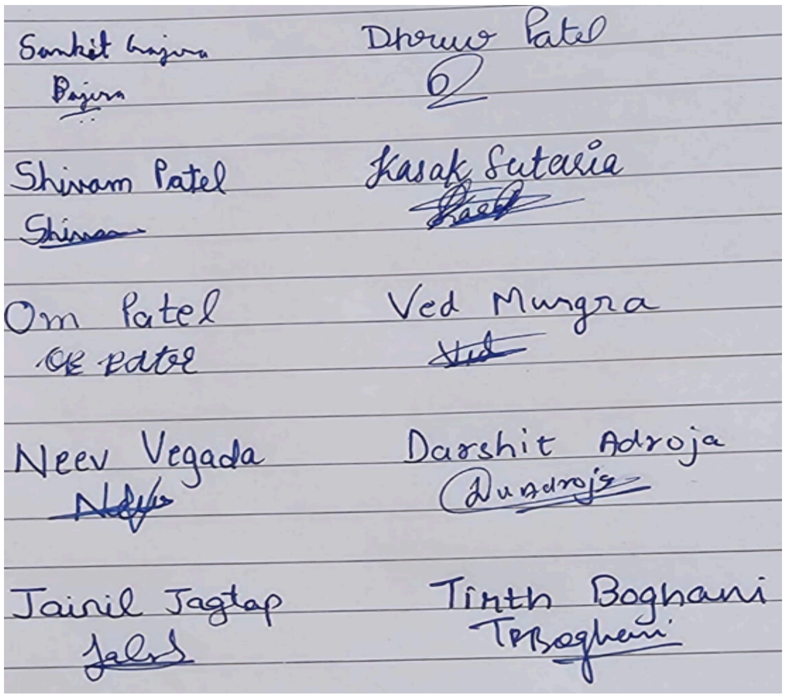
Group Number: 3
Convolution codes Matlab Code

We declare that

- The work that we are presenting is our own work.
- We have not copied the work (the code, the results, etc.) that someone else has done.
- Concepts, understanding, and insights we will be describing are our own.
- We make this pledge truthfully. We know that violation of this solemn pledge can carry grave consequences.



Sanket Gajera
Gajera

Dhruv Patel
[illegible]

Shivam Patel
[illegible]

Kasak Sutaria
[illegible]

Om Patel
[illegible]

Ved Mungra
[illegible]

Neev Vegada
[illegible]

Darshit Adroja
[illegible]

Jainil Jagtap
[illegible]

Tirth Boghani
[illegible]

