load('data.mat');

sampling\_frequency = 360;

time\_axis = (0:length(data)-1) / sampling\_frequency;

% Plot the original ECG signal

figure;

subplot(2,1,1);

plot(time\_axis, data);

xlabel('Time (seconds)');

ylabel('Voltage (mV)');

title('Original ECG Signal');

grid on;

% Quantization of the ECG signal (lossy compression)

quantization\_factor = 10; % Change this value for more or less precision

quantized\_data = round(data / quantization\_factor) \* quantization\_factor;

% Unique symbols and their probabilities for quantized data

symbols = unique(quantized\_data);

symbols = reshape(symbols, [], 1); % Ensure symbols is a column vector

bin\_edges = [symbols; max(symbols) + 1];

% Calculate probabilities based on the occurrence of each symbol

probabilities = histcounts(quantized\_data, bin\_edges) / numel(quantized\_data);

% Create nodes for each symbol

nodes = cell(1, numel(symbols));

for i = 1:numel(symbols)

nodes{i} = struct('symbol', symbols(i), 'probability', probabilities(i));

end

% Build the Huffman Tree

while numel(nodes) > 1

% Sort the nodes by probability

[~, sort\_idx] = sort(cellfun(@(x) x.probability, nodes));

% Extract two nodes with the smallest probabilities

node1 = nodes{sort\_idx(1)};

node2 = nodes{sort\_idx(2)};

% Create a new combined node

new\_node = struct('symbol', [], 'probability', node1.probability + node2.probability, 'left', node1, 'right', node2);

% Remove the two nodes and add the new node

nodes = [nodes(sort\_idx(3:end)), new\_node];

end

% Generate Huffman codes using containers.Map to avoid issues with field names

function codes = huffman\_codes(node)

codes = containers.Map('KeyType', 'double', 'ValueType', 'char');

traverse\_tree(node, '', codes);

end

function traverse\_tree(node, code, codes)

% Traverse the Huffman tree recursively to assign binary codes

if isempty(node.symbol)

% If the node is internal, traverse left and right

traverse\_tree(node.left, [code '0'], codes);

traverse\_tree(node.right, [code '1'], codes);

else

% If the node is a leaf, assign the code to the symbol

codes(node.symbol) = code;

end

end

% Generate Huffman codes

codes = huffman\_codes(nodes{1});

% Compress the data using the generated Huffman codes

compressed\_data = arrayfun(@(x) codes(x), quantized\_data, 'UniformOutput', false);

% Sort symbols and probabilities from most probable to least probable

[sorted\_probabilities, sorted\_indices] = sort(probabilities, 'ascend');

sorted\_symbols = symbols(sorted\_indices);

% Display the Huffman codes for each symbol, sorted by probability

for i = 1:numel(sorted\_symbols)

fprintf('Symbol: %d, Probability: %.5f, Code: %s\n', sorted\_symbols(i), sorted\_probabilities(i), codes(sorted\_symbols(i)));

end

% Entropy calculation

entropy = -sum(probabilities(probabilities > 0) .\* log2(probabilities(probabilities > 0)));

% Average code length calculation

average\_code\_length = 0;

for i = 1:numel(symbols)

code\_length = length(codes(symbols(i)));

average\_code\_length = average\_code\_length + probabilities(i) \* code\_length;

end

% Redundancy calculation

redundancy = average\_code\_length - entropy;

% Display results

fprintf('Entropy: %.5f bits\n', entropy);

fprintf('Average Code Length: %.5f bits\n', average\_code\_length);

fprintf('Redundancy: %.5f bits\n', redundancy);

% Save the compressed data as a cell array of binary strings

save('compressed\_data.mat', 'compressed\_data');

% Decoding the compressed data

decode\_map = containers.Map(values(codes), keys(codes)); % Invert the Huffman code map for decoding

% Convert compressed data into a single binary string

encoded\_str = strjoin(compressed\_data, '');

% Initialize variables for decoding

decoded\_data = zeros(1, length(quantized\_data)); % Preallocate array for decoded data

current\_code = '';

decoded\_idx = 1;

% Decode the binary string

for i = 1:length(encoded\_str)

current\_code = [current\_code, encoded\_str(i)]; % Append the current bit

if isKey(decode\_map, current\_code)

decoded\_symbol = decode\_map(current\_code); % Get the corresponding symbol

decoded\_data(decoded\_idx) = decoded\_symbol; % Store in the decoded array

decoded\_idx = decoded\_idx + 1;

current\_code = ''; % Reset the current code to start decoding the next symbol

end

end

% Verify that the decoded data matches the quantized data

if isequal(quantized\_data, decoded\_data)

disp('Decoding successful. The decoded data matches the quantized data.');

else

disp('Decoding failed. The decoded data does not match the quantized data.');

end

% Plot the decoded ECG signal

subplot(2,1,2);

plot(time\_axis, decoded\_data); % Plot the decoded data

xlabel('Time (seconds)');

ylabel('Voltage (mV)');

title('Decoded ECG Signal (Lossy)');

grid on;

% Create a bar graph of sorted symbols and their probabilities

figure;

bar(sorted\_symbols, sorted\_probabilities);

xlabel('Symbols');

ylabel('Probability');

title('Probability Distribution of Symbols');

grid on;

% Calculate original and compressed sizes

original\_size = numel(data) \* 8; % Assuming original data uses 8-bit symbols

compressed\_size = sum(cellfun(@length, compressed\_data)); % Total length of Huffman encoded data

% Data for the bar graph

sizes = [compressed\_size, original\_size - compressed\_size];

% Plot the bar graph

figure;

bar(sizes, 'FaceColor', 'flat');

set(gca, 'XTickLabel', {'Compressed', 'Uncompressed'});

ylabel('Size (Bits)');

title('Compression Ratio');

grid on;

% Add values on top of bars

text(1:length(sizes), sizes, num2str(sizes'), 'vert', 'bottom', 'horiz', 'center');