrate 2:** $\sigma \approx 0.4$–0.5

$$\text{variance} \approx 0.16\text{–}0.25$$

- **Bitrate 3:** $\sigma \approx 0.20$–0.25

$$\text{variance} \approx 0.04\text{–}0.0625$$

These results clearly show that **lower bitrates are significantly more robust to noise**. This matches the theoretical explanation: when the amplitude levels are closer together (high bitrate), noise is more likely to push the correlation value into the wrong interval, causing decoding errors.

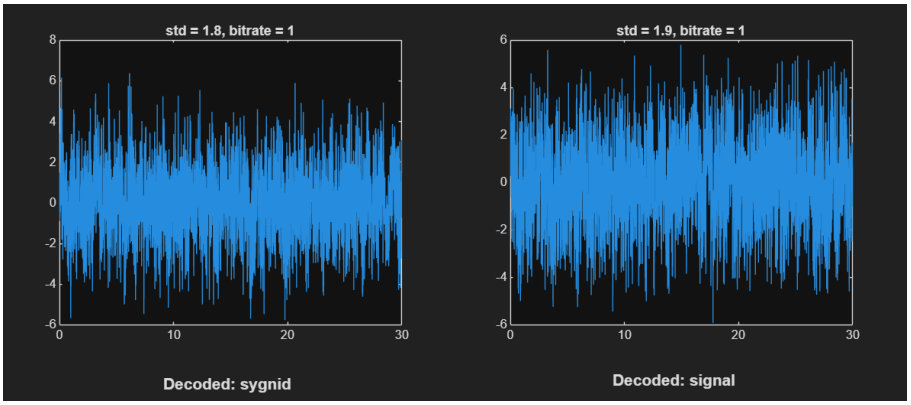


Figure 7: Maximum tolerable noise for bitrate 1.

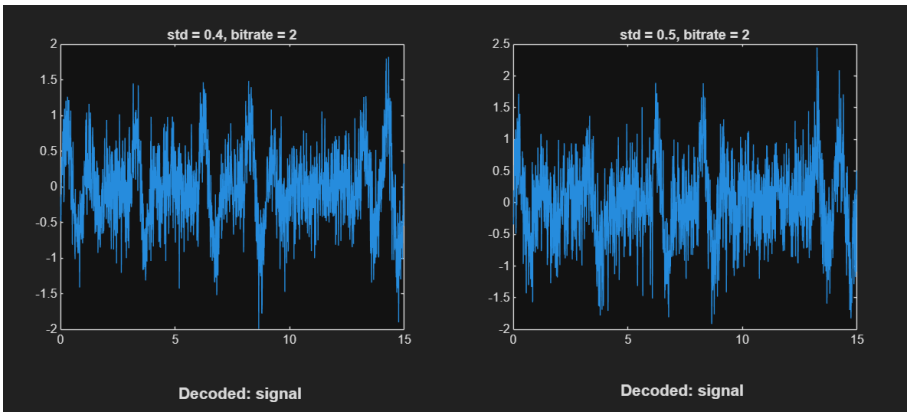


Figure 8: Maximum tolerable noise for bitrate 2.

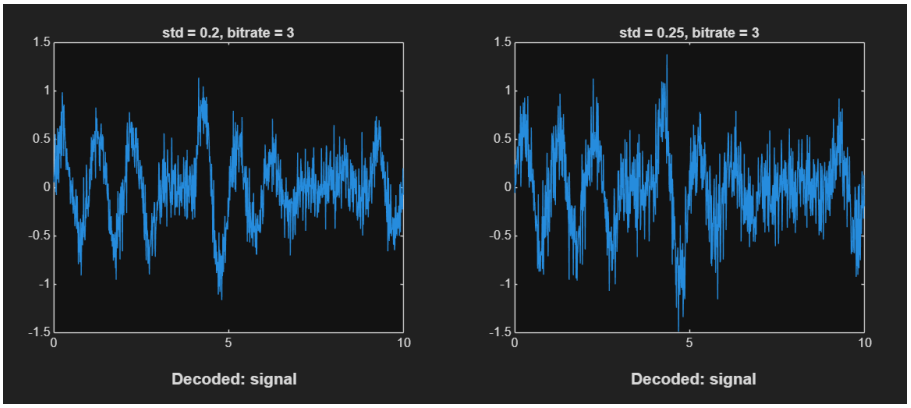


Figure 9: Maximum tolerable noise for bitrate 3.

2.9 Exercise 1-9: Improving Bitrate Robustness

Increasing the transmitter power increases the amplitude spacing between possible symbol levels. Larger spacing makes decision thresholds farther apart, reducing the chance that noise pushes the correlation value into an incorrect interval. Therefore, higher power improves noise robustness.

2.10 Exercise 1-10: Maximum Bitrate in a Noiseless System

In the absence of noise, the bitrate can theoretically be increased without limit by adding more amplitude levels. In practice, the limitation comes from finite numerical precision (e.g., 32-bit floating-point), which eventually makes adjacent symbol amplitudes indistinguishable.

2.11 Exercise 1-11: Effect of Scaling by $10\sin(2\pi t)$

Scaling the correlation template by 10 increases both the signal correlation and the noise correlation proportionally. Since the signal-to-noise ratio (SNR) does not improve, the system does not become more noise-resistant.

2.12 Exercise 1-12: Comparison with ADSL Speed

Typical home ADSL connections reach up to 16 Mbps (16 million bits per second). In contrast, the maximum bitrate used in this assignment was only 3 bits/s, illustrating how simplified the simulated channel is compared to real-world communication systems.