Encoder,BPSK,AWGN,Decoder

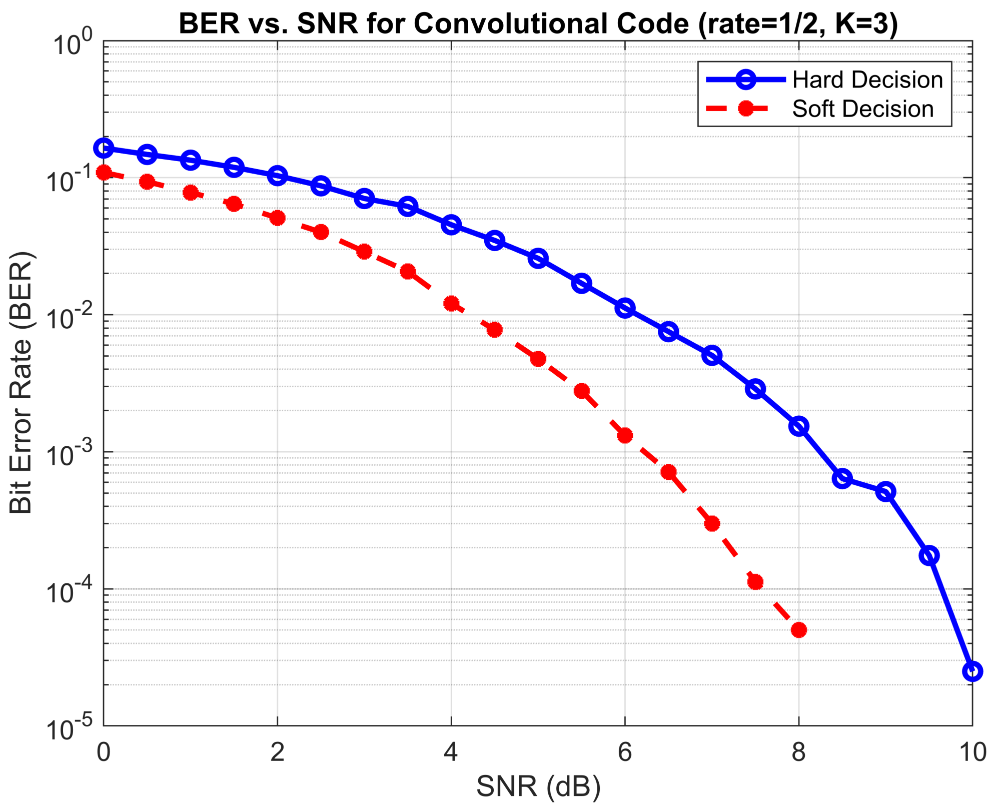
```
clc;
clear all;
```

Simulation Parameters:

```
snr_dB_range = 0:0.5:10;
snr_linear = 10.^(snr_dB_range / 10);
num_trials = 10000;
msg_bits = 6;
[~, num_snr_points] = size(snr_dB_range);
ber_uncoded = (1/2) * erfc(sqrt(snr_linear));
original_msg = generate_random_message(msg_bits);
```

(1) Code Rate:1/2 Constraint Length:3

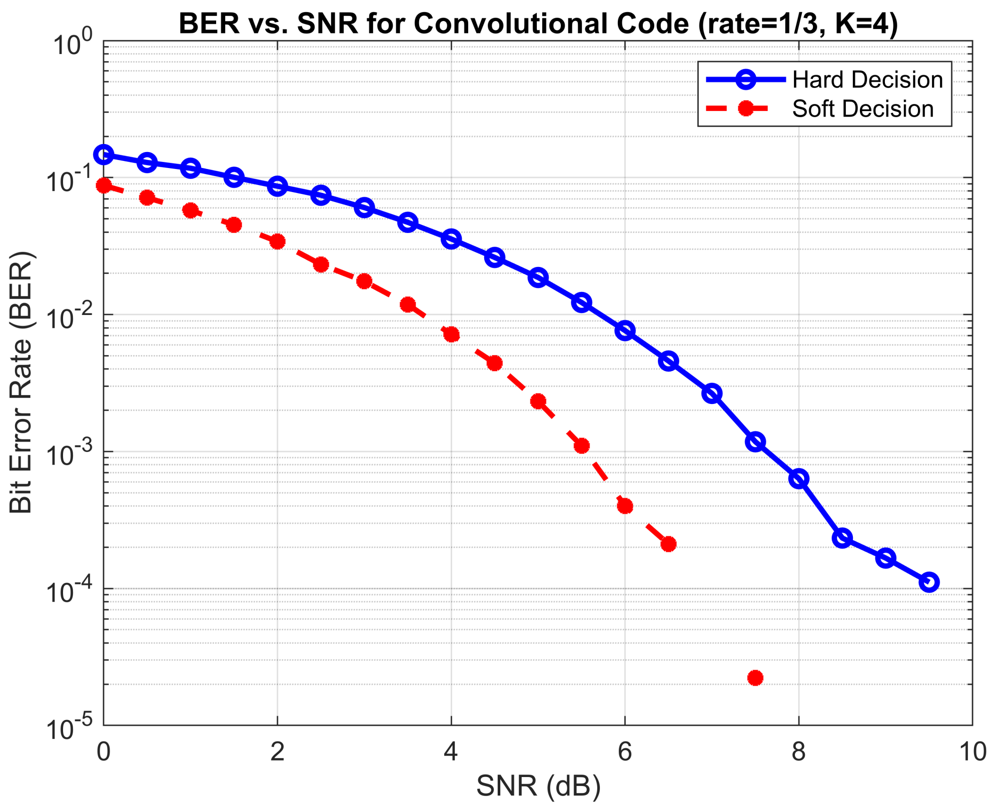
```
constraint_length = 3;
code_rate = 1/2;
generator_matrix = [1, 0, 1;
                    1, 1, 1];

[ref_msg, coded_bits] = encode_msg(constraint_length, original_msg,
generator_matrix);
modulated_symbols = map_bits_to_symbols(coded_bits);
noise_std_dev = sqrt(1 ./ (code_rate .* snr_linear));

% Initialize BER Arrays
ber1_hard = zeros(1, num_snr_points);
ber1_soft = zeros(1, num_snr_points);

% Simulation Loop
for snr_idx = 1:num_snr_points
    sigma = noise_std_dev(1, snr_idx);
    errors_hard = 0;
    errors_soft = 0;

    for trial = 1:num_trials
        noisy_symbols = add_awgn_noise(modulated_symbols, sigma);
        hard_bits = threshold_decoder(noisy_symbols);

        decoded_hard = viterbi_hard(generator_matrix, hard_bits);
        decoded_soft = viterbi_soft(generator_matrix, noisy_symbols);

        errors_hard = errors_hard + sum(decoded_hard ~= ref_msg);
        errors_soft = errors_soft + sum(decoded_soft ~= ref_msg);
    end

    total_bits = length(ref_msg) * num_trials;
    ber1_hard(1, snr_idx) = errors_hard / total_bits;
    ber1_soft(1, snr_idx) = errors_soft / total_bits;
end

% Plot Results
figure(1);
semilogy(snr_dB_range, ber1_hard, 'bo-', 'LineWidth', 2, 'MarkerSize', 6); hold on;
semilogy(snr_dB_range, ber1_soft, 'r*--', 'LineWidth', 2, 'MarkerSize', 6); hold
off;

xlabel('SNR (dB)');
ylabel('Bit Error Rate (BER)');
grid on;
legend('Hard Decision', 'Soft Decision');
title('BER vs. SNR for Convolutional Code (rate=1/2, K=3)');
```



