UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO
DEPARTAMENTO DE COMUNICAÇÕES

Tese de Doutorado

UM ESQUEMA DE CODIFICAÇÃO ADAPTATIVA DE IMAGENS USANDO TRANSFORMADA COSSENO DISCRETA

Ayres Mardem Almeida do Nascimento

Orientador: Prof. Dr. Yuzo Iano

Tese apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos exigidos para a obtenção do título de Doutor em Engenharia Elétrica.

Banca Examinadora:
Prof. Dr. Osamu Saotome (ITA/CTA/ELE)
Prof. Dr. Leonardo Silva Resende (UFSC/CTC/EEL)
Prof. Dr. José Antônio Siqueira Dias (UNICAMP/FEEC/DEMIC)
Prof. Dr. Edson Moschim (UNICAMP/FEEC/DSIF)
Prof. Dr. João Baptista Tadanobu Yabu-uti (UNICAMP/FEEC/DECOM)

Campinas, SP – Brasil
Dezembro de 2001

RESUMO

Estudos iniciais de codificação de imagem com VQ convencional revelaram várias dificuldades como a degradação de bordas e alta complexidade computacional. Uma nova abordagem de codificação adaptativa eficiente usando Transformada Rápida Cosseno Discreta (FDCT) para a redução de taxa de compressão de imagens é proposto. Esse método está baseado em um modelo de fonte composta que visa reduzir o efeito dessas degradações. Blocos são analisados segundo o nível de energia *ac* e ordenados em classes de acordo com seu nível de atividade. Cada classe com distintas características perceptuais, tais como borda, é gerada. Um classificador determina a subclasse de cada bloco de acordo com a orientação da borda presente no mesmo. A adaptatividade é conseguida com a distribuição de bits entre as classes, favorecendo aquelas com maior nível de atividade. Demonstra-se que, com o método proposto, a qualidade visual da imagem recuperada é comparável com aquela produzida pelos codificadores existentes para a taxa média de 1 bit/pixel, porém obteve-se melhor relação sinal/ruído.

ABSTRACT

Initial image coding studies with conventional VQ have shown some problems such as edge degradation and highly computational complexity. One new efficient method of adaptive coding using Fast Discrete Cosine Transform (FDCT) which reduces image compression rate is proposed. This method is based on composite source model, which helps in reduction of degradation effects. The blocks are analyzed according to their ac energy level and ordered in classes according to its activity level. Each class with its distinct perceptual characteristics, such as edge, are generated. One classifier determines the subclass of each block according to the present edge orientation of the same block. The adaptativity is obtained through the bits distribution between the classes, favoring the class of higher activity level. We have proved that with this proposed method, we can get visual image quality of the same produced quality by the existent coders of the bit rate of 1 bit/pixel but we get better Signal to Noise Ratio.

Dedico este Trabalho a minha esposa Maria do Rosário, a meus filhos Thiago, Caroline e Matheus pelo amor, carinho e compreensão a mim dispensados diariamente, e aos meus pais e irmãos pelo apoio espiritual.

AGRADECIMENTOS

Ao fim de uma longa jornada e ao atingir um objetivo tão almejado por mim e meus familiares, gostaria de externar meus agradecimentos a todas as pessoas que de alguma forma me apoiaram e me acompanharam durante todo o período de realização deste trabalho e em especial, gostaria de agradecer:

Ao Professor Yuzo Iano, meu orientador, pela amizade, confiança, incentivo e apoio a mim dedicado e orientação deste trabalho;

Aos colegas do curso do doutorado, pelo apoio e amizade;

Aos colegas da EMBRATEL, setor satélite, pelo incentivo e amizade;

Ao amigo Wilson de Oliveira, pela amizade e valioso apoio;

Aos funcionários da FEEC/UNICAMP, pelo auxílio;

Aos professores da Universidade do Amazonas, Departamento de Telecomunicações, pelo apoio, incentivo e amizade;

A todos os professores da UNICAMP, em particular aqueles do Departamento de Eletrônica e Comunicações da Faculdade de Engenharia Elétrica, que contribuíram para o meu aprendizado e crescimento científico.