3 Part 2: Classification Learner

3.1 Exercise 2-1: Training a Linear SVM Classifier

The dataset `diabetes-training` was imported into MATLAB and used inside the *Classification Learner* app. A new session was created from the workspace using the default settings, including 5-fold cross-validation. From the model list, a **Linear SVM** classifier was selected and trained.

The resulting cross-validated accuracy reported by the app was:

$$\text{Accuracy} = \boxed{77.3\%}$$

A screenshot of the trained model and the Classification Learner performance summary is shown below.

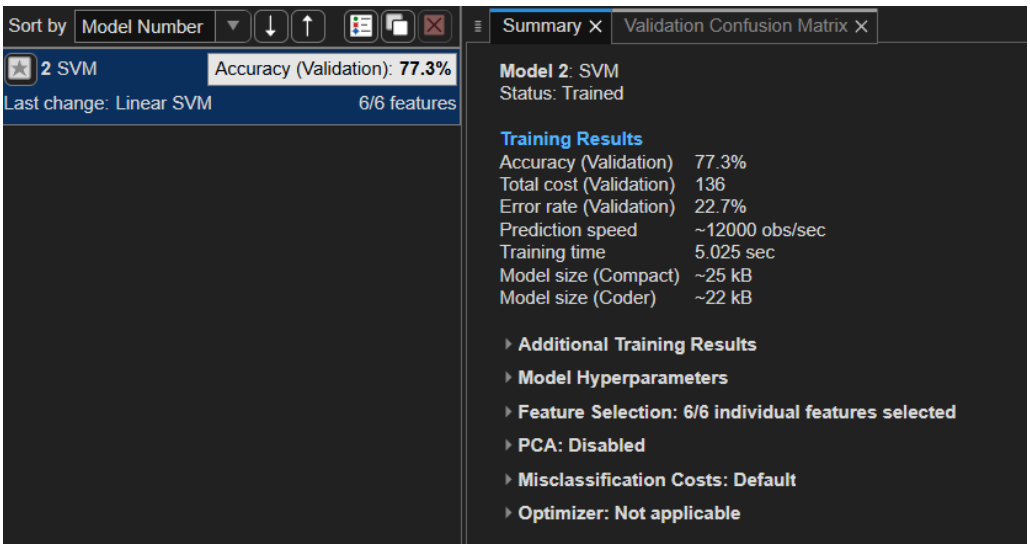


Figure 10: Performance and accuracy of the Linear SVM model in MATLAB Classification Learner.

3.2 Exercise 2-2: Effect of Individual Features

Using the *Feature Selection* tool in the Classification Learner app, each of the six medical features was tested individually. For each experiment, only a single feature was enabled and a Linear SVM classifier was trained using the default 5-fold cross-validation settings.

Table 1 shows the obtained accuracies.

Feature	Accuracy (%)
Age	65.3
BMI	65.2
Insulin	65.3
SkinThickness	65.3
BloodPressure	65.3
Glucose	**74.5**

Table 1: Classification accuracy when using each feature independently.

From the results, the feature with the strongest predictive power is clearly:

$$\boxed{\text{Glucose}}$$

which achieves an accuracy of 74.5%, significantly higher than the remaining features (all around 65%). This indicates that blood glucose level is the single most informative indicator of diabetes in this dataset.

3.3 Exercise 2-3: Manual Evaluation of the Trained Model

The trained Linear SVM classifier was exported from the Classification Learner app using the *Export Model* option. This produced a structure named `TrainedModel`, which includes the function handle `predictFcn` for performing predictions on new input data.

To verify the classifier's performance on the training dataset, the exported model was applied to the original `diabetes-training` table. The true labels were extracted from the last column of the table using `table2array`, and the fraction of correctly predicted samples was computed. The MATLAB code used for this evaluation is shown below:

```
predictedLabels = TrainedModel.predictFcn(diabetes_training);
trueLabels = table2array(diabetes_training(:, end));

correct = sum(predictedLabels == trueLabels);
accuracy_manual = correct / length(trueLabels) * 100;

fprintf('Training-phase accuracy = %.2f%%\n', accuracy_manual);
```

The manually computed training accuracy was found to be:

$$\boxed{77.50\%}$$

Note: This value is slightly higher than the cross-validation accuracy reported earlier (77.3%). This behavior is expected. The Classification Learner reports *cross-validation*

accuracy, which measures performance on unseen folds of the data, whereas the manual computation evaluates *training accuracy*, where the model predicts labels for the same data it was trained on. Training accuracy is usually slightly higher due to the model having already seen these samples.

3.4 Exercise 2-4: Evaluation on the Validation Dataset

A separate dataset named `diabetes-validation`, containing the medical features and diabetes labels for 100 new individuals, was imported into MATLAB. Using the previously exported classifier structure `TrainedModel`, the function handle `predictFcn` was applied to these unseen samples.

The true labels (stored in the last column of the table) were extracted using `table2array`, and the percentage of correctly predicted samples was computed using the following code:

```
pred_val = TrainedModel.predictFcn(diabetes_validation);
true_val = table2array(diabetes_validation(:, end));

correct_val = sum(pred_val == true_val);
accuracy_test = correct_val / length(true_val) * 100;

fprintf('Test-phase accuracy = %.2f%%\n', accuracy_test);
```

The resulting evaluation (test-phase) accuracy on the new data was:

$$\boxed{78.00\%}$$

This accuracy reflects the classifier's true generalization performance on previously unseen individuals. As expected, the result is close to the cross-validation accuracy obtained during training, indicating that the model generalizes reasonably well to new data.