(2) Code Rate:1/3 Constraint Length:4

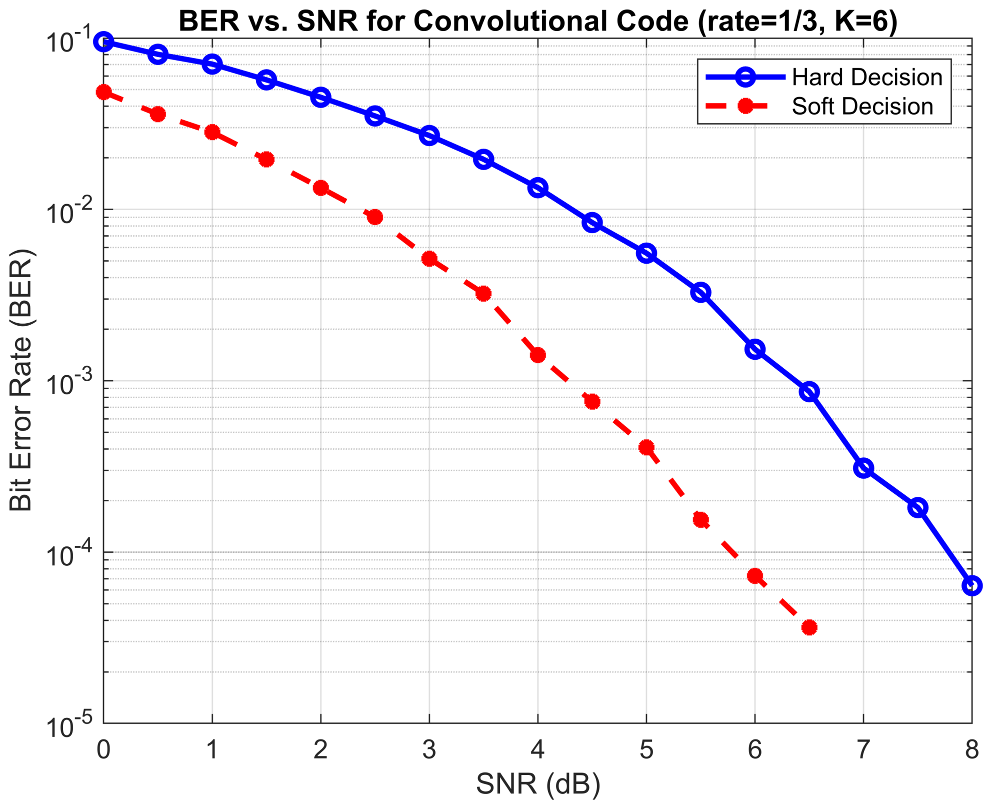
```
constraint_length = 4;
code_rate = 1/3;
generator_matrix = [1, 0, 1, 1;
                    1, 1, 0, 1;
                    1, 1, 1, 1];

[ref_msg, coded_bits] = encode_msg(constraint_length, original_msg,
generator_matrix);
```

```matlab
modulated_symbols = map_bits_to_symbols(coded_bits);
noise_std_dev = sqrt(1 ./ (code_rate .* snr_linear));

% Initialize BER Arrays
ber2_hard = zeros(1, num_snr_points);
ber2_soft = zeros(1, num_snr_points);

% Simulation Loop
for snr_idx = 1:num_snr_points
    sigma = noise_std_dev(1, snr_idx);
    errors_hard = 0;
    errors_soft = 0;

    for trial = 1:num_trials
        noisy_symbols = add_awgn_noise(modulated_symbols, sigma);
        hard_bits = threshold_decoder(noisy_symbols);

        decoded_hard = viterbi_hard(generator_matrix, hard_bits);
        decoded_soft = viterbi_soft(generator_matrix, noisy_symbols);

        errors_hard = errors_hard + sum(decoded_hard ~= ref_msg);
        errors_soft = errors_soft + sum(decoded_soft ~= ref_msg);
    end

    total_bits = length(ref_msg) * num_trials;
    ber2_hard(1, snr_idx) = errors_hard / total_bits;
    ber2_soft(1, snr_idx) = errors_soft / total_bits;
end

% Plot Results
figure(2);
semilogy(snr_dB_range, ber2_hard, 'bo-', 'LineWidth', 2, 'MarkerSize', 6); hold on;
semilogy(snr_dB_range, ber2_soft, 'r*--', 'LineWidth', 2, 'MarkerSize', 6); hold
off;

xlabel('SNR (dB)');
ylabel('Bit Error Rate (BER)');
grid on;
legend('Hard Decision', 'Soft Decision');
title('BER vs. SNR for Convolutional Code (rate=1/3, K=4)');
```



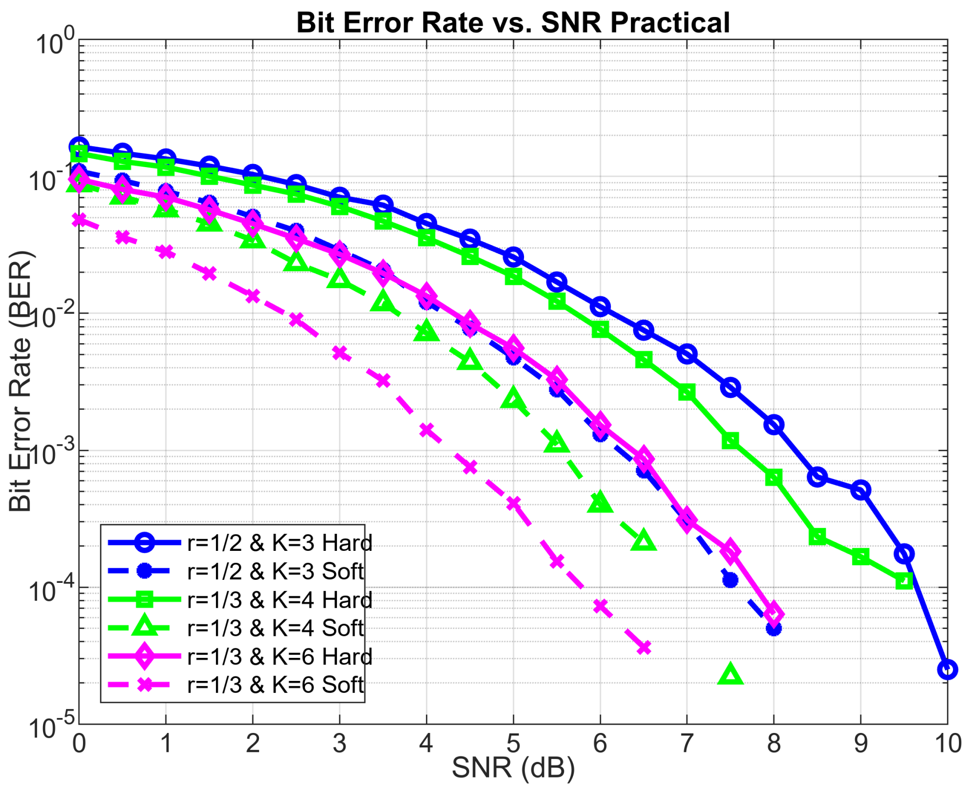
(3) Code Rate:1/3 Constraint Length:6

```
constraint_length = 6;
code_rate = 1/3;
generator_matrix = [1,0,0,1,1,1;
                    1,0,1,0,1,1;
                    1,1,1,1,0,1];

[ref_msg, coded_bits] = encode_msg(constraint_length, original_msg,
generator_matrix);
modulated_symbols = map_bits_to_symbols(coded_bits);
noise_std_dev = sqrt(1 ./ (code_rate .* snr_linear));

% Initialize BER Arrays
ber3_hard = zeros(1, num_snr_points);
ber3_soft = zeros(1, num_snr_points);

% Simulation Loop
for snr_idx = 1:num_snr_points
    sigma = noise_std_dev(1, snr_idx);
    errors_hard = 0;
    errors_soft = 0;

    for trial = 1:num_trials
        noisy_symbols = add_awgn_noise(modulated_symbols, sigma);
        hard_bits = threshold_decoder(noisy_symbols);
        decoded_hard = viterbi_hard(generator_matrix, hard_bits);
        decoded_soft = viterbi_soft(generator_matrix, noisy_symbols);

        errors_hard = errors_hard + sum(decoded_hard ~= ref_msg);
        errors_soft = errors_soft + sum(decoded_soft ~= ref_msg);
    end

    total_bits = length(ref_msg) * num_trials;
    ber3_hard(1, snr_idx) = errors_hard / total_bits;
    ber3_soft(1, snr_idx) = errors_soft / total_bits;
end

% Plot Results
figure(3);
semilogy(snr_dB_range, ber3_hard, 'bo-', 'LineWidth', 2, 'MarkerSize', 6); hold on;
semilogy(snr_dB_range, ber3_soft, 'r*--', 'LineWidth', 2, 'MarkerSize', 6); hold
off;

xlabel('SNR (dB)');
ylabel('Bit Error Rate (BER)');
grid on;
legend('Hard Decision', 'Soft Decision');
title('BER vs. SNR for Convolutional Code (rate=1/3, K=6)');
```



