Projeto Interdisciplinar NEC120 e NEC130

Alunos:
Bruno Casu - 12.218.121-7
Rodrigo Yudji Katagiri de Oliveira - 12.119.228-0
Victor Garcia - 12.218.395-7
Bruno Duarte - 12.212.104-9

Execução do projeto:
Chamadas principais se encontram no arquivo 'main.m'
O projeto deve ser executado pela função 'Run Section', seguindo as orientações a seguir e observados os comentários pra cada secção.

1 Section:
Limpa o workspace do matlab.
Criação do vídeo em formato .avi (sem compressão): devem ser inseridas imagens na pasta local com o nome "scene0000x.jpeg", seguindo a numeração sequencial no nome dos arquivos.
Na pasta do projeto, já está disponibilizado um vídeo de exemplo com resolução baixa (128x72) para testes das funções.
Caso seja necessário a criação de um novo vídeo, remover os comentario do código.

2 Section:
Inicia o processamento do vídeo. A variável 'numfiles' define quantos frames serão processados (pode ser reduzida para testes), visto que o vídeo de exemplo possuiu 60 quadros de 128x72 pixels, rodando a uma taxa de 12 quadors por segundo (duração de 5s).
Após a definição dos quadros, a primeira função ('fonte') extrai as variáveis R, G e B do vídeo. Com isso, é executado frame por frame a quantização (função 'dct') e a serialização (função 'zig') das matrizes de crominânica (Pb e Pr) e da matriz de Luminância (Y).
Por fim, essas variáveis, contendo as amplitudes das frequências serializadas, são salvas no arquivo “video_ypbpr.mat”.

3 Section:
Carrega os vetores Y, Pb e Pr e executa a análise estatística das matrizes serializadas (função 'statistic'), criando os dicionários de Huffman para as variáveis Y_DC, Y_AC, PbPr_DC e PbPr_AC. Ao término do processo, esses dicionários são salvos para uso psterior (arquivo ‘huffman_dict.mat’).
A função seguinte ('huff') implementa a codificação de Huffman, usando os dicionário salvos, retornando o vídeo na forma binária (variável 'binary_serial_video').
Após esse processo é feito uma divisão para estimar a taxa de compressão em relação ao uso de 8 bits para codificar cada amplitude RGB (variável 'compression_rate').

4 Section:
Implementação do codificador convolucional (função ‘trellis_conv_encoder’), com taxas 1/2, 2/3, 3/4, 5/6, 7/8, podendo ser definidas na variável ’rate’.

5 Section:
Conversor serial para paralelo dos dados codificados para as entradas Q, I e A do Modulador 8QAM.
A taxa de bits da transmissão foi definida para 9Mbps. Após o conversor serial para paralelo, a taxa dos sinais Q, I e A é, portanto, de 3Mbps. Com isso, o valor do tempo da simulação (variável ‘tsym’) é calculado multiplicando o total de símbolos de 3 bits pelo tempo de símbolo, que nesse caso é 1/(3*10^6) segundos.

Deve ser mencionado que a simulação deve ter um período de símbolo adicionado, devido ao atraso do filtro em conjunto com o Sample-and-Hold no demodulador.

6 Section:
Execução do modelo no simulink: ‘modem_8QAM_v2018b.slx’.
Para a simluação, podem ser ajustados a atenuação do canal, em conjunto com o nível de ruído Gaussiano adicionado ao sinal modulado.
A portadora escolhida tem frequência 687,25MHz, na faixa de UHF, amostrada em uma taxa de 6,8725GHz.

7 Section:
Recuperação dos sinais demodulados no modelo do simulink.
Conversor Paralelo para serial dos sinais Q, I e A demodulados, resultando no vetor binário ‘binary_serial_video_demodulated’. Nessa etapa é feita uma avaliação entre os dados enviados ao modulador (variável ‘coded_data’), e os dados recuperados (variável ‘binary_serial_video_demodulated’), calculando o Bit Error Rate (variável ‘BER’).

8 Section:
Implementação do decodificador Viterbi (função ‘viterbi_decoder’), permitindo a correção dos possíveis erros encontrados no vetor serial.
Uma segunda avaliação do BER é feita na saida desse decodificador, permitindo a análise de sua eficiência.
Como sugestão para testes, definir o ‘numfiles’ da Section 2 para apenas 1 frame (usando o vídeo de exemplo), e utilizar atenuação do canal, no modelo do simulink, com valor 0.68, e amplitude do ruído (AWGN) em 2. Nessa situação, se for utilizado a taxa de codificação ½ no codificador convolucional, deve ser gerado um BER de aproximadamente 1% na saida do demodulador.
Após a execução do decodificador Viterbi, o BER obtido (variável ‘BER_after_FEC’) é reduzido para zero, ou seja, todos os erros foram corrigidos.

9 Section:
A secção final implementa os processos inversos para recuperação do vídeo. A primeira função ‘inv_huff’, recupera as amplitudes serializadas de luminância e crominância, utilizando os dicionários armazenados.
Após essa recuperação, o vetor serial é recomposto nas matrizes Y, Pb e Pr (função ‘inv_zig’), e a inversão da quantização é feita (função ‘inv_huff’), retornando os valores RGB para cada frame do vídeo, e organizando os quadros em uma única struct.
No final do processamento de todos os frames, o video é inserido na função ‘destino’, que então agrupa os quadros novamente na forma de vídeo, criando o arquivo “compressed_video.avi”. Executando o vídeo recuperado, nota-se que a resolução da imagem foi comprometida, pois o processo de quantização é feito com perdas. Apesar disso, o volume de dados que passou pelos processos de codificação de canal e modulação foi drasticamente reduzido, sem que o vídeo sofresse uma perda de qualiade equivalente. Portanto, os processos de compressão e recuperação do se mostraram bastante eficientes.