Meus agradecimentos muito especiais aos amigos Evaldo Gonçalves Pelaes, pela ajuda e sugestões sempre valiosas e a presença amiga e ao Antonio Moraes pela amizade e acolhida calorosa que me dispensou em momentos importantes;

A todas as pessoas que, de alguma maneira, contribuíram nesta empreitada;

A minha esposa Maria do Rosário e meus filhos Thiago, Caroline e Mateus, que sempre estiveram ao meu lado dando-me apoio, amor e carinho;

A meus pais Dimas Rodrigues e Maria Almeida que sempre me incentivaram.

Finalmente,

Agradeço a DEUS por tudo, pois sem a sua graça nada teria conseguido.

AGRADECIMENTOS ESPECIAIS

Agradeço especialmente aos órgãos de apoio a pesquisa como a **CAPES** – Coordenação de Aperfeiçoamento de Pessoal de Nível Superior que, através da bolsa **CAPES/PICD**, viabilizou minha estadia e de minha família em Campinas durante o período de realização do meu programa de doutorado na UNICAMP, a **FAPESP** – Fundação de Amparo à Pesquisa do Estado de São Paulo, ao **CNPq –** Conselho Nacional de Desenvolvimento Cientifico e Tecnológico pelo apoio institucional, a **FAEP/UNICAMP –** Fundação de Apoio ao Ensino e à pesquisa da UNICAMP, ao **CPqD** – Fundação Centro de Pesquisa e Desenvolvimento que possibilitaram a aquisição de recursos e a existência de suportes computacionais para a realização do presente trabalho, a **UNICAMP –** Universidade Estadual de Campinas pelo apoio recebido dessa instituição durante a realização do meu curso de doutorado e um agradecimento muito especial a **UA** – Universidade do Amazonas que possibilitou minha liberação para a realização do meu curso de doutorado viabilizando assim a realização deste trabalho.

SUMÁRIO

Capítulo 1 Introdução

1.0 Considerações Iniciais 1

1.1 Objetivos, Vantagens e Limitações 2

1.2 Codificação Digital de Imagens por Transformada 4

1.3 Contribuições e Organização do Trabalho 6

Capítulo 2 Transformada Cosseno Discreta - DCT

2.1 Introdução 9

2.2. Transformada de Fourier (FT) 11

2.3. Transformada Cosseno de Fourier (FCT) 12

2.4. Transformada Discreta de Fourier (DFT) 14

2.5. Transformada Cosseno Discreta (DCT) 14

2.5.1. Transformada Cosseno Discreta do Tipo I (DCT-I) 16

2.5.2. Transformada Cosseno Discreta do Tipo II (DCT-II) via Transformada Discreta de Fourier (DFT) 19

2.5.3. Transformada Cosseno Discreta do Tipo III e IV 26

2.5.4. Visualização do Efeito da Transformada 1D 27

2.5.5. Transformada de Fourier Bidimensional (DFT-2D) 28

2.5.6. Transformada Cosseno Discreta Bidimensional (DCT-2D) 29

2.5.6.1. DCT-2D via DFT-2D 29

2.5.6.2. DCT-2D pela Redução em DCT-1D 31

2.5.7. Visualização do Efeito da Transformada 2D 34

Capítulo 3 Codificação de Imagens por Transformada DCT

3.1 Introdução 37

3.2. Codificação de Imagens 39

3.3 Transformadas de Imagens por Blocos 44

3.3.1 A transformada DCT 46

3.3.2 Seleção dos Coeficientes da Transformada 48
3.3.3 Quantização 52
3.3.3.1 Quantização Escalar 52
3.3.3.1.1 Quantizador Uniforme 56
3.3.3.1.2 Quantizador Não Uniforme 60
3.3.3.2 Quantização Vetorial 63
3.3.4 Projeto do Quantizador e Alocação de Bits para os Coeficientes Transformados 64
3.3.4.1 Técnicas baseadas em Experimentos Psicovisuais 65
3.3.4.2 Técnica baseada em Controle da Taxa de Bits 68