(4) Combined graph together

```
figure(4);
semilogy(snr_dB_range, ber1_hard, 'b-o', 'LineWidth', 2, 'MarkerSize', 6); hold on;
semilogy(snr_dB_range, ber1_soft, 'b--*', 'LineWidth', 2, 'MarkerSize', 6);
semilogy(snr_dB_range, ber2_hard, 'g-s', 'LineWidth', 2, 'MarkerSize', 6);
semilogy(snr_dB_range, ber2_soft, 'g--^', 'LineWidth', 2, 'MarkerSize', 6);
semilogy(snr_dB_range, ber3_hard, 'm-d', 'LineWidth', 2, 'MarkerSize', 6);
semilogy(snr_dB_range, ber3_soft, 'm--x', 'LineWidth', 2, 'MarkerSize', 6); hold
off;

xlabel('SNR (dB)');
ylabel('Bit Error Rate (BER)');
grid on;
legend('r=1/2 & K=3 Hard', 'r=1/2 & K=3 Soft', ...
       'r=1/3 & K=4 Hard', 'r=1/3 & K=4 Soft', ...
       'r=1/3 & K=6 Hard', 'r=1/3 & K=6 Soft', ...
       'Location', 'southwest');
title('Bit Error Rate vs. SNR Practical');
```



<u>Encoder Function</u>:

```
function [X, encodedBits] = encode_msg(K, M, g)

    Mlen = length(M);
    % appending K-1 zeros
    X = [M, zeros(1, K-1)];
    numGens = size(g, 1);
    totalSteps = length(X);
    encodedBits = zeros(1, numGens * totalSteps);

    % Initialize shift register to zeros
    shiftReg = zeros(1, K);
    outIdx = 1;

    % Slide through each bit of X
    for n = 1:totalSteps
        % Shift in the new bit
        shiftReg = [X(n), shiftReg(1:end-1)];

        % For each generator, compute output = (g(i,:), shiftReg) mod 2
        for i = 1:numGens
            encodedBits(outIdx) = mod( sum(g(i,:) .* shiftReg), 2 );
            outIdx = outIdx + 1;
        end
    end
end
```

Generate message bits function:

```
function msg = generate_random_message(length_bits)
    msg = rand(1, length_bits) > 0.5;
end
```

BPSK simulation:

```
function symbols = map_bits_to_symbols(bits)
    symbols = 1 - 2 * bits;
end
```

AWGN noise:

```
function noisy_signal = add_awgn_noise(signal, noise_std)
    noisy_signal = signal + noise_std * randn(1, length(signal));
end
```

Demodulation code:

```
function bits = threshold_decoder(values)
    bits = values < 0;
end
```

Hard Decision Viterbi Decoder:

```
function decodedBits = viterbi_hard(generatorMatrix, receivedBits)
```

```
    numGenerators = size(generatorMatrix, 1);
    memoryLength = size(generatorMatrix, 2);
    totalStates = 2^(memoryLength - 1);

    stateTransitions = zeros(totalStates, 2);
    stateOutputs = zeros(totalStates, 2);

    % Build trellis
    for currentState = 0:totalStates-1
        for inputBit = 0:1
            nextState = bitshift(currentState, -1);
            if inputBit == 1
                nextState = nextState + 2^(memoryLength - 2);
            end
            stateTransitions(currentState + 1, inputBit + 1) = nextState;

            currentBits = dec2bin(currentState, log2(totalStates)) - '0';
            fullState = [inputBit, currentBits];
            outputBits = [];
            for k = 1:numGenerators
                outputBits = [outputBits, mod(sum(bitand(generatorMatrix(k, :),
fullState)), 2)];
            end
            bitString = strrep(num2str(outputBits), ' ', '');
            stateOutputs(currentState + 1, inputBit + 1) = bin2dec(bitString);
        end
    end

    % Forward traversal using Hamming distance
    numColumns = length(receivedBits) / numGenerators + 1;
    pathMetrics = repmat(1000, totalStates, numColumns);
    pathMetrics(1, 1) = 0;
    bitCounter = 1;

    for col = 1:numColumns - 1
        segment = receivedBits(bitCounter:bitCounter + numGenerators - 1);
        bitCounter = bitCounter + numGenerators;

        receivedDecimal = sum(segment .* 2.^(length(segment)-1:-1:0));

        for state = 1:totalStates
            hamming0 = sum(bitget(bitxor(receivedDecimal, stateOutputs(state, 1)),
1:32));
            hamming1 = sum(bitget(bitxor(receivedDecimal, stateOutputs(state, 2)),
1:32));

            next0 = stateTransitions(state, 1) + 1;
            next1 = stateTransitions(state, 2) + 1;
            pathMetrics(next0, col + 1) = min(pathMetrics(next0, col + 1),
pathMetrics(state, col) + hamming0);
            pathMetrics(next1, col + 1) = min(pathMetrics(next1, col + 1),
pathMetrics(state, col) + hamming1);
        end
    end

   % Backtracking
currentState=0;
decodedBits = [];

for col = numColumns - 1:-1:1
    predecessors = [];

    for state = 1:totalStates
        for inputBit = 0:1
            if stateTransitions(state, inputBit + 1) == currentState
                predecessors = [predecessors; state, inputBit];
            end
        end
    end

    % Assume at most two predecessors
    previous1 = predecessors(1, :);
    previous2 = [];
    if size(predecessors, 1) > 1
        previous2 = predecessors(2, :);
    end

    % Get the received segment
    segment = [];
    for k = numGenerators - 1:-1:0
        segment = [segment, receivedBits(numGenerators * col - k)];
    end
    receivedDecimal = sum(segment .* 2.^(length(segment)-1:-1:0));

    % Compare costs
    hamming0 = sum(bitget(bitxor(receivedDecimal, stateOutputs(previous1(1),
previous1(2) + 1)), 1:32));
    cost0 = pathMetrics(previous1(1), col) + hamming0;

    if ~isempty(previous2)
        hamming1 = sum(bitget(bitxor(receivedDecimal, stateOutputs(previous2(1),
previous2(2) + 1)), 1:32));
        cost1 = pathMetrics(previous2(1), col) + hamming1;
    else
        cost1 = Inf;
        hamming1 = Inf;
    end
    % Collect both costs and hamming distances in arrays
    costs = [cost0, cost1];
    hammings = [hamming0, hamming1];
    previousStates = [previous1; previous2];

% Find the index of the minimum cost
    [minCost, idx] = min(costs);

% If both costs are equal, use hamming distance as tie-breaker
    if sum(costs == minCost) > 1
        [~, idx] = min(hammings);
    end

% Use the selected index to determine decoded bit and current state
    decodedBits = [decodedBits, previousStates(idx, 2)];
    currentState = previousStates(idx, 1) - 1;

end
decodedBits = flip(decodedBits);
end
```

Soft Decision Viterbi Decoder:

```
function decodedBits = viterbi_soft(generatorMatrix, receivedSignal)
    numGenerators = size(generatorMatrix, 1);
    memoryLength = size(generatorMatrix, 2);
    totalStates = 2^(memoryLength - 1);
    stateTransitions = zeros(totalStates, 2);
    stateOutputs = zeros(totalStates, 2);

    % Build trellis structure
    for currentState = 0:totalStates-1
        for inputBit = 0:1
            nextState = bitshift(currentState, -1);
            if inputBit == 1
                nextState = nextState + 2^(memoryLength - 2);
            end
            stateTransitions(currentState + 1, inputBit + 1) = nextState;

            currentBits = dec2bin(currentState, log2(totalStates)) - '0';
            fullState = [inputBit, currentBits];
            outputBits = [];
            for k = 1:numGenerators
                outputBits = [outputBits, mod(sum(bitand(generatorMatrix(k, :), 
fullState)), 2)];
            end
            bitString = strrep(num2str(outputBits), ' ', '');
            stateOutputs(currentState + 1, inputBit + 1) = bin2dec(bitString);
        end
    end
    % Forward traversal using Euclidean distance
    numColumns = length(receivedSignal) / numGenerators + 1;
    pathMetrics = repmat(500, totalStates, numColumns);
    pathMetrics(1, 1) = 0;
    bitCounter = 1;

    for col = 1:numColumns - 1
        segment = receivedSignal(bitCounter:bitCounter + numGenerators - 1);
        bitCounter = bitCounter + numGenerators;

        for state = 1:totalStates
            output0 = dec2bin(stateOutputs(state, 1), numGenerators) - '0';
            output1 = dec2bin(stateOutputs(state, 2), numGenerators) - '0';

            symbols0 = 1 - 2 .* output0;
            symbols1 = 1 - 2 .* output1;

            metric0 = sum((segment - symbols0).^2);
            metric1 = sum((segment - symbols1).^2);

            next0 = stateTransitions(state, 1) + 1;
            next1 = stateTransitions(state, 2) + 1;

            pathMetrics(next0, col + 1) = min(pathMetrics(next0, col + 1),
pathMetrics(state, col) + metric0);
            pathMetrics(next1, col + 1) = min(pathMetrics(next1, col + 1),
pathMetrics(state, col) + metric1);
        end
    end

    % Backtracking
    %[~, currentState] = min(pathMetrics(:, end));
    %currentState = currentState - 1;
    currentState=0;
    decodedBits = [];

    for col = numColumns - 1:-1:1
        predecessors = [];

        for state = 1:totalStates
            for inputBit = 0:1
                if stateTransitions(state, inputBit + 1) == currentState
                    predecessors = [predecessors; state, inputBit];
                end
            end
        end

        % Assume at most two predecessors
        previous1 = predecessors(1, :);
        previous2 = [];
        if size(predecessors, 1) > 1
            previous2 = predecessors(2, :);
        end

        % Get the received segment
        segment = [];
        for k = numGenerators - 1:-1:0
            segment = [segment, receivedSignal(numGenerators * col - k)];
        end

        % Output symbols and metrics
        out0 = dec2bin(stateOutputs(previous1(1), previous1(2) + 1), numGenerators)
- '0';
        symb0 = 1 - 2 .* out0;
        metric0 = sum((segment - symb0).^2);
        cost0 = pathMetrics(previous1(1), col) + metric0;

        if ~isempty(previous2)
            out1 = dec2bin(stateOutputs(previous2(1), previous2(2) + 1),
numGenerators) - '0';
            symb1 = 1 - 2 .* out1;
            metric1 = sum((segment - symb1).^2);
            cost1 = pathMetrics(previous2(1), col) + metric1;
        else
            cost1 = Inf;
            metric1 = Inf;
        end

        costs = [cost0, cost1];
        metrics = [metric0, metric1];
        previousStates = [previous1; previous2];

        [minCost, idx] = min(costs);
        if sum(costs == minCost) > 1
            [~, idx] = min(metrics);
        end

        decodedBits = [decodedBits, previousStates(idx, 2)];
        currentState = previousStates(idx, 1) - 1;
    end

    decodedBits = flip(decodedBits);
end
```