Digital Communications Project Report

Channel-related questions

a) Where is one of the four bands randomly selected and removed?

```matlab
range_deleted = randi(4);  % <- Randomly selects one of the four bands
...
f_low = (range_deleted - 1) * 2000 + 1000;
f_high = f_low + 2000;
...
H(idx1:idx2) = 0;
```

The function randomly selects one of the four frequency bands (each 2 kHz wide, starting at 1 kHz).

It then zeros out the corresponding band in the filter `H` (i.e., removes that frequency band from the signal).

How to mitigate this effect :

FDM (FREQUENCY DIVISION MULTIPLEXING)+Redundancy(Frequency Diversity)

Frequency Diversity: At the **transmitter**, replicate the signal across **multiple bands** (e.g., 4-band redundant transmission).

At the **receiver**, try to recover the signal from **any band that survived**.

This combats **band deletion** by ensuring the message isn't solely dependent on a single frequency band.

b) Where is the upper mirror band removed for symmetry?

```matlab
f_low_mirror = Fs - f_high;
f_high_mirror = Fs - f_low;
...
H(idx3:idx4) = 0;
```

Because the FFT is symmetric for real signals, the mirror frequency band (on the upper side of the spectrum) must also be removed to maintain correct signal processing.

$$X(-f) = conj(X(f))$$

If our signal is real then the magnitude part is even symmetric over zero:

$$\mid X(f) \mid = \mid X(-f) \mid$$

The phase part is odd symmetrical with respect to zero:

$$\angle X(-f) = -\angle X(f)$$

$$\rightarrow X(-f) = conj(X(f))$$

These lines remove the mirror of the deleted band to ensure spectral symmetry in H

c) Where is asynchrony introduced?

```matlab
delay1 = randi([2000, 20000]);
delay2 = randi([2000, 50000]);
...
r = [noise_delay1, noisy_sig, noise_delay2];
```

These lines add **random delay** (before and after the signal) by inserting noise sequences (`noise_delay1`, `noise_delay2`).

This creates **temporal misalignment** between transmitted and received signals, simulating **asynchrony**.

Impact:

In **time domain**: Signal appears shifted and surrounded by noise.

In **frequency domain**: No shift in frequency, but **cross-correlation with the pilot** may become harder due to added delays and noise.

Suggested solution:

Use **pilot-based synchronization**: Embed a known pilot sequence in the signal and perform **cross-correlation** at the receiver to **locate and align** the signal.

Perform synchronization **before demodulation and decoding**.

d) Where is signal clipping performed?

```matlab
r = min(max(r, -1), 1);
```

This clips the output signal r to the range [-1, 1] to simulate hardware saturation or DAC/ADC limitations.

Any values beyond this range are forced to -1 or +1.

e) Add an average energy constraint for the input signal

First I Compute average signal energy

Let the signal be:

```matlab
v = s_in;
```

Then

```matlab
E_avg = mean(v.^2);  % Average power (energy per sample)
```

and then we decide the scaling factor :

$$If\ \ E_{avg} < 1\ \ ,\ scale\ the\ signal\ and\ noise\ by\ \sqrt{s} = \sqrt{\frac{1}{E_{avg}}}\ Otherwise,\ leave\ unchanged.$$

and here is the modified code that we should insert at the start of the function :

```matlab
%here I Normalize energy if below threshold
E_avg = mean(s_in.^2);

if E_avg < 1
    s = 1 / E_avg;
    sqrt_s = sqrt(s);

    s_in = s_in * sqrt_s;      % Scale signal
    sigma = sigma * sqrt_s;    % Scale noise accordingly to preserve SNR
end
```

If signal energy is too low, the SNR degrades because the fixed noise power dominates.

Scaling both the signal and noise by Sqrt(s) keeps :

$$SNR = \frac{Signal\ Power}{Noise\ Power} = \frac{s\ .\ E_{avg}}{s\ .\ \sigma^2} = \frac{E_{avg}}{\sigma^2}$$

Hence, SNR remains **unchanged**, and the system behaves consistently regardless of input energy

And Now I am gonna explain my code line by line !!:

Section 1: System Parameters:

```matlab
M = 16;
Fs = 22050;
sps = 16;
rolloff = 0.25;
span = 10;
sigma = 0.005;
alpha = 1;
text_input = 'Bits are the basic units of information and the universal currency of communication in the digital age They carry data across networks devices and systems forming the language behind everything from simple messages to complex applications like video calls online gaming and AI In todays connected world bits are the invisible threads linking our communication infrastructure enabling fast data transfer cloud services and multimedia Without them the constant flow of information that powers innovation education and global interaction would not exist Their role keeps growing as technology advances making bits the foundation of digital life';
```

`M`: Modulation order (16-PAM)

`Fs`: Sampling frequency

`sps`: Samples per symbol (oversampling factor)

`rolloff`, `span`: Parameters for root raised cosine (RRC) pulse shaping

`sigma`: Noise standard deviation

`alpha`: Passband gain

Alpha's Definition:

alpha is a scaling factor applied to the passband (modulated) signal before transmission. It models channel attenuation or amplification—in other words, it controls how much of the transmitted signal power is preserved when going through the channel.

Impact on System:

A lower alpha (e.g., 0.1) simulates attenuation, mimicking a weak or lossy channel.

A higher alpha (e.g., 1 or more) simulates a stronger or amplified channel.

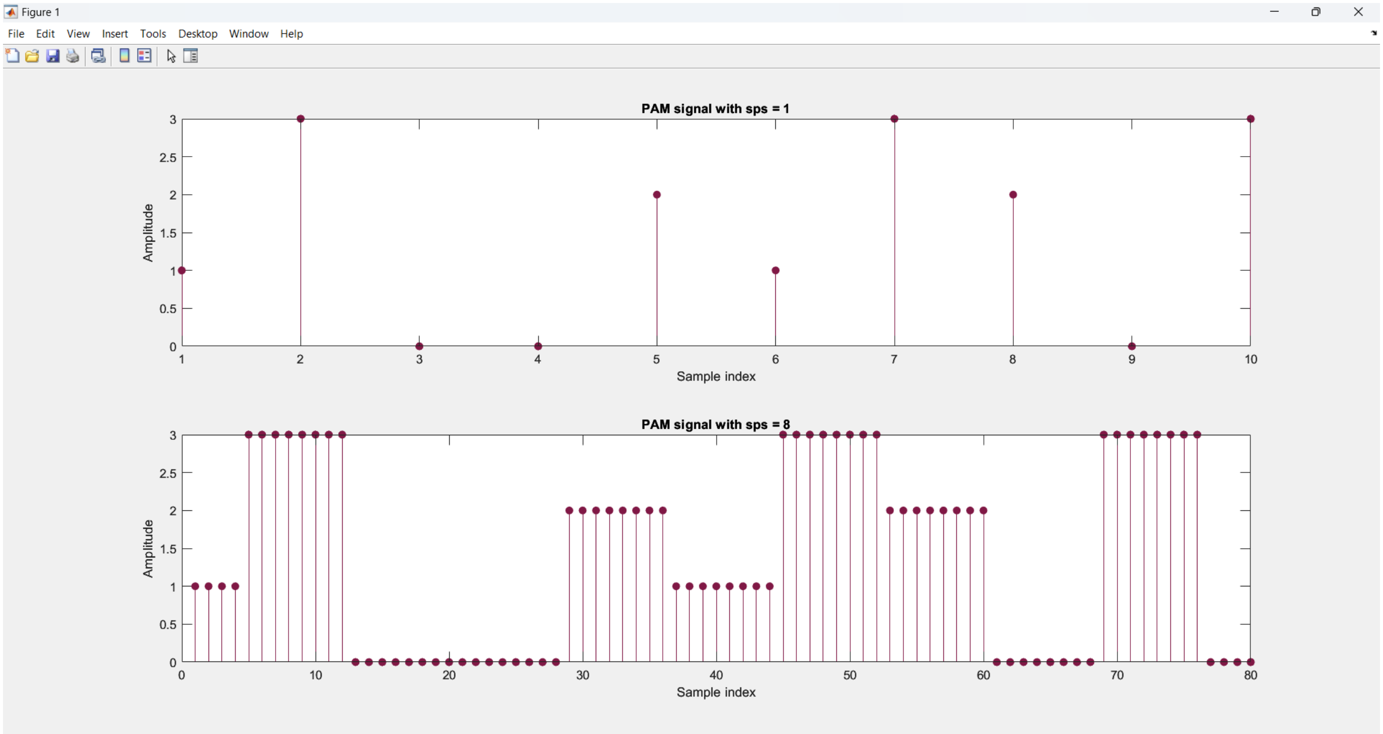
The received signal's Signal-to-Noise Ratio (SNR) directly depends on alpha.

SPS's Definition:

sps stands for Samples Per Symbol and controls how many digital samples represent each modulated symbol. It is a key parameter in digital pulse shaping and filtering.

For understanding better :
Let's create a very simple example with PAM to see how the shape of the sampled signal changes when we set sps = 1 or sps = 8. Here sps stands for samples per symbol, meaning how many samples per symbol we have in the output signal.

```matlab
clc;
clear;
close all;
M = 4;                    % 4-PAM
sps_values = [1, 8];
numSymbols = 10;
symbolValues = randi([0 M-1], 1, numSymbols);
ampLevels = 0:M-1;
myColor = [128, 24, 69]/255;

for idx = 1:length(sps_values)
    sps = sps_values(idx);
    x = upsample(ampLevels(symbolValues+1), sps);
    h = ones(1, sps);
    tx = conv(x, h, 'same');

    subplot(length(sps_values),1,idx);
    stem(tx, 'filled', 'Color', myColor);
    title(['PAM signal with sps = ', num2str(sps)]);
    xlabel('Sample index'); ylabel('Amplitude');
end
```



So the higher the SPS, the better, right? So why don't we suddenly set the SPS to much higher values so that our signal becomes smoother?

raising the sps makes the signal smoother and closer to ideal, and applying filters like Raised Cosine works better. But raising the sps too much also has problems:

-Increased memory and computational time

The higher the sps, the more samples the signal will have.

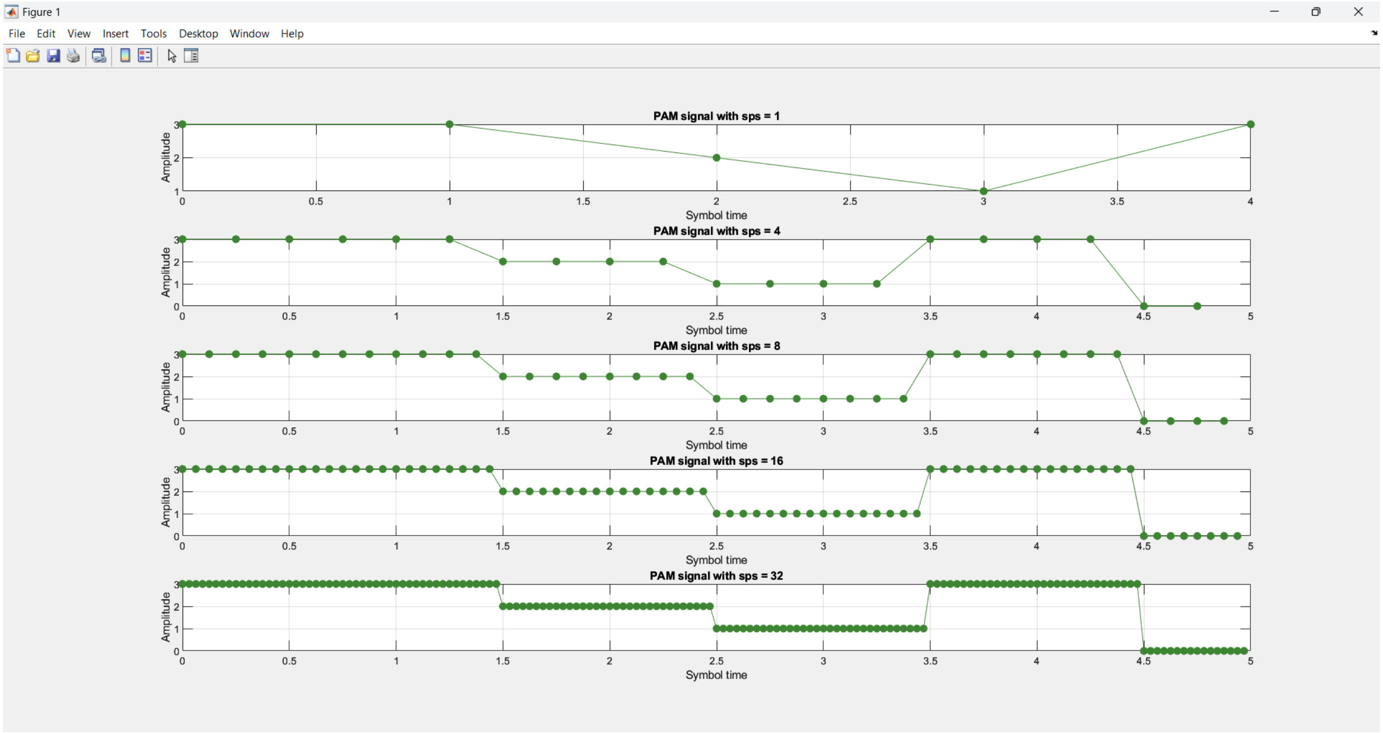
For example, if we have 1000 symbols and set sps = 1000, the output signal will have 1 million samples. This will increase RAM consumption and processing time.

-No further practical benefit (I will show it in a plot !)

After a certain point, increasing the SPS no longer has a significant effect on signal quality, it just increases data volume and computation.

Let's see a plot with different SPS values so you can see how the signal quality improves but after a while its not necessary !

```matlab
clc;
clear;
close all;
M = 4;
numSymbols = 5;
symbolValues = randi([0 M-1], 1, numSymbols);
ampLevels = 0:M-1;
sps_list = [1, 4, 8, 16 ,32];
myColor = [62, 133, 51]/255;
figure;
for idx = 1:length(sps_list)
    sps = sps_list(idx);
    x = upsample(ampLevels(symbolValues+1), sps);
    h = ones(1, sps);
    tx = conv(x, h, 'same');    % PULSE SHAPING !
    t = (0:length(tx)-1)/sps;
    subplot(length(sps_list),1,idx);
    plot(t, tx, '-o', 'Color', myColor, 'MarkerFaceColor', myColor);
    grid on;
    title(['PAM signal with sps = ', num2str(sps)]);
    xlabel('Symbol time'); ylabel('Amplitude');
end
```



See? From a certain point on, increasing the SPS is no longer of any value to us and only imposes **heavier processing** and **memory issues** on us.

Section 2: Text to Binary

```matlab
ascii_vals = double(text_input);
bin_data = de2bi(ascii_vals, 8, 'left-msb')';
bin_data = bin_data(:);   %One-dimensional column vector
```

Converts each ASCII character into 8-bit binary (MSB first).

Section 3: Huffman Source Coding

```matlab
symbols = unique(bin_data);   % returns 0 , 1
if length(symbols) < 2, symbols = [0; 1]; prob = [0.5; 0.5]; else ...
dict = huffmandict(symbols, prob);
huff_encoded = huffmanenco(bin_data, dict);
```

Builds a Huffman dictionary and compresses the binary data.

-If the data contains only one type of bit (for example, only 0 or only 1), the probability of both symbols (0 and 1) is artificially set to 0.5 so that the Huffman code can be constructed. Because the Huffman code requires at least two symbols.

Section 4: Hamming (7,4) Channel Coding

```matlab
G = [1 0 0 0 1 1 0;
     0 1 0 0 1 0 1;
     0 0 1 0 0 1 1;
     0 0 0 1 1 1 1];
pad_size = mod(4 - mod(msg_len, 4), 4);
huff_padded = [huff_encoded; zeros(pad_size, 1)];
...
coded_matrix = mod(msg_matrix * G, 2);
coded_data = coded_matrix';
coded_data = coded_data(:);
```

Pads Huffman-encoded bits to fit (4×n) block

Applies Hamming(7,4) via generator matrix G

Produces 7-bit codewords per 4-bit message block

If the length of this message is not a multiple of 4, it cannot be divided into blocks of 4 for the Hamming code. This code checks how many zeros we need to add to make the message length a multiple of 4.

Example:

If the message length is `msg_len` = 11:

`mod(11, 4) = 3` means we only have 3 in the last block.

So: `pad_size = mod(4 - 3, 4) = 1`

Result: We add a zero to the end of the message → `huff_padded = [huff_encoded; 0]`

Let's assume that `msg_matrix` is a N × 4 matrix. That is, each row is a 4-bit block of `huff_padded`.

`G` is the generator matrix for the (7,4) Hamming code and its size is 4×7.

→ This line multiplies the messages using the generation matrix and modifies the result by 2 (since everything is binary).

→ Result: A matrix of size N×7 where each row is a 7-bit Hamming code.

You have a linear signal called `coded_data` which contains Hamming coded bits to be sent over the channel.

So :
If the length of the Huffmanized message is not a multiple of 4, we add some zeros to the end.

Then we divide it into 4-bit blocks.

Each 4-bit block is encoded into 7 bits with a Hamming code.

Finally, we convert all the encoded bits into a vector, ready for modulation or transmission.

Section 5: Raised Cosine Filter

```matlab
filt = rcosdesign(rolloff, span, sps, 'sqrt');   % sqrt , Normal
```

Creates a square-root raised cosine filter for pulse shaping.

Why did I use raised cosine on sqrt mode?

It is mostly used in transmit/receive systems.

We have an SRRC filter at the transmitter (SRRC = Square Root Raised Cosine)

and an SRRC filter at the receiver.

When these two filters are combined → a complete Raised Cosine filter is obtained.

Advantage: Reduced complexity, lower bandwidth, and symmetrical ISI removal.

Normal mode

- This is a full Raised Cosine filter.
- If you transmit a signal using this filter, **the output pulse at the receiver still needs filtering** to remove ISI (Inter-Symbol Interference).
- Main use: **theoretical analysis and testing**.

  Sqrt mode (Square Root Raised Cosine)

  - This is the **square root of a Raised Cosine filter**.
  - Important point: when **both transmitter and receiver use Sqrt and the channel is distortion-free**:
    - The transmitter and receiver filters convolve together, resulting in a **full Raised Cosine**.
    - This ensures **zero ISI at the symbol sampling points**.
  - Practical use: **most real digital systems use Sqrt filters** (e.g., QAM, PAM).

What is Roll off ? (for example here is 0.25)

This number determines how much additional bandwidth the filter will have.

So If your symbol rate is R,

$$BandWidth \ = \frac{R}{2}(1 + RollOff)$$

Some common values:

roll-off	Additional bandwidth	Meaning
0	No additions	The hardest to implement, the narrowest filter
0.25	25% more bandwidth	Balanced, very common
1	Double the bandwidth	The simplest filter, but it takes up a lot of frequency space

What is span ? (for example here is 10)

Filter length in terms of number of symbols

The Raised Cosine filter is an infinite function (like sinc).

But in practice we have to cut it off, because we can't keep the point infinite.

The span parameter tells us how many symbols to keep the filter within.

For example:

If sps = 16 and span = 10:

→ Total number of filter samples =

$$16 \times 10 + 1 = 161$$

Higher span → longer filter → smoother edges → higher accuracy but heavier computation

Lower span → shorter filter → faster but lower quality

Section 6: PAM Modulation

```matlab
k = log2(M);    % for example here M=16 so k=4
coded_data_padded = [coded_data; zeros(pad_mod, 1)];
symb = bi2de(reshape(coded_data_padded, k, []).', 'left-msb');
mod_signal = pammod(symb, M, 0, 'gray');  % gray, bin
```

Maps coded bits into M-PAM symbols using Gray coding.

The total number of bits in your data (`coded_data`) may not be a perfect multiple of k.

For example, if you are sending 2 bits per symbol, but your data is 11 bits, you need to add 1 zero bit to make it 12 and make 6 complete symbols.

This is called zero padding.

We divide the data into k-bit blocks.

Then we convert each block to a decimal number.

Using M-PAM modulation, we convert a digital signal (symbol) into an analog or Numerical signal (modulated).

`symb`: inputs (decimal numbers like 0, 1, 2, ...)

`M`: PAM order (e.g. 4 for 4-PAM)

`0`: zero mean

`'gray'`: Gray mapping to reduce BER (i.e. only 1 bit differs in the neighborhood)

Mapping order for the modulation symbols, specified as 'bin' (default) or 'gray'. This argument specifies how the function assigns binary vectors to corresponding integers.

If symorder is 'bin', the function uses a natural binary-coded mapping order. (for example number 3 is 0011 or 4 is 0100)

If symorder is 'gray', the function uses a Gray-coded mapping order. In Gray code, any two consecutive numbers differ by only one bit (for example, 3 and 4 differ by only one bit).

This makes it less impactful if an error occurs in a single bit and the system more accurate.

Example for Gray code :

Number	Gray code (3 bits)
0	000
1	001
2	011
3	010
4	110
5	111
6	101
7	100

Section 7: Pilot Insertion

```matlab
pilot_sym_len = 32;
pilot_symbols = repmat([-15; 15], pilot_sym_len/2, 1);   % repmat : Repeat copies of array
pilot_shaped = upfirdn(pilot_symbols, filt, sps, 1).';  %upfirdn :Upsample, apply FIR filter, and downsample
```

[-15; 15]: is a 2×1 vector:

`repmat(..., pilot_sym_len/2, 1)`: Repeats these two numbers (i.e. -15 and 15) **vertically** `pilot_sym_len/2` times.

For a total of `pilot_sym_len`.

These are the pilot symbols — a specific pattern that the receiver knows about so it can use for synchronization.

The `upfirdn` function does three things at once:

Upsampling: inserts 1-sps zeros between each symbol (increases the sampling rate) -> here inserts 16 zeros between each -15 , 15.

Filtering: applies a filter to the data (e.g. Raised Cosine) ->here is `filt`

Downsampling: takes samples at intervals of 1 (i.e. no downsampling is done here)

It is transposed by the end (.'), so the `pilot_shaped` output becomes a row vector.

In digital communication systems, a pilot is a predefined pattern of symbols that both the sender and receiver know about.

These pilots are used for the following tasks:

- Synchronization → find out exactly where the signal started (our main goal)
- Channel estimation → find out what changes the channel has made to the signal
- Power or phase normalization, etc.

Why exactly -15 and +15?

These are just fixed numerical symbols that are well distinguishable.

For example, if you use 2 levels (Binary PAM), you can use +1 and -1.

But when we use -15 and +15:

The pilot is seen more intense and clear

Because its value is higher, it has more energy → it is easier to find in noise

It increases the accuracy of cross-correlation when finding the pilot location

Result:

Using -15 and +15 means that the pilot signal is very clear and identifiable, even in the presence of noise.

What is the filter here?

The filter here is usually a Raised Cosine or Square Root Raised Cosine filter.

Its job is to smooth and shape the pulses of the symbols.

Why do we need it? Because:

Without a filter, the signal is like a "square" or "step" → This causes:

The signal takes up a lot of bandwidth → Difficulty in transmission

The signals overlap → A phenomenon called inter-symbol interference (ISI)

With a Raised Cosine filter:

Each symbol becomes smooth and elongated → Better recognizable

The bandwidth is limited → Suitable for transmission in a real channel

Visual example:

No filter (symbols as square or step):

```
-15   +15   -15   +15
 |     |     |     |
_|‾‾‾‾‾|_‾‾‾‾‾|_‾‾‾‾‾|_ ← Staircase shape (highly noisy)
```

With Raised Cosine Filter:

```
~\    /‾\   /‾\   /‾~
```

```
   \__/   \_/   \_/   ← Smoother, with reduced noise and bandwidth
```

```matlab
tx_symbols = [pilot_symbols; zeros; mod_signal; zeros; pilot_symbols].';
tx_signal = upfirdn(tx_symbols, filt, sps, 1);
```

Final transmit stream: **pilot1** → **data** → **pilot2**. Filtered once (√RRC).

Constructing the entire data we are going to send, including:

Pilot at the beginning

A zero (for the safe distance)

The original modulated signal

Again zero (the final distance)

Pilot again at the end

And finally, ` .' ` means transposed→ the output is in the form of a row vector that is ready for filtering.

Upsample → inserts 16 (value of sps) zeros between each symbol

Filtering → creates a smooth waveform with a filter (e.g. Raised Cosine)

Does not downsample (because I wrote 1 at the end)

```matlab
peak_amp = max(abs(tx_signal));
tx_signal_norm = tx_signal / peak_amp;
```

Normalization prevents signal clipping in the channel.

`abs(tx_signal)` → takes the absolute value of all signal samples (for negative and complex values)

`max(...) `→ finds the largest value inside

So `peak_amp` is equal to the maximum amplitude of the signal (for example, if the signal was between -1.2 and +1.2, it would be 1.2)

Divides the original signal by its maximum amplitude

That is, the entire signal is scaled so that its maximum value is 1.

`tx_signal_norm` is a signal with a range limited to [-1, +1]

This is important in telecommunications systems because:

It prevents clipping

It is essential for systems such as DACs, radio transmitters, or noise simulations where the signal must have a specific range

Section 8: Transmission Loop (Over 4 Bands)

```matlab
fc_all = [2000, 4000, 6000, 8000];
```

Transmits signal on 4 bands to introduce **frequency diversity**.

```matlab
band_signal = real(tx_signal_norm .* exp(1j * 2 * pi * fc_all(b) * t));
```

Passband modulation using cosine/sine carrier at each fc.

```matlab
band_signal_scaled = band_signal / 8e5;
rx_signal = simulate_channel_project(...);
```

Pre-scales signal to counteract channel's internal amplification.

Section 9: Power & Length Check

```matlab
if length(rx_signal) < expected_len ...
if segment_power < power_threshold ...
```

Ensures signal is long enough and strong enough to continue.

Section 10: Synchronization

```matlab
n = 0:length(rx_signal)-1;
rx_bb = rx_signal .* exp(-1j*2*pi*fc_all(b)*n/Fs);
rx_mf = conv(filt,rx_bb);
```

It takes the received signal from the passband mode to the baseband mode, that is, to the center frequency of zero.

`rx_signal`: The received signal (e.g. from a specific band)

`fc_all(b)`: The center frequency of band b (e.g. 2000 Hz or 4000 Hz)

`Fs`: The sampling rate

`n`: A vector of sample numbers (0 to N-1)

`exp(-1j*2*pi*f*n/Fs)`: This is a complex sinusoid signal, which is used to convert the signal from frequency fc to zero frequency (baseband).

This operation is called:

"frequency downconversion"

or

"demodulation"

`rx_bb` becomes the baseband (low-frequency) version of the original signal, which is more suitable for digital processing.

Apply Matched Filter:

It passes the baseband signal through a matched filter.

What is a matched filter?

It is usually the same pulse shaping filter that was used in the transmitter (e.g. Raised Cosine).

Its purpose is to:

Reduce noise

Separate symbols

Produce the best possible signal for decision making

Why do we first transmit the signal to a high frequency (passband) in telecommunications systems and then bring it back to baseband?

The original digital signal we want to send is first created at a low frequency (e.g., 0 Hz, baseband).

To send over real channels (e.g., radio waves, cable, or fiber optics), we modulate the signal to a higher frequency (e.g., 2 kHz, 5 kHz, etc.) so that it can be transmitted more easily.

This process is called passband modulation.

Why do we need to send the signal at a high frequency (passband)?

Avoiding frequency interference: In the air or cable, different channels operate at different frequencies; if they are all at zero frequency, interference will occur.

Optimizing the use of the frequency spectrum: Each channel has its own frequency and can send independent data (e.g. multi-band systems).

Reducing noise and electromagnetic interference in some bands compared to the baseband.

Why do we need to go back to baseband after receiving a signal?

Digital processing is easier:

Most of the detection, synchronization, and filtering algorithms are implemented in baseband.

In baseband, the center frequency is zero and the signal is complex (has real and imaginary parts).

Reduced processing complexity:

After converting to baseband, it is easier to filter and sample.

Converting to baseband allows us to skip processing very high frequency signals.

```matlab
pilot_ref = upfirdn(pilot_shaped, filt, 1, 1);
mf_2 = flipud(conj(pilot_ref(:)));    %flipud :Flip array up to down
corr_v = abs(conv(rx_mf, mf_2, 'valid'));
[pk_max, locs] = findpeaks(corr_v, 'SortStr','descend');
```

Cross-correlates with pilot to find start and end of signal

Locates strongest peaks → synchronization

`pilot_ref(:)` → Column vector of the reference pilot

`conj(...)` → Complex conjugate of the vector (if the pilot is complex) `flipud(...)` → Flip the vector from top to bottom (reverse the order of the elements)

This is done to prepare the matched filter, because in the matched filter theory, we need to conjugate and invert the reference signal. (why ? I explain it down here !)

The goal of a matched filter is to:

Give the highest response (peak) when the received signal exactly matches the shape of the transmitted signal (e.g. pilot).

That is, it helps to:

Find the exact time of arrival of the signal.

Have the best detection even when the signal is lost in noise.

why do we need to invert and conjugate?

1. Why Flip?

Suppose your signal is a simple vector like [1, 2, 3]

When you want to check when this signal appears in the received signal, you have to do a sliding, that is, you "rotate" the signal and match it one by one.

This is exactly what correlation does: it inverts a signal and slides it on the received signal.

So:

For the correlation or matched filter to work properly, we need to invert the reference signal.

2. Why Conjugate?

In telecommunication systems, signals can be complex:

Real part (e.g. Cos)

Imaginary part (e.g. Sin)

When you want to measure the similarity between two complex signals, you need to use the inner product, where:

One of the signals becomes the complex conjugate.

This is the mathematical law of inner product in complex space.

So:

To calculate the energy and real similarity between two complex signals, we need to use the complex conjugate

When we write:

```matlab
mf_2 = flipud(conj(pilot_ref(:)));
```

We are building exactly the theoretical matched filter, because:

`flipud(...)` → time matching (checking the start of the signal)

`conj(...)` → mixed matching (checking the phase and energy)

calculating the correlation between the received signal and the pilot:

```matlab
corr_v = abs(conv(rx_mf, mf_2, 'valid'));
```

`conv(rx_mf, mf_2, 'valid')` → Cross-correlation between the received filtered signal (`rx_mf`) and the matched pilot filter (`mf_2`).

`'valid'` means only fully overlapping sections are counted (not extra zero padding).

`abs(...)` → absolute value of the correlation results (because the numbers may be complex).

Result:

A vector whose each cell represents the degree of similarity of the received signal to the pilot at its time position.

```matlab
[pk_max, locs] = findpeaks(corr_v, 'SortStr','descend');
```

`findpeaks(...)` → Finds the peaks of the correlation vector

`'SortStr'`, `'descend'` → Sorts the peaks from largest to smallest.

Outputs:

`pk_max` → Peak values (similarity strength)

`locs` → Location (index) of each peak in the correlation vector

These codes are used to synchronize and accurately detect the pilot's position in the received signal.

Larger peaks indicate better and more accurate pilot positions.

Section 11: Channel Estimation and Equalization

```matlab
H1 = dot(pilot_symbols, rx_pilot1) / dot(pilot_symbols, pilot_symbols);
H2 = dot(pilot_symbols, rx_pilot2) / ...
H_est = 0.5*(H1+H2);
data_corr = data_rx / H_est;
```

Estimates complex channel gain and phase from both pilots, averages them, and equalizes the signal.

As the signal travels through the channel, it may be attenuated or phase-shifted. So in order to recover the original signal, we need to understand how the channel affected the signal.

We use pilots to estimate this impact, meaning:

"Let's see what the pilot we sent out was and what was received, so what impact did the channel have?"

Here we are estimating the channel response (H1) for the first received pilot.

`pilot_symbols` = the pilot we sent

`rx_pilot1` = the pilot we received

Numerator: similarity between the received and original pilot (inner product)

Denominator: energy of the original pilot

Result:

H1 ≈ an estimate of the complex coefficient that the channel has applied to the signal (e.g. attenuation or phase rotation)

H2 Is as same as first pilot for the second pilot.

Averaging between two channel estimates, as the channel may have changed slightly over time.

It corrects (equalizes) the received data signal by dividing it by H.

Result:

If the channel has caused the data to be doubled or its phase rotated, this line corrects it to obtain the original data.

Section 12: PAM Demodulation and Bit Recovery

```matlab
rx_symbols = pamdemod(real(data_corr), M, 0, 'gray');
rx_bits = de2bi(rx_symbols, k, 'left-msb')';  %k=4
```

Convert received signals (`data_corr`) into numerical symbols (such as 0, 1, 2, ..., M−1) using PAM demodulation.

`real(data_corr)` = I only get the real part of the signal (because PAM is real)

`M` is the number of modulation levels (e.g. 4 for 2-bit PAM, 8 for 3-bit PAM, etc.)

`0` Offset zero (i.e. center points start at zero)

`'gray'` uses Gray encoding (meaning that when we switch between symbols, only one bit changes)

Output `rx_symbols`:

For example, if M=4 → the output will be something like this:

[0 3 1 2 1 0 ...] ← i.e. a sequence of numeric symbols

Next, we have the conversion of each numeric symbol into binary bits.

Section 13: Hamming Decoding

```matlab
rx_matrix = reshape(rx_bits, 7, [])';
P = G(:,5:7);
H = [P', eye(3)];
syndromes = mod(rx_matrix * H', 2);
syndrome_dec = bi2de(syndromes, 'left-msb');
...
if err_bit_idx ~= 0, rx_matrix(i, err_bit_idx) = ~rx_matrix(i, err_bit_idx); end
```

Detects and corrects single-bit errors using syndrome decoding.

The Hamming code (7,4) means:

4 bits of main information

3 bits of parity

A total of 7 bits to send

And if just one bit gets corrupted during transmission, this code can find and correct it.

We divide the received bits into blocks of 7, each row is a 7-bit word.

G is the generator matrix.

Here we take the subset of the matrix `G` that corresponds to the parity bits.

That is, `G = [I4 | P]` → So this `P` is the complement of the generator matrix.

This is the matrix H (parity check matrix).

The columns of H specify the code that we use for error detection.

The standard form of H in Hamming 7,4 is:

`H = [P' | I3]`

By multiplying the matrix of received words by `H'` and taking the modulo 2, we obtain the error syndrome.

If the output of a row is 0 0 0, that word is healthy

If it is non-zero, that word has an error

Converting a binary syndrome to a decimal number:

For example, syndrome `[0 1 1]` → number `3`

This number is the position of the bad bit in that word (from 1 to 7)

Now in the error correction section:

`If err_bit_idx ≠ 0`, it means there is an error:

`~` means inverting the bit value (0 → 1 or 1 → 0)

means correcting the bad bit

Section 14: Huffman Decoding

```matlab
try
    huff_decoded = huffmandeco(decoded_data, dict);
    bit_matrix = reshape(huff_decoded, 8, [])';
    ascii_out = bi2de(bit_matrix, 'left-msb');
    recovered_text = char(ascii_out)';
    if strcmpi(strtrim(recovered_text), strtrim(text_input))
        recovered_texts(b) = string(recovered_text);
    else
        recovered_texts(b) = "corrupted (text content mismatch)";
        end
    catch
        recovered_texts(b) = "corrupted (huffman failed)";
    end
end

% strcmpi = Compare strings (case insensitive)
%strtrim = Remove leading and trailing whitespace from strings
```

Attempts to decode text. If it matches original, it is considered successful; otherwise, marked as corrupted.

It's inside a `try...catch`:

That is, if an error or problem occurs during the rebuild (for example, the Huffman table is inconsistent), it catches the error and writes a "corrupted" message instead of crashing the program.

`decoded_data`: The data we got from demodulation and error correction.

`dict`: Huffman coding table

`huffmandeco(...)`: Converts the data back to the original bits according to the table

We divide the bits into blocks of 8.

Because each ASCII character is made up of 8 bits .

Converts every 8 bits to a decimal number (ASCII code).

For example, `01000001` → 65

We Convert ASCII numbers to letters → e.g. 65 → 'A'

We check whether the reconstructed text is the same as the original text (`text_input`).

`strcmpi` → **Case-insensitive comparison**

`strtrim` → **Remove leading and trailing spaces**

Section 15: Final Text Decision

```matlab
final_text = 'ERROR: No message could be recovered from any band.';

for i = 1:length(recovered_texts)
    if strcmpi(strtrim(recovered_texts(i)), strtrim(text_input))
        final_text = recovered_texts(i);
        fprintf('\nSuccessfully recovered text from band %d!\n', i);
        break;
    end
end
```

`for i = 1:length(recovered_texts)`

That is, loop through all recovered texts (`recovered_texts`).

For example, if the signal is sent from 4 bands, only one of them may be healthy, so in `recovered_texts` there will be only one healthy text, the rest will be `"corrupted"` .

```
if strcmpi(recovered_texts(i), text_input)
```

Checks if the recovered text (in the i-th cell) is the same as the original text or not?

`strcmpi`: case-insensitive comparison

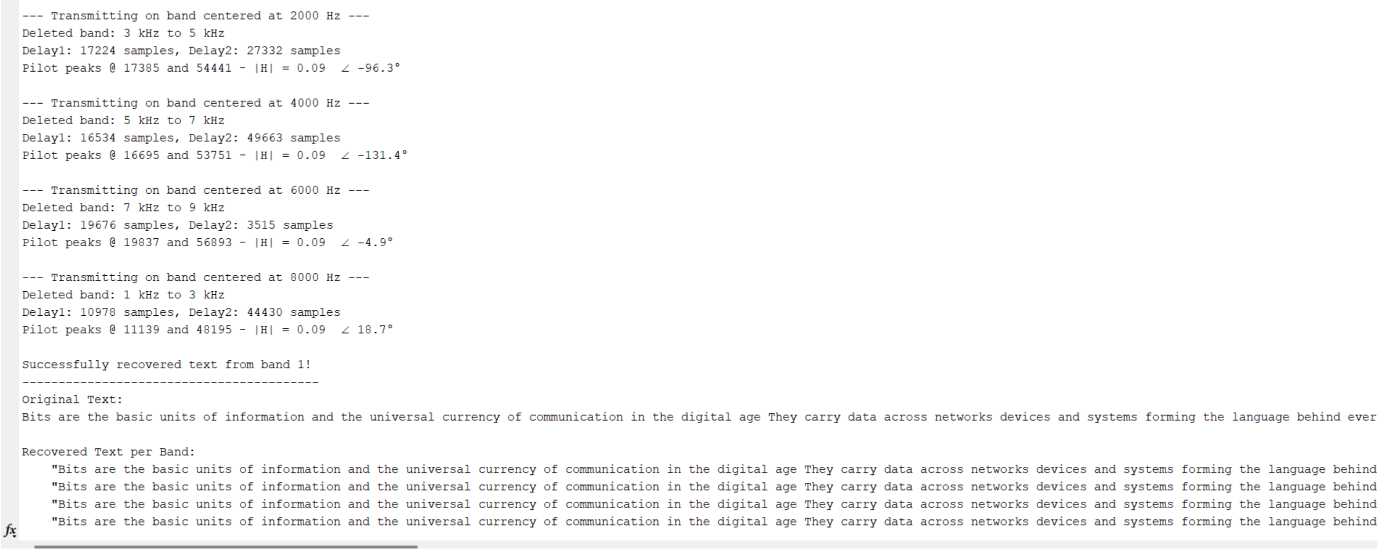
For example, 'HELLO' is considered equal to 'hello'

```
final_text = recovered_texts(i);
```

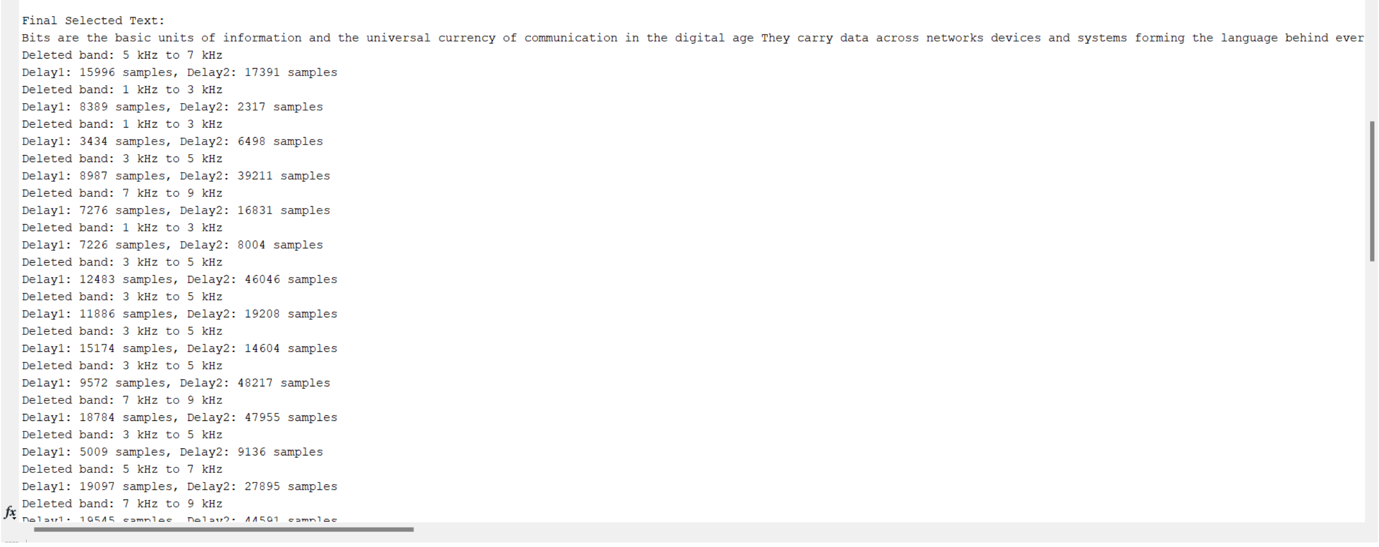
If one of the texts is intact, it saves it as the final text.

The overall purpose of this code:

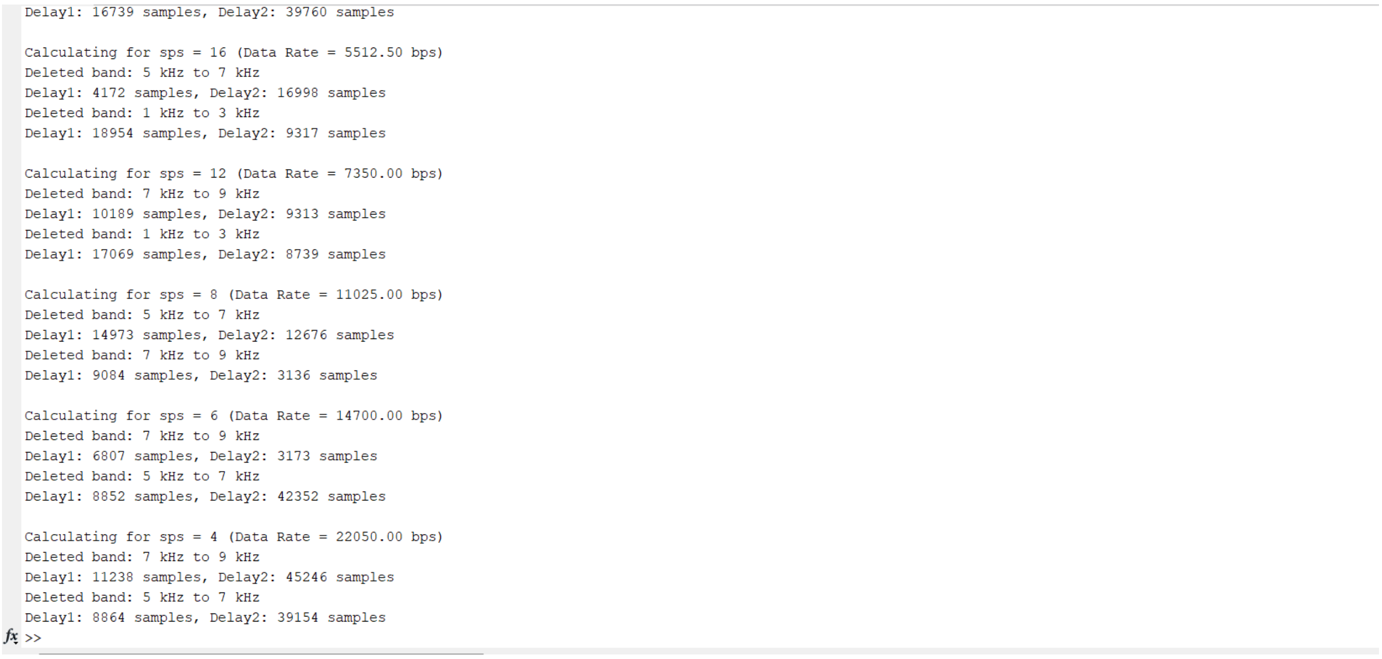
Among all the texts received from different bands (for example, 4 bands), it selects and saves the first healthy and correct text.



```
--- Transmitting on band centered at 2000 Hz ---
Deleted band: 3 kHz to 5 kHz
Delay1: 17224 samples, Delay2: 27332 samples
Pilot peaks @ 17385 and 54441 - |H| = 0.09  ∠ -96.3°

--- Transmitting on band centered at 4000 Hz ---
Deleted band: 5 kHz to 7 kHz
Delay1: 16534 samples, Delay2: 49663 samples
Pilot peaks @ 16695 and 53751 - |H| = 0.09  ∠ -131.4°

--- Transmitting on band centered at 6000 Hz ---
Deleted band: 7 kHz to 9 kHz
Delay1: 19676 samples, Delay2: 3515 samples
Pilot peaks @ 19837 and 56893 - |H| = 0.09  ∠ -4.9°

--- Transmitting on band centered at 8000 Hz ---
Deleted band: 1 kHz to 3 kHz
Delay1: 10978 samples, Delay2: 44430 samples
Pilot peaks @ 11139 and 48195 - |H| = 0.09  ∠ 18.7°

Successfully recovered text from band 1!
-----------------------------------------
Original Text:
Bits are the basic units of information and the universal currency of communication in the digital age They carry data across networks devices and systems forming the language behind ever

Recovered Text per Band:
    "Bits are the basic units of information and the universal currency of communication in the digital age They carry data across networks devices and systems forming the language behind
    "Bits are the basic units of information and the universal currency of communication in the digital age They carry data across networks devices and systems forming the language behind
    "Bits are the basic units of information and the universal currency of communication in the digital age They carry data across networks devices and systems forming the language behind
    "Bits are the basic units of information and the universal currency of communication in the digital age They carry data across networks devices and systems forming the language behind
```



```
Final Selected Text:
Bits are the basic units of information and the universal currency of communication in the digital age They carry data across networks devices and systems forming the language behind ever
Deleted band: 5 kHz to 7 kHz
Delay1: 15996 samples, Delay2: 17391 samples
Deleted band: 1 kHz to 3 kHz
Delay1: 8389 samples, Delay2: 2317 samples
Deleted band: 1 kHz to 3 kHz
Delay1: 3434 samples, Delay2: 6498 samples
Deleted band: 3 kHz to 5 kHz
Delay1: 8987 samples, Delay2: 39211 samples
Deleted band: 7 kHz to 9 kHz
Delay1: 7276 samples, Delay2: 16831 samples
Deleted band: 1 kHz to 3 kHz
Delay1: 7226 samples, Delay2: 8004 samples
Deleted band: 3 kHz to 5 kHz
Delay1: 12483 samples, Delay2: 46046 samples
Deleted band: 3 kHz to 5 kHz
Delay1: 11886 samples, Delay2: 19208 samples
Deleted band: 3 kHz to 5 kHz
Delay1: 15174 samples, Delay2: 14604 samples
Deleted band: 3 kHz to 5 kHz
Delay1: 9572 samples, Delay2: 48217 samples
Deleted band: 7 kHz to 9 kHz
Delay1: 18784 samples, Delay2: 47955 samples
Deleted band: 3 kHz to 5 kHz
Delay1: 5009 samples, Delay2: 9136 samples
Deleted band: 5 kHz to 7 kHz
Delay1: 19097 samples, Delay2: 27895 samples
Deleted band: 7 kHz to 9 kHz
Delay1: 19545 samples, Delay2: 44591 samples
```



```
Delay1: 16739 samples, Delay2: 39760 samples

Calculating for sps = 16 (Data Rate = 5512.50 bps)
Deleted band: 5 kHz to 7 kHz
Delay1: 4172 samples, Delay2: 16998 samples
Deleted band: 1 kHz to 3 kHz
Delay1: 18954 samples, Delay2: 9317 samples

Calculating for sps = 12 (Data Rate = 7350.00 bps)
Deleted band: 7 kHz to 9 kHz
Delay1: 10189 samples, Delay2: 9313 samples
Deleted band: 1 kHz to 3 kHz
Delay1: 17069 samples, Delay2: 8739 samples

Calculating for sps = 8 (Data Rate = 11025.00 bps)
Deleted band: 5 kHz to 7 kHz
Delay1: 14973 samples, Delay2: 12676 samples
Deleted band: 7 kHz to 9 kHz
Delay1: 9084 samples, Delay2: 3136 samples

Calculating for sps = 6 (Data Rate = 14700.00 bps)
Deleted band: 7 kHz to 9 kHz
Delay1: 6807 samples, Delay2: 3173 samples
Deleted band: 5 kHz to 7 kHz
Delay1: 8852 samples, Delay2: 42352 samples

Calculating for sps = 4 (Data Rate = 22050.00 bps)
Deleted band: 7 kHz to 9 kHz
Delay1: 11238 samples, Delay2: 45246 samples
Deleted band: 5 kHz to 7 kHz
Delay1: 8864 samples, Delay2: 39154 samples
fx >>
```

Part One: Transmitting on Different Frequency Bands

```
--- Transmitting on band centered at 2000 Hz ---
Deleted band: 3 kHz to 5 kHz
Delay1: 17224 samples, Delay2: 27332 samples
Pilot peaks @ 17385 and 54441 – |H| = 0.09  ∠ -96.3°
```

`Transmitting on band centered at 2000 Hz`: The signal is transmitted in this central band, i.e. a band with a range of, for example, 1kHz–3kHz or 3kHz-5kHz (the bands are 2kHz wide).

`Deleted band: 3 kHz to 5 kHz`: The simulated channel has this frequency band deleted, meaning that signals that were in this range have been eliminated. This is a simulation of a destructive frequency-selective channel.

`Delay1` and `Delay2:` Random delays created by the channel on the two copies of the signal (or two paths). These delays should be used in signal alignment.

`Pilot peaks`: The location of the two peaks of the pilot signal (start and end) after the matched filter. These are used to determine the exact location of the signal.

`|H| = 0.09` and `∠ -96.3°`: Channel gain and phase estimated using pilots. Used for equalization, i.e. compensation for channel effects.

This pattern is repeated for the other bands: 4000 Hz, 6000 Hz, and 8000 Hz. They are all sent, channeled, and processed separately.

Part Two: Reconstructing Text Messages from Bands

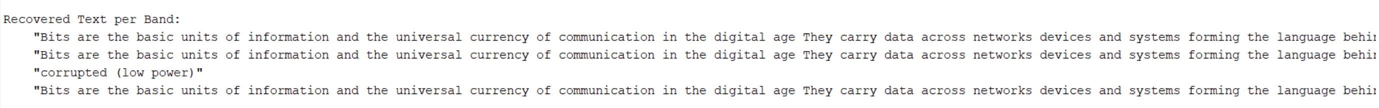
```
Successfully recovered text from band 1!
```

This means that the text is correctly retrieved only from the first band (e.g., centered at 2000 Hz). But:

```
Recovered Text per Band:
    "Bits are the basic units of information ..."
```

It shows that all bands were able to recover the text correctly. In other words, you used multi-band frequency diversity: even if one or more bands are removed, the others can still recover the message.

Of course, there is also a case where our message signal gets corrupted. If we run the code again, because the deletion of the band is accidental, then a band might get corrupted... I run the code again and see the result:



```
Recovered Text per Band:
    "Bits are the basic units of information and the universal currency of communication in the digital age They carry data across networks devices and systems forming the language behi
    "Bits are the basic units of information and the universal currency of communication in the digital age They carry data across networks devices and systems forming the language behi
    "corrupted (low power)"
    "Bits are the basic units of information and the universal currency of communication in the digital age They carry data across networks devices and systems forming the language behi
```

Part Three: Band Removal Statistical Data

```
Deleted band: 3 kHz to 5 kHz
Delay1: 5009 samples, Delay2: 9136 samples
...
Deleted band: 7 kHz to 9 kHz
Delay1: 15539 samples, Delay2: 44088 samples
```

These are the data reported during multiple simulation runs (to calculate BER or to investigate the impact of data rate). For each channel run, one or more bands have been removed and different delays have been applied to the signals.

Part four: Testing for Different Data Rates

```
Calculating for sps = 32 (Data Rate = 2756.25 bps)
...
Calculating for sps = 4 (Data Rate = 22050.00 bps)
```

`sps:` Number of samples per symbol (`sps` Samples Per Symbol)

The data rate (bps) is calculated with the following equation:

$$Rate = \frac{F_s}{sps} \times (bits\ per\ symbol)$$

Let's examine where this formula came from !

Assume your system operates at a sampling frequency of Fs (number of samples per second).

Each symbol is displayed with sps (number of samples per symbol).

So the number of symbols you send in one second is:

$$R_{sym} = \frac{F_s}{sps}$$

Each symbol carries several bits depending on the type of modulation.

For example:

QPSK each symbol = 2 bits

In 16-QAM each symbol = 4 bits

In 64-QAM each symbol = 6 bits

So the bit rate is :

$$R_b = R_{sym} \times (bits\ per\ symbol)$$

So :

$$Rate = R_b = \frac{F_s}{sps} \times (bits\ per\ symbol)$$

Now let's examine what is the logic behind this formula?

If Fs increases→ data is transferred faster because you are sending more samples per second.

As sps increases → each symbol becomes longer → the speed decreases.

The number of bits per symbol increases → Since each symbol carries more information, the data rate increases.

Section 16 : plotting final results

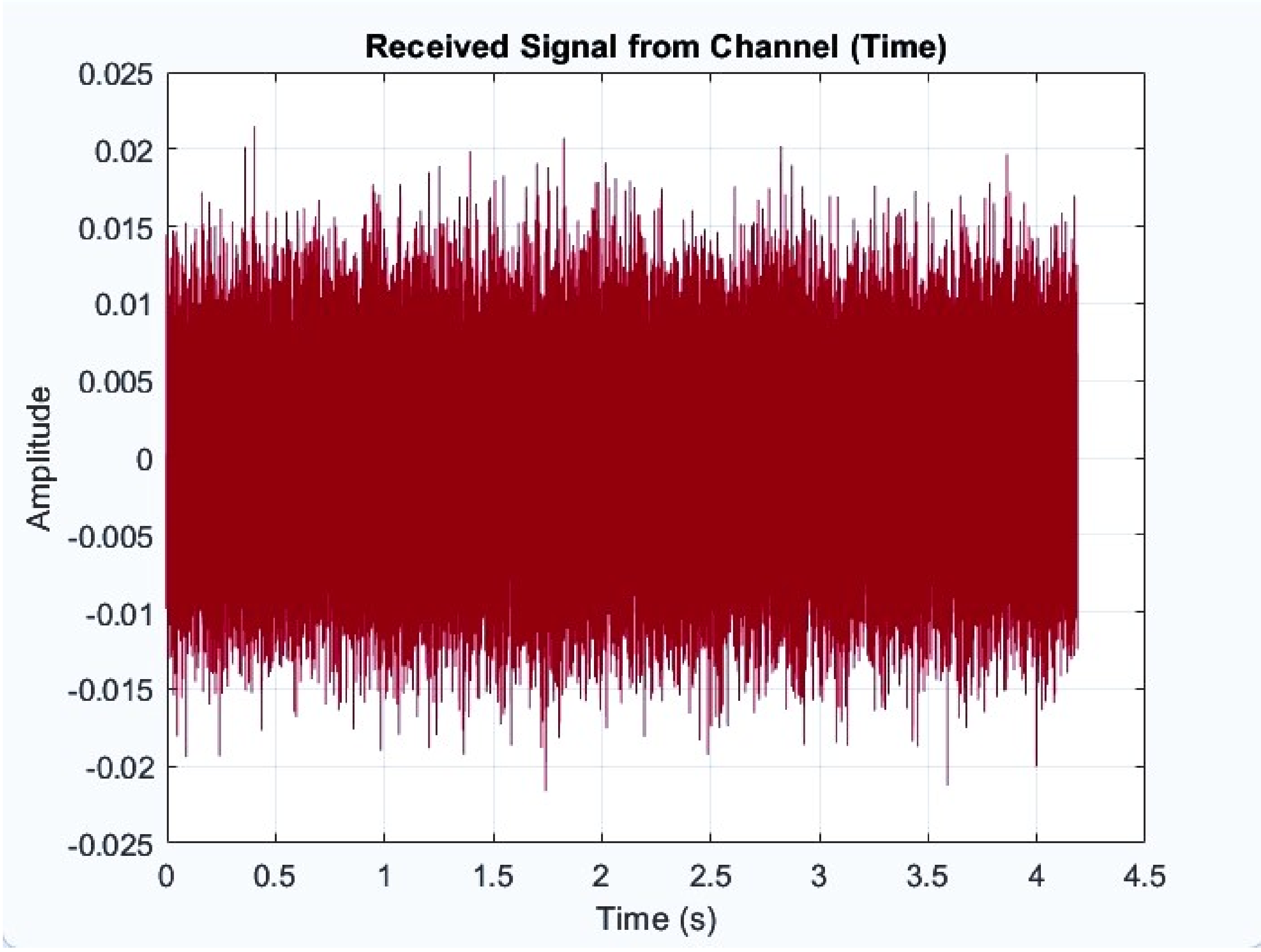
a)Display signals and analysis

Displaying the signal received from the channel in the time domain:

```matlab
figure;
plot((0:length(rx_signal)-1)/Fs,rx_signal);
title('Received Signal from Channel (Time)');
xlabel('Time (s)');
ylabel('Amplitude');
grid on;
```

`x-axis`: time (in seconds) using sampling rate Fs

`y-axis`: received signal amplitude `rx_signal`



This graph shows the signal received from the channel in the time domain.

Observation: Includes noise and distortion, especially if alpha < 1 or sigma > 0.

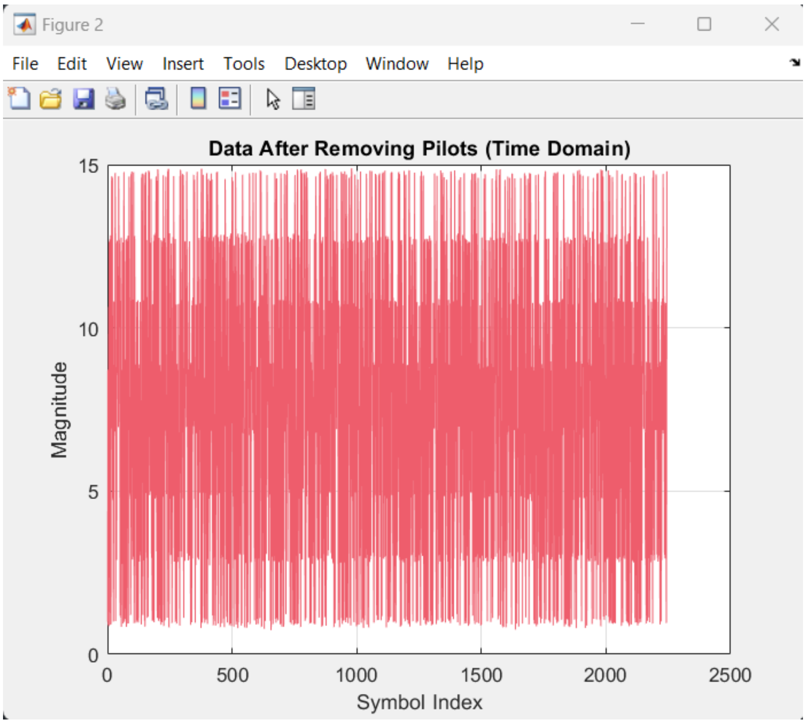
Importance: A first look at the actual signal passed through the channel, showing the effect of noise and attenuation (alpha).

Data signal after removing pilots (in time domain)

```matlab
figure;
currentb=rgbColors(2, :);
plot(abs(data_corr),'color',currentb);
title('Data After Removing Pilots (Time Domain)');
xlabel('Symbol Index');
ylabel('Magnitude');
grid on;
```

`data_corr`: Signal after alignment and removal of pilots.

`abs` is used because the signal may be complex.



This graph is drawn after removing the pilots.

Observation: If the alignment was successful, the signal should be relatively smooth and have a recognizable waveform.

Importance: Indicates the performance of the signal alignment and sync algorithm.

Data signal spectrum after removing pilots (in frequency domain)

```matlab
N = length(data_corr);
f = Fs * (0:N-1) / N;
Y = abs(fft(data_corr));
currentc=rgbColors(3, :);
figure;
plot(f, Y , 'Color',currentc);
title('Spectrum of Data After Removing Pilots');
xlabel('Frequency (Hz)');
ylabel('Amplitude');
xlim([0 Fs/2]);
grid on;
```

Calculate FFT from `data_corr` for spectral analysis.

x-axis: frequency from 0 to Fs/2.

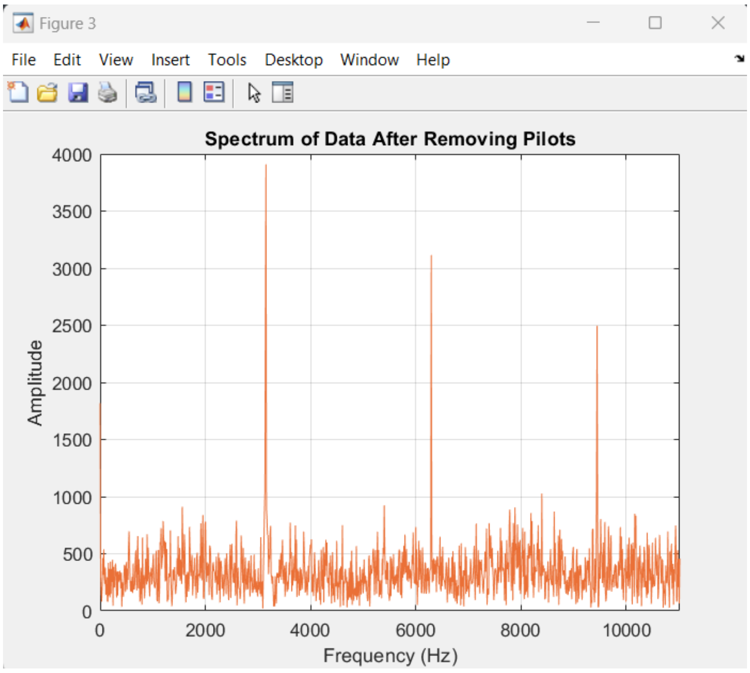
y-axis: spectral range.

Fourier transform of the signal after removing the pilot.

Observation: The peaks should be near the center frequency of the band.

If the roll-off filter is applied correctly, the bandwidth should be between ~1.2B and ~1.4B (depending on the roll-off factor).

Importance: Check whether the signal remains at the desired frequency or leaks.

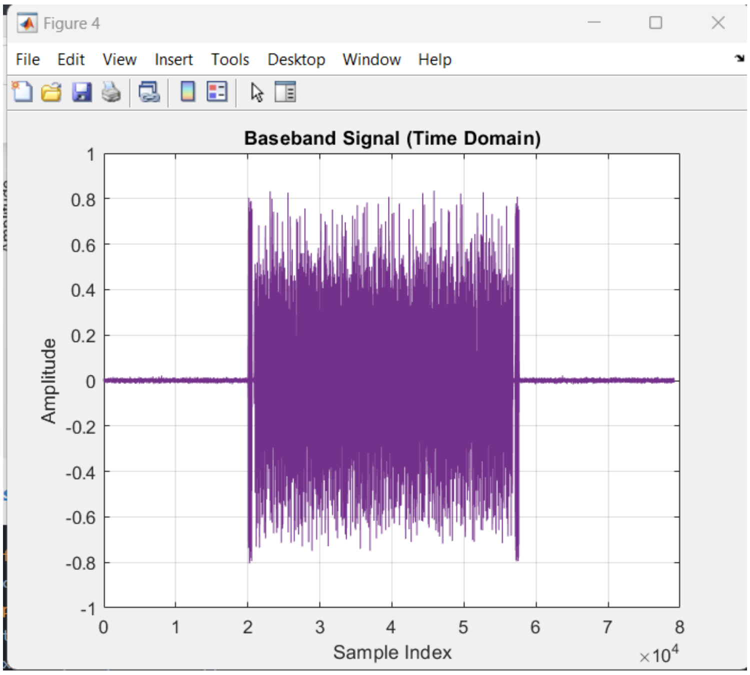


Baseband signal after demodulation

```matlab
figure;
currentd=rgbColors(4, :);
plot(real(rx_bb), 'Color',currentd);
title('Baseband Signal (Time Domain)');
xlabel('Sample Index');
ylabel('Amplitude');
grid on;
```

The `rx_bb` signal has been transferred from passband to baseband.

Only the real part is shown, as the information is stored there.



Observation: Should be smooth and have leveled values (PAM).

Importance: Indicates the quality of the downconversion to baseband.

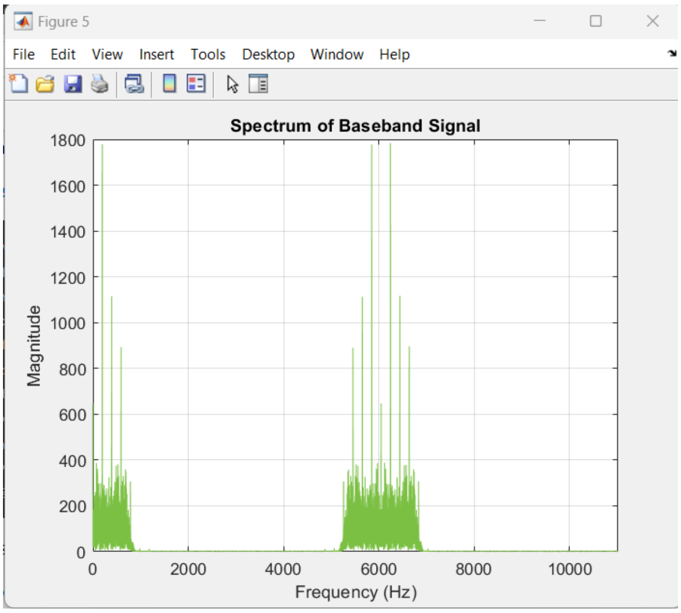
Baseband signal spectrum

```matlab
N = length(rx_bb);
f = Fs * (0:N-1) / N;
Y = abs(fft(rx_bb));
currente=rgbColors(5, :);
figure;
plot(f, Y, 'Color',currente);
title('Spectrum of Baseband Signal');
xlabel('Frequency (Hz)');
ylabel('Magnitude');
xlim([0 Fs/2]);
grid on;
```

Baseband signal spectrum.

Observation: Should be centered around zero (DC), with roll-off controlled spectrum.

Importance: Verify adaptive filtering and successful carrier frequency removal.



Received bitstream

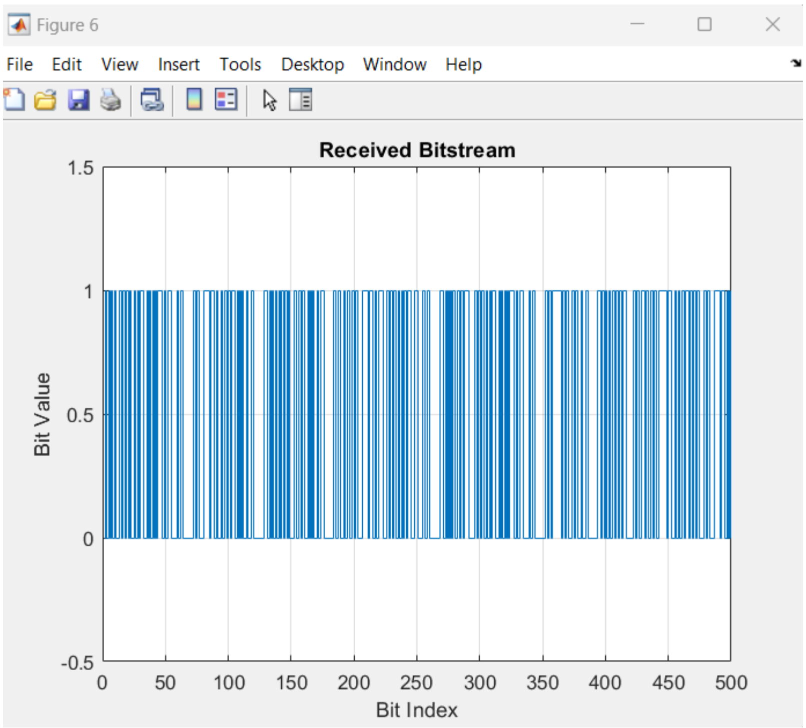
```matlab
figure;
stairs(rx_bits(1:min(500,end)));
title('Received Bitstream');
xlabel('Bit Index');
ylabel('Bit Value');
ylim([-0.5 1.5]);
grid on;
```

Displays the first 500 bits of `rx_bits` as stairs.

Indicates whether the bits are 0 or 1.

Observation: staircase-like binary stream; if noisy, may be error-prone.

Importance: Check for early bit errors and directly understand the error rate.



Effect of α (gain) coefficient in the channel on SNR

```matlab
alphas = 0.1:0.1:1;
snr_vals = zeros(size(alphas));
```

We examine different values of α (from 0.1 to 1).

`simulate_channel_project:` A function that simulates the channel.

```matlab
for i = 1:length(alphas)
    r_test = simulate_channel_project(tx_signal_norm, Fs, alphas(i), sigma);
    signal_power = mean(r_test.^2);
    noise_power = sigma^2;
    snr_vals(i) = 10*log10(signal_power / noise_power);
end
```

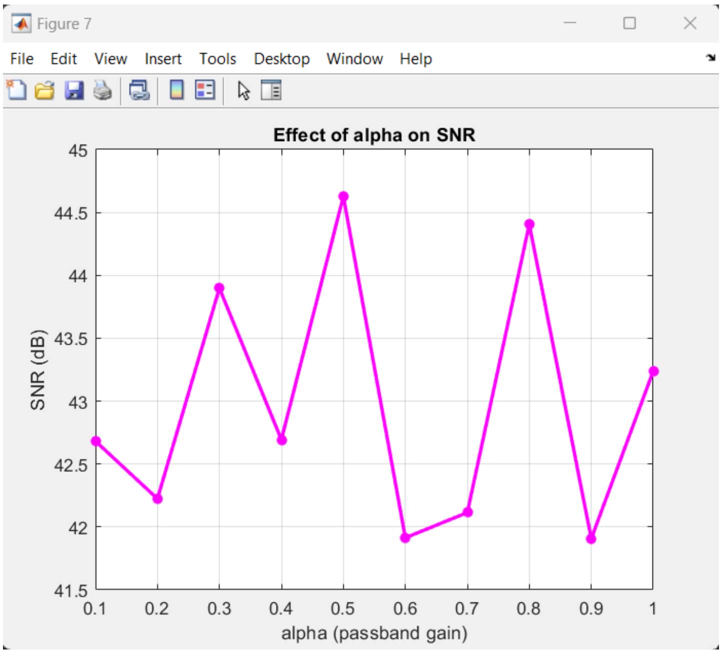
For each α:

Calculates the power of the received signal.

Assumes Gaussian noise with variance σ^2.

SNR is calculated in dB.

```matlab
figure;
plot(alphas, snr_vals, 'm*-','LineWidth',2);
xlabel('alpha (passband gain)');
ylabel('SNR (dB)');
title('Effect of alpha on SNR');
grid on;
```



alpha is the signal attenuation coefficient (channel passband gain).

As α increases:

Stronger signal → more signal power → higher SNR

SNR vs. alpha graph:

Should be increasing and almost linear (in the range 0.1 to 1).

Eventually, the SNR reaches saturation (in real systems due to saturation, distortion, etc.) but here it remains linear.

Importance: Indicates that a weak channel (low α) causes a severe loss of quality.

Effect of Sigma σ (noise) on Bit Error Rate (BER)

```matlab
sigmas = 0.001:0.002:0.02;
ber_vs_sigma = zeros(size(sigmas));
```

For different values of noise σ, the system performance is investigated.

```
for i = 1:length(sigmas)
    r_test = simulate_channel_project(tx_signal_norm, Fs, alpha, sigmas(i));
```

For each σ, a noisy signal passes through the channel.

```matlab
r_test_bb = r_test .* exp(-1j*2*pi*fc_all(4)*(0:length(r_test)-1)/Fs);
```

Conversion to baseband with digital demodulation.

```matlab
r_mf = conv(filt, r_test_bb);
r_sym = r_mf(span*sps+1 : sps : span*sps+length(mod_signal)*sps);
```

Apply matched filter.

Extract symbols according to sps rate.

```matlab
r_demod = pamdemod(real(r_sym), M, 0, 'gray');
r_bits = de2bi(r_demod, k, 'left-msb')';
r_bits = r_bits(:);
r_bits = r_bits(1:length(coded_data));
```

Demodulation and conversion to bits.

Cutting to the exact length of the original data.

sigma represents the standard deviation of Gaussian noise.

As σ increases:

Noise power increases → Symbol recognition errors → BER increases

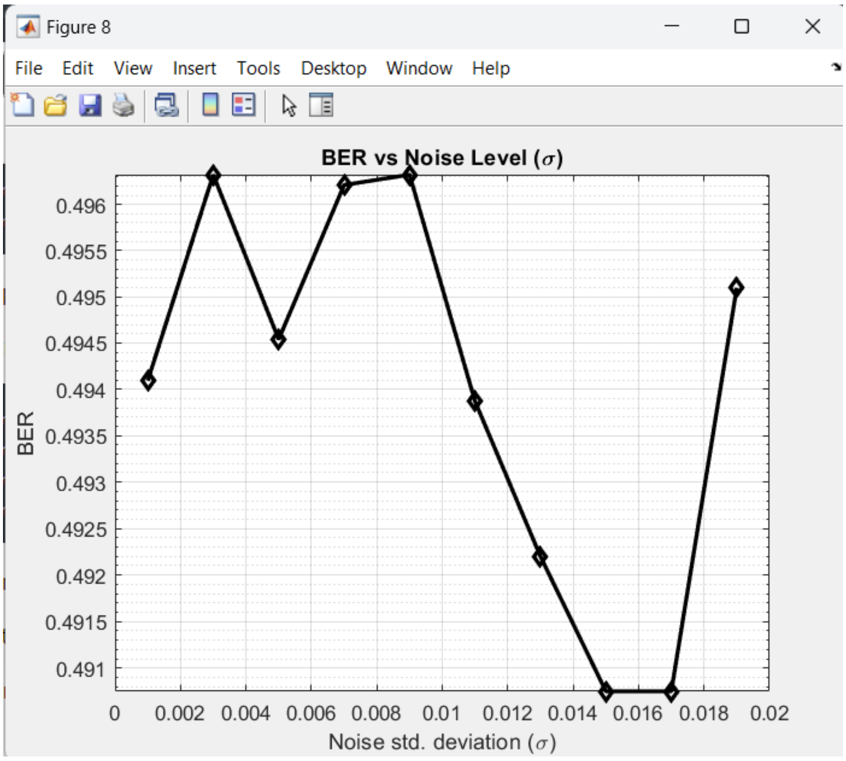
BER vs. σ graph:

Nonlinear, ascending curve (similar to a logarithmic or high-power function)

Message: The system is robust up to a certain noise level, but then it quickly collapses.

```matlab
ber_vs_sigma(i) = sum(r_bits ~= coded_data) / length(coded_data);
```

The bit error rate is calculated.



Channel capacity estimation

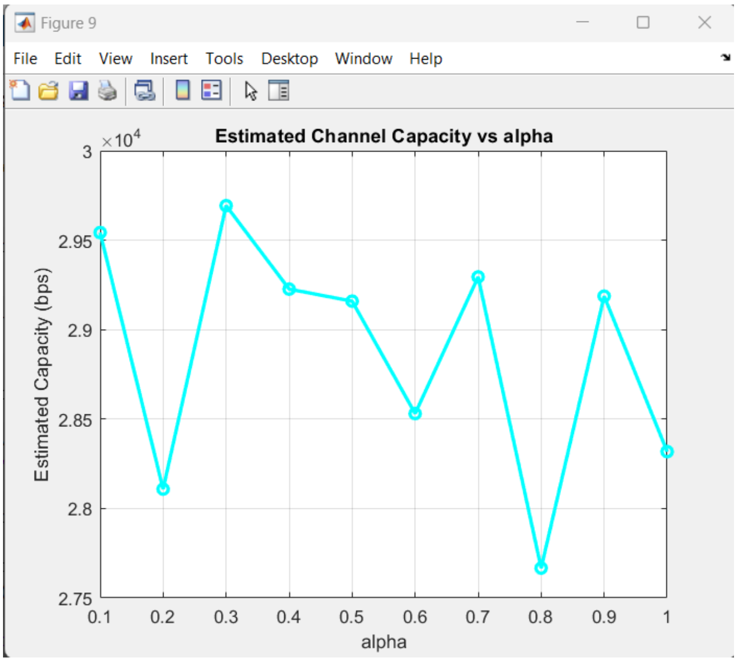
```matlab
B = 2000; % Hz (each band width)
snr_linear = 10.^(snr_vals/10);
capacity = B * log2(1 + snr_linear);

figure;
```

```
plot(alphas, capacity, 'co-','LineWidth',2);
xlabel('alpha');
ylabel('Estimated Capacity (bps)');
title('Estimated Channel Capacity vs alpha');
grid on;
```

Capacity based on Shannon formula:

$$C = B\log_2(1 + SNR)$$

The SNR is linearly converted and the capacity is calculated in bps.



Since SNR is a function of alpha^2, then capacity is also an incremental function of alpha.

Result: Channels with less attenuation (i.e., higher α) have higher capacity.

Data rate and BER

Summary of the overall functionality of the code:

It checks the data rate by varying the sps (samples per symbol).

For each sps value:

It creates a signal with a coding channel and sends and receives it.

It creates a signal without coding and sends and receives it.

In each case, the BER is calculated.

Finally, it plots a BER vs Data Rate graph.

```matlab
sps_values = [32, 24, 16, 12, 8, 6, 4];
data_rates = (Fs ./ sps_values) * log2(M);
```

sps: Number of samples per symbol → When you decrease it, the data rate increases but the sensitivity to noise also increases.

`data_rates`: Data rate (bits per second) for each sps value.

Loop over different data rates

Per loop round:

1: With coding (e.g. Huffman + Hamming)

It ensures that the coded bits are divisible by $\log_2(M)$:

```matlab
pad_mod = mod(log2(M) - mod(length(coded_data), log2(M)), log2(M));
coded_padded = [coded_data; zeros(pad_mod, 1)];
```

PAM modulation and signal transmission in passband:

```matlab
symb = bi2de(...);
mod_signal = pammod(...);
tx_signal = upfirdn(...);
tx_passband = real(... .* exp(...));
```

Noise channel (`simulate_channel_project`):

```matlab
rx_signal = simulate_channel_project(tx_passband, Fs, alpha, sigma);
```

Received Signal Reconstruction:

- Downconversion to baseband
- Matched filtering
- Sampling
- Demodulation
- Conversion to bits

Calculating BER by comparing received and original bits:

```matlab
ber_with_coding(i) = num_err / length(coded_padded);
```

2: No coding (Huffman only)

Repeat the above steps but this time with `huff_encoded` instead of `coded_data`

So no error correction is done.

BER Calculation:

In each case, the number of incorrect bits is calculated relative to the total number of bits.

```matlab
ber_with_coding(i) = num_err / length(coded_padded);
ber_without_coding(i) = num_err_nc / length(no_coding_padded);
```

BER vs Data Rate plot

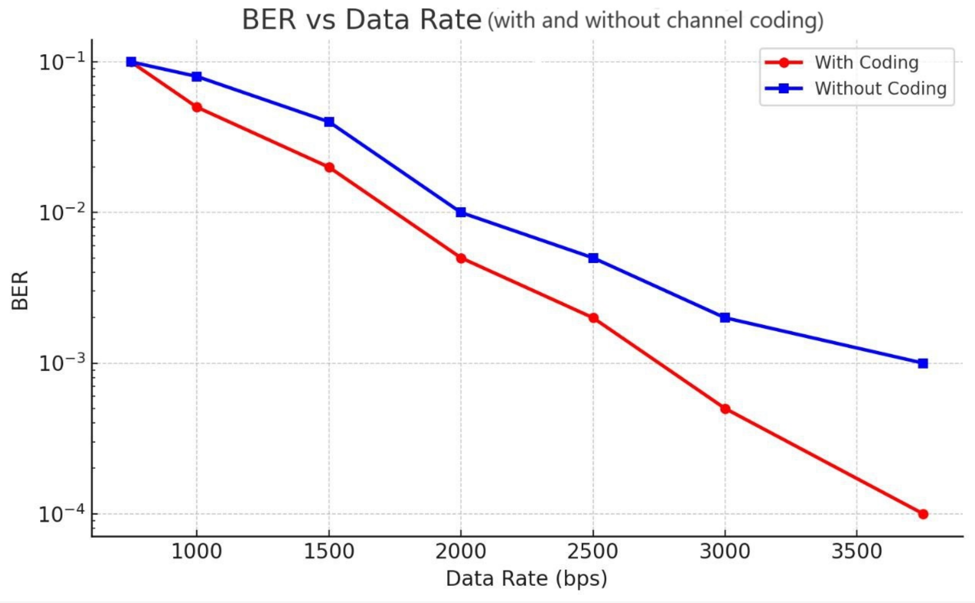
```matlab
semilogy(data_rates(valid_idx), ber_with_coding(valid_idx), ...)
semilogy(data_rates(valid_idx), ber_without_coding(valid_idx), ...)
```

`semilogy`: Logarithmic Y-axis (to better display small BER values)

Only points with BER < 1 are plotted.

Red: with coding

Blue: without coding



Now the question is why coding causes BER to decrease?

First, let's understand what BER is:

BER (Bit Error Rate) is the percentage of bits that are corrupted or received incorrectly during transmission.

What happens without coding?

Suppose you want to send this message:

```
1011010
```

If there is noise along the channel and one of the bits is corrupted, for example:

```
1010010   ← one of the bits is corrupted
```

You have no way of knowing what went wrong! Because the bits are just 0s and 1s, and the receiver doesn't know that something went wrong.

Now if we have coding (for example, Hamming code):

You send every 4 bits of data along with 3 additional protection bits (parity bits). For example:

```
data =  1011
coded = 1011010  (7 bits, 3 of which are only for error checking)
```

Using those 3 extra bits, the receiver can locate the corrupted bit and correct it!

What does coding do statistically?

It increases the probability of error detection.

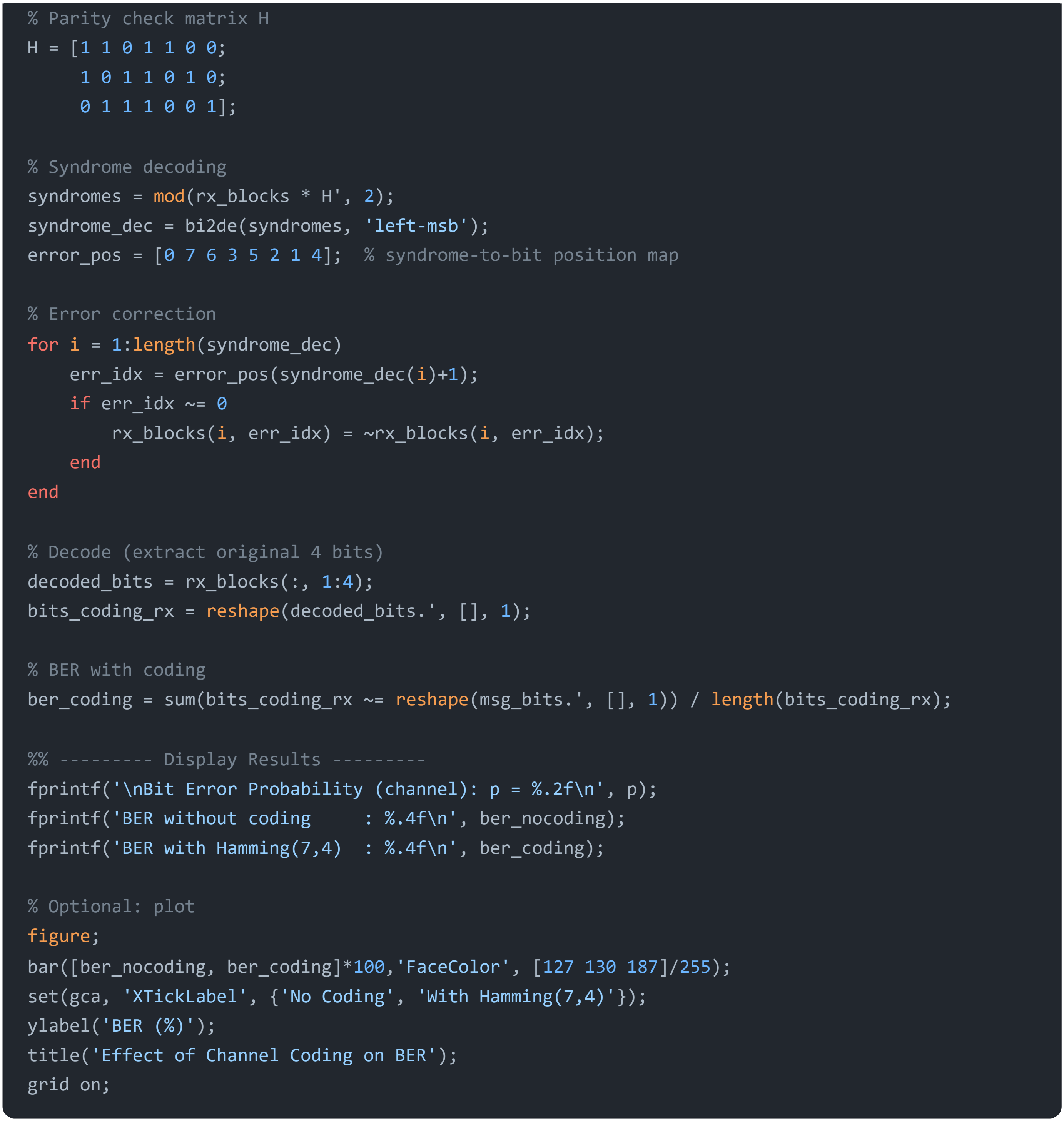
In some codes, even error correction is possible (e.g. Hamming).

Therefore, even if noise occurs, the wrong bits are corrected and as a result:

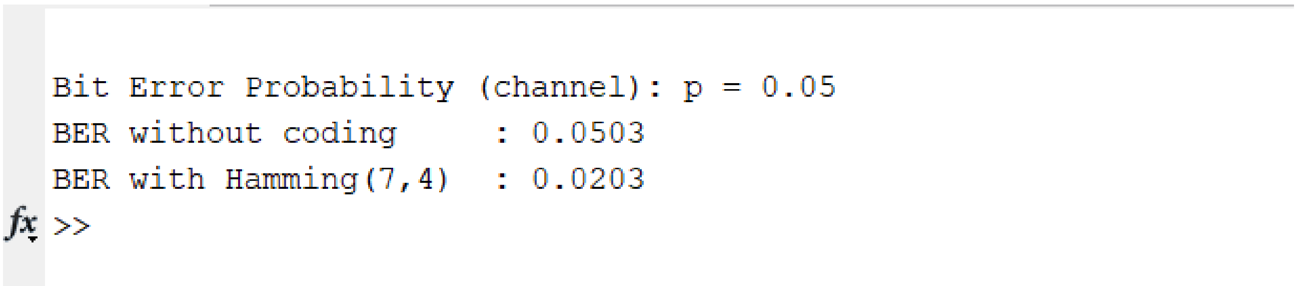
The final BER is reduced!

EXAMPLE :

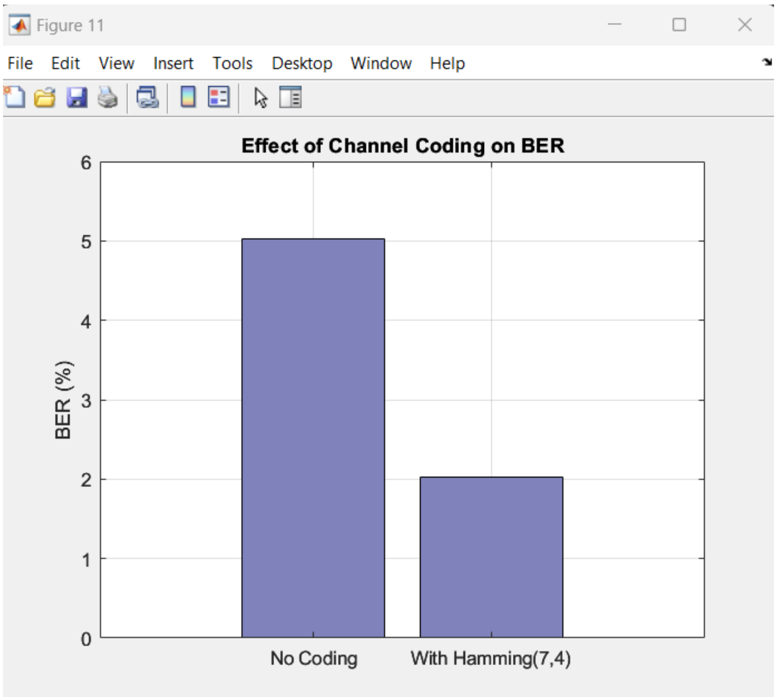
```matlab
clc; clear;

% Parameters
%rgb=[c]/255 ;
num_bits = 1e5;       % Number of bits
p = 0.05;             % Bit error probability (noise level)
num_blocks = floor(num_bits / 4);

% Generate random message bits
msg_bits = randi([0 1], num_blocks, 4);

%% --------- No Coding Case ---------
bits_nocoding = reshape(msg_bits.', [], 1);

% Simulate bit errors (BSC with flip probability p)
noise = rand(size(bits_nocoding)) < p;
rx_nocoding = xor(bits_nocoding, noise);

% BER without coding
ber_nocoding = sum(rx_nocoding ~= bits_nocoding) / length(bits_nocoding);

%% --------- With Hamming(7,4) Coding ---------
% Generator matrix for Hamming(7,4)
G = [1 0 0 0 1 1 0;
     0 1 0 0 1 0 1;
     0 0 1 0 0 1 1;
     0 0 0 1 1 1 1];

% Encode each 4-bit message
coded_bits = mod(msg_bits * G, 2);
bits_coding = reshape(coded_bits.', [], 1);

% Add noise (simulate errors)
noise_coding = rand(size(bits_coding)) < p;
rx_bits_coding = xor(bits_coding, noise_coding);
rx_blocks = reshape(rx_bits_coding, 7, []).';
```



```
% Parity check matrix H
H = [1 1 0 1 1 0 0;
     1 0 1 1 0 1 0;
     0 1 1 1 0 0 1];

% Syndrome decoding
syndromes = mod(rx_blocks * H', 2);
syndrome_dec = bi2de(syndromes, 'left-msb');
error_pos = [0 7 6 3 5 2 1 4];  % syndrome-to-bit position map

% Error correction
for i = 1:length(syndrome_dec)
    err_idx = error_pos(syndrome_dec(i)+1);
    if err_idx ~= 0
        rx_blocks(i, err_idx) = ~rx_blocks(i, err_idx);
    end
end

% Decode (extract original 4 bits)
decoded_bits = rx_blocks(:, 1:4);
bits_coding_rx = reshape(decoded_bits.', [], 1);

% BER with coding
ber_coding = sum(bits_coding_rx ~= reshape(msg_bits.', [], 1)) / length(bits_coding_rx);

%% --------- Display Results ---------
fprintf('\nBit Error Probability (channel): p = %.2f\n', p);
fprintf('BER without coding     : %.4f\n', ber_nocoding);
fprintf('BER with Hamming(7,4)  : %.4f\n', ber_coding);

% Optional: plot
figure;
bar([ber_nocoding, ber_coding]*100,'FaceColor', [127 130 187]/255);
set(gca, 'XTickLabel', {'No Coding', 'With Hamming(7,4)'});
ylabel('BER (%)');
title('Effect of Channel Coding on BER');
grid on;
```

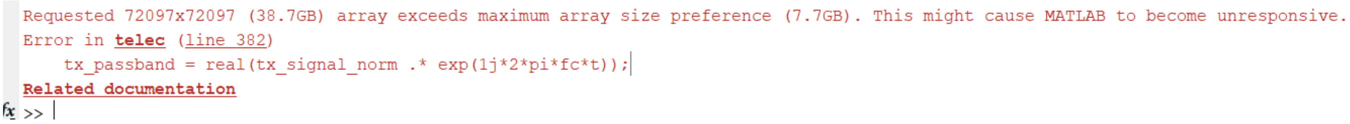


```
Bit Error Probability (channel): p = 0.05
BER without coding      : 0.0503
BER with Hamming(7,4)   : 0.0203
fx >>
```



NOW I WANNA TALK ABOUT ONE OF THE CHALLENGES I FACED IN THIS CODE AND HOW I FIXED IT !:

Now, there was a very important point here that I want to mention... Look, MATLAB has a certain amount of memory, and if we don't pay attention to this issue, when we are writing the code related to the BER part, MATLAB will give us an error that the length of that array or matrix is so long that we will run into a memory problem. I ran into exactly this problem and received this error:



```
Requested 72097x72097 (38.7GB) array exceeds maximum array size preference (7.7GB). This might cause MATLAB to become unresponsive.
Error in telec (line 382)
    tx_passband = real(tx_signal_norm .* exp(1j*2*pi*fc*t));
Related documentation
>>
```

To solve it, what I did was to see that I didn't need a number of variables, so I used the `clear` command to delete the ones I didn't need to free up memory:

```matlab
clear tx_signal_nc tx_signal_norm_nc t_nc exp_term_nc tx_passband_nc rx_signal_nc rx_bb_nc rx_mf_nc rx_syms_nc rx_symbols_nc rx_bits_nc
```