Capítulo 4 Aplicações da DCT em Compressão de Imagens

4.1 Introdução 73
4.2 DCT/VQ 73
4.2.1 Classificação de VQ com DCT (CVQ) 77
4.2.2 ADCT/VQ de Imagens Monocromáticas e Coloridas 80
4.2.3 DCT/VQ usando Categorização com Particionamento Adaptativo de Faixa 84
4.3 Codificação de Imagens por Transformada baseada na Estrutura/Distorção 86
4.3.1 Sobreposição de Blocos 87
4.3.2 Classificação de Atividades em Codificação Adaptativa por Transformada 89
4.3.2.1 Energia AC 90
4.3.2.2 Soma das Magnitudes dos Coeficientes AC 91
4.3.2.3 MACE e Direção 92
4.3.2.4 Distribuição de Potência dos Coeficientes 93
4.3.2.5 Segmentação Adaptativa dos Blocos DCT-2D em Regiões 93

Capítulo 5 Codificação Adaptativa de Imagens Usando DCT

5.1 Introdução 95

5.2 Codificação Adaptativa de Imagens usando DCT 97

5.2.1 Transformada Rápida Cosseno Discreta 97

5.2.2 Classificação de Atividade dos Blocos Transformados 101

5.2.3 Alocação de Bits para os Blocos Transformados 103

5.2.4 Normalização e Quantização dos Coeficientes Transformados 104

5.3. Simulações e Resultados 107

Capítulo 6 Conclusão

6.1 Considerações sobre as Transformadas 125

6.2 Resultados deste Trabalho 128

6.3 Sugestões para Trabalhos Futuros e Correlatos 130

6.4 Conteúdo do Trabalho Descrição por Capítulo 131

6.5 Destaques das Contribuições realizadas 132

REFERÊNCIAS ……...……..…… 135

Apêndice A 151

Apêndice B 163

Apêndice C 183

Publicações 259

LISTA DE FIGURAS

1.1 – Seqüência de codificação e decodificação de imagens por transformadas. 6

2.1a – Seqüência ***x[n]*** para o caso de ***N = 5***. 20

2.1b – Seqüência simétrica ***y[n]*** em relação ao ponto ***n = (2N-1)/2***, para ***N = 5*** 20

2.2 – Seqüência periódica $\tilde{x}[n]$ obtida pela repetição de *x[n]* 27

2.3 – Seqüência periódica $\tilde{y}[n]$ obtida pela repetição de *y[n]* com *2N* pontos. 28

2.4a – Seqüência ***x[m,n]*** 34

2.4b – Seqüência periódica $\tilde{x}[m,n]$ 34

2.5a – Seqüência *y[m,n] = x[m,n]+x[2M-1-m,n]+x[m,2N-1-n]+x[2M-1-m,2N-1-n]* 35

2.5b – Seqüência periódica $\tilde{y}[m,n]$ obtida de *x[m,n]* e *y[m,n]*. 35

3.1 – Diagrama de blocos simplificado de um codificador de imagens por transformada. ... 44

3.2 – Uma imagem *C x L* dividida de blocos *N x N* 45

3.3 – Diagrama simplificado de codificador de imagens por blocos usando DCT. 46

3.4 – Bloco da imagem original e bloco transformado 48

3.5 – Imagens base 8x8 da DCT 49

3.6 – Amostragem por zona geométrica 49

3.7 – Classificação por regiões de acordo com a estrutura dominante no bloco 50

3.8 – Distribuição de freqüência dos coeficientes DCT-2D e suas características 51

3.9 – Varredura em zig-zag e seqüência de transmissão dos coeficientes 51

3.10 – Quantizador Simétrico Uniforme a) *midtread*; b) *midriser* 54

3.10 c) – Quantizador Simétrico Não Uniforme 54

3.11 – Relação entrada-saída de um quantizador genérico 55

3.12 – Quantizador uniforme de 8 níveis 57

3.13 – Erro de quantização do quantizador da Fig. 3.12. 58

3.14 – Relação entrada-saída de 8 níveis para um quantizador 62

3.15 – Quantização vetorial com 2 escalares por vetor e 9 níveis de reconstrução 64

3.16 – Matrizes de quantização usadas no JPEG (Anexo K) 66

3.17 – Matrizes de quantização usadas no MPEG-2 67

4.1 – Codificador vetorial básico 76
4.2 – Combinação DCT/VQ de um arranjo 3D 76
4.3 – Sistema adaptativo de quantização vetorial com transformada DCT 77
4.4 – Classificação dos blocos para CVQ 78
4.5 – Codificador CVQ 79
4.6 – Codificador de blocos de classe de intervalo médio 79
4.7 – Codificador CVQ no domínio da DCT aplicado a seqüência de imagens 80
4.8 – Atividades parciais de um bloco transformado 4x4 80
4.9 – Codificador adaptativo DCT-VQ 81
4.10 – Particionamento de um bloco DCT em 14 vetores 82
4.11 – Quantização vetorial de dois estágios 84
4.12 – Esquema de VQ adaptativo por particionamento de faixa 85
4.13 – Padrões de máscara para categorização dos blocos transformados 85
4.14 – Particionamento de banda em cada categoria 86
4.15 – Sobreposição de blocos unidimensional 87
4.16 – Idéia base do processo de transformação de um bloco de tamanho *L* 88
4.17 – Reconstituição da Seqüência x[m,n] original por transformada inversa 89
4.18 – Segmentação adaptativa em blocos DCT-2D em regiões 94
5.1 – Visualização da equação (5.4) para seqüência ímpar 98
5.2 – Visualização da equação (5.4) para seqüência par 99
5.3 – Gráfico de fluxo da FDCT. 100
5.4 – Gráfico de fluxo da IFDCT. 100
5.5 – Classificação dos blocos em níveis de energia 101
5.6 – Direcionamento de bordas 102
5.7 – Sistema de codificação adaptativa de imagens por transformada cosseno 106
5.8 – Sistema de decodificação 107
5.9 – Imagem Lenna (512x512) original 109
5.10 – Codificada com DCT, taxa 1bpp, PSNR=31,52 dB e matriz de alocação de bits 109
5.11 – Codificada com LOT, taxa 1 bpp, PSNR=31,85 dB e matriz de alocação de bits.....109
5.12 – Codificada com DSTr, taxa 1bpp, PSNR=31,17 dB e matriz de alocação de bits 110

5.13 – Codificada com DCT, taxa 1bpp e PSNR=36.75 com 4 classes de energia 111
5.14 – Codificada com LOT, taxa 1bpp e PSNR=36.68 com 4 classes de energia 111
5.15 – Codificada com DSTr, taxa 1 bpp e PSNR=36.59 com 4 classes de energia 112
5.16 – Classificação dos blocos em classes e subclasses de energia 113
5.17 – Matrizes de alocação de bits das subclasses 114
5.18 – Imagem 512x512 original 115
5.19 – Quantização laplaciana e 4 classe 115
5.20 – Quantização gaussiana 4 clas./3sub 116
5.21 – Quantização laplaciana 4clas./3sub 116
5.22 – Classificação dos blocos em classes e subclasses de energia 116
5.23 – Quantização laplaciana 4clas./4sub 117
5.24 – Imagem original “Cozinha” 118
5.25 – Imagem codificada com 4 classes e 3 subclasses, laplaciana 118
5.26 – Imagem codificada com 4 classes 118
5.27 – Imagem original “Zelda” e 4 subclasses, laplaciana 118
5.28 – Imagem codificada com 4 classes e 3 subclasses, laplaciana 119
5.29 – Imagem codificada com 4 classes e 4 subclasses, laplaciana 119
5.30 – Imagem original “Bird” 119
5.31 – Imagem codificada com 4 classes e 3 subclasses, laplaciana 119
5.32 – Imagem codificada com 4 classes e 4 subclasses, laplaciana 119

LISTA DE TABELAS

1 – Desempenho de PSNR para a imagem Lenna codificada com DCT, LOT e DSTr, com uma matriz de alocação de bits 110

2 - Desempenho de PSNR para a imagem Lenna codificada com DSTr, LOT e DCT com 4 classes de energia e quantização laplaciana 112

3 – Distribuição de blocos e número de bits por categoria 113

4 - Desempenho de PSNR para a imagem Lenna codificada pelo método adaptativo proposto 115

5 – Desempenho de PSNR para imagem Lenna codificada com média de 1 bit/pixel 117

6 - Desempenho de PSNR para as imagens codificadas adaptativamente por DCT com o sistema proposto 118

7 - Codificação com 4 classes de energia, quantização por laplace e gauss com coeficiente de normalização $x/(f_n\, 2^{bij-1})$ f_n = maior σ para bij = 1 Imagem Lenna 256x256 Blocos DCT 8x8 Overhead = 0.811767 % 120

8 - Codificação com 4 classes de energia, quantização por laplace e gauss com coeficiente de normalização $x/(f_n\, 2^{bij-1})$ f_n = maior σ para bij = 1 Imagem Lenna 512x512 Blocos DCT 8x8 Overhead = 0.202942 % 120

9 - Codificação com 4 classes de energia, quantização por laplace e gauss com coeficiente de normalização $(x - \mu) / \sigma$ Imagem Lenna 256x256 Blocos DCT 8x8 Overhead = 2.37426758 % 120

10 - Codificação com 4 classes de energia, quantização por laplace e gauss com coeficiente de normalização $(x - \mu) / \sigma$ Imagem Lenna 512x512 Blocos DCT 8x8 Overhead = 0.59356689 % 121

11 - Codificação com 4 classes de energia com 3 subclasses cada, quantização por laplace e gauss com coeficiente de normalização $x/(f_n\, 2^{bij-1})$ f_n = maior σ para bij = 1 Imagem Lenna 256x256 Blocos DCT 8x8 Overhead = 2.8137207 % 121

12 - Codificação com 4 classes de energia com 3 subclasses cada, quantização por laplace e gauss com coeficiente de normalização $x/(f_n\ 2^{bij-1})$ f_n = maior σ para bij = 1 Imagem Lenna 512x512 Blocos DCT 8x8 *Overhead* = 1.875305 % 121

13 - Codificação com 4 classes de energia com 3 subclasses cada, quantização por laplace com coeficiente de normalização $x/(f_n\ 2^{bij-1})$ f_n = maior σ para bij = 1 Imagem BIRD 256x256 Blocos DCT 8x8 *Overhead* = 2.8137207 % 121

14 - Codificação com 4 classes de energia com 3 subclasses cada, quantização por laplace com coeficiente de normalização $x/(f_n\ 2^{bij-1})$ f_n = maior σ para bij = 1 Imagens Cozinha, Zelda e Sala respectivamente 512x512 Blocos DCT 8x8 *Overheader* = 1.875305 % 122

15 - Codificação com 4 classes de energia com 4 subclasses cada, quantização por laplace e gauss com coeficiente de normalização $x/(f_n\ 2^{bij-1})$ f_n = maior σ para bij = 1 Imagen: Lenna 512x512 Bloco DCT 8x8 *Overhead* = 1.979064941 % 122

16 - Imagem Cozinha 512x512 Blocos DCT 8x8 *Overhead* = 1.979064941 % 122

17 - Imagem Zelda 512x512 Blocos DCT 8x8 *Overhead* = 1.979064941 % 123

18 - Imagem BIRD 256x256 Blocos DCT 8x8 123

ABREVIATURAS

A/D : *Conversor Analógico/Digital*

BC : *Block Coding*

CCITT : *International Telegraph and Telephone Consultative Committee*

CCIR : *International Radio Consultative Committee*

CDROM : *Compact Disk Read Only Memory*

CMTT : *Committee for Mixed Telephone and Television*

CVQ : *Classified Vector Quantization*

D/A : *Conversor Digital/Analógico*

DCT : *Discrete Cosine Transform*

DCT-I : *Discrete Cosine Transform Type I*

DCT-II : *Discrete Cosine Transform Type II*

DCT-III : *Discrete Cosine Transform Type III*

DCT-IV : *Discrete Cosine Transform Type IV*

DCT–2D : *Discrete Cosine Transform Two Dimentional*

DFT : *Discrete Fourier Transform*

DFT-2D : *Discrete Fourier Transform Two Dimentional*

DM **:** *Delta Modulation*

DPCM : *Differential Pulse Code Modulation*

DSM : *Digital Storage Media*

DST : *Discrete Sine Transform*

DSTr : *Discrete Sine Transform with Axis Rotation*

FCT : *Fourier Cosine Transform*

FDCT : *Fast Discrete Cosine Transform*

FFT : *Fast Fourier Transform*

HDTV : *High-Definition Television*

HHT : *Hadamard-Haar Transform*

HIVITS : *High-quality Videofone and High-quality Television System*

HT : *Haar Transform*

HVS : *Human Visual System*

IDCT : *Inverse Discrete Cosine Transform*

IDFT-2D : *Inverse Discrete Fourier Transform Two Dimentional*

IEC : *International Engineering Consortium*

IFDCT : *Inverse Fast Discrete Cosine Transform*

ISDN : *Integrated Services Digital Network*

ISO : *International Standard Organization*

ISO/IEC : *International Standard Organization/International Engineering Consortium*

JPEG : *Joint Photographic Experts Group*

KLT : *Karhunen-Loève Transform*

LOT : *Lapped Orthogonal Transform*

MAE : *Mean Absolute Error*

MBPS : *Mega Bits Por Segundo*

MC : *Motion Compensation*

MC DPCM/DCT : *Motion Compensation Differential Pulse Code Modulation/Discrete Cosine Transform*

ME-MC : *Motion Estimation-Motion Compensation*

MMSE : *Minimum Mean Square Error*

MPEG : *Moving Pictures Experts Group*

MPEG-2 : *Moving Pictures Experts Group-2*

MSE : *Mean Square Error*

MVZS : *Maximum Variance Zonal Sampling*

NTSC : *National Television System Committee*

PAL : *Phase Alternating by Line*

PCM : *Pulse Code Modulation*

PDS : *Processamento Digital de Sinais*

PSNR : *Peak Signal-to-Noise Ratio*

RACE : *Research in Advanced Communication Technologies for Europe*

RDI : *Rede Digital Integrada*

SBC : *Sub-Band Coding*

SCT : *Simmetric Cosine Transform*

SHT : *Slant-Haar Transform*

SLT : *Slant Transform*

SMPTE : *Society of Motion Picture and Television Engineering*

SNR : *Signal-to-Noise Ratio*

VQ : *Vector Quantization*

VTRs : *Video Tape Records*

VWLC : *Variable Wordlength Code*

WFT : *Walsh-Hadamard Transform*

WLT : *Wavelet Transform*

CAPÍTULO 1

INTRODUÇÃO

1.0 Considerações Iniciais

Nos últimos anos, os sistemas de processamento e transmissão de sinais digitais tem crescido consideravelmente comparados aos sistemas analógicos. Esse crescimento deve-se aos avanços tecnológicos na área da microeletrônica, com o desenvolvimento de componentes complexos a baixo custo e às maiores vantagens oferecidas pelos sistemas digitais tais como: maior imunidade a ruídos, facilidade para proteção com códigos secretos, confiabilidade, compatibilidade com redes digitais, uso comum com computadores, facilidade de armazenamento etc. **[1]**.

Os sinais digitais de voz e de dados foram os primeiros a serem explorados comercialmente, aproveitando-se a infra-estrutura já existente para sistemas analógicos, surgindo portanto a Rede Digital Integrada – RDI, por onde trafegavam sinais de serviços analógicos e digitais. Porém, os serviços para os sinais de imagem tem despertado grande interesse comercial devido aos novos campos de aplicação, graças a digitalização dos mesmos. O processamento digital de imagem tem sido objeto de estudo em todo o mundo, visando as mais variadas áreas de aplicações, tais como: reconhecimento de padrão, enriquecimento de imagens, tratamento de sinais em estúdio, videofone, fac-símile, teleconferência, transmissão comercial de TV Digital, sensoriamento remoto através de análise e interpretação de imagens recebidas via satélite com o objetivo de estudo de mapeamento geográfico, análise de crescimento urbano e demográfico, desmatamento, queimadas, meteorologia, estudo estatístico de poluição, etc. Outras aplicações, não menos importantes, se encontram no armazenamento de imagens com aplicações comerciais, tais como: digitalização de arquivos históricos, bibliotecas, redes de comunicação de computadores, robótica, inspeção industrial automatizada, processamento de imagens médicas, radar e sonar **[2,3]**.

O crescimento da aplicabilidade do processamento digital de imagens deve-se ao aparecimento crescente de novos algoritmos de processamento que contemplam as diversas áreas de interesse. Os algoritmos de destaque referem-se as áreas de modelamento e representação de imagens, interpolação, restauração, análise de textura, codificação e compressão **[1-3].** Este trabalho se concentra no estudo da codificação e compressão de imagens digitalizadas, visando representá-las com o menor número possível de bits, com o objetivo de armazenamento e ou transmissão da mesma, preservando sua qualidade visual em um nível aceitável quanto a análise subjetiva e objetiva.

1.1 Objetivos, Vantagens e Limitações

Uma das limitações para a transmissão e armazenamento de sinais de imagens analógicas ou digitais é a grande capacidade de canal requerida e o espaço físico necessário, respectivamente, para a recuperação da imagem, preservando os níveis exigidos de qualidade e inteligibilidade necessária para uma dada aplicação **[1,3]**. Isso deve-se principalmente ao custo de transmissão ou armazenamento, que cresce inevitavelmente com o aumento do número de bits **[4]**. Tomando-se como referência o sinal de TV padrão M adotado no Brasil, que tem uma largura de faixa de 4,2MHz e usando uma freqüência de amostragem super-Nyquist de 10MHz com uma quantização uniforme de 8 bit/amostra, teremos uma taxa de bits codificados de 80Mbit/segundo, apenas para o sinal correspondente a imagem. Considerando que o sistema PCM (*Pulse Code Modulation*) 1ª hierarquia tem capacidade para 30 canais de voz para telefonia, conclui-se que essa taxa representaria mais de 1100 canais de voz. Observa-se, portanto, a existência da necessidade da redução da taxa de bits para a transmissão ou armazenamento de sinais digitalizados de TV ou outra imagem. Com esse objetivo, muito se tem pesquisado nos últimos anos, visando o fornecimento de embasamento teórico/prático para a implementação física de sistemas de redução da taxa de bits de sinais de imagens para transmissão ou armazenamento e que mantenham uma qualidade aceitável recomendada pelos órgãos normativos. Isso mostra que o processamento digital de imagens para transmissão ou

armazenamento será vantajoso caso se tenha uma técnica eficiente de codificação, que de alguma forma, remova as informações redundantes e irrelevantes contidas na imagem para minimizar a taxa de bits de representação da mesma. Portanto, a taxa de compressão da imagem codificada está relacionada tanto ao objetivo quanto à área de aplicação. Por exemplo, se uma imagem for codificada para transmissão, então o objetivo é a redução da largura de faixa de maneira tal a ajustá-la às limitações do meio de transmissão. Por outro lado, se for codificada para armazenamento, o objetivo é a redução do espaço físico necessário para armazená-la dado às limitações físicas dos bancos de dados que armazenam imagens digitalizadas. Quanto à área de aplicação, a preocupação está relacionada à qualidade e à inteligibilidade da imagem recuperada. Por exemplo, em serviços que envolvem a transmissão de imagens de televisão digital, teleconferência, fac-símile, etc, a qualidade da imagem recuperada é importante, porém, as informações que não forem perceptíveis ao olho humano, podem ser descartadas sem afetar a qualidade da imagem vista pelo observador final. Em serviços que envolvem o processamento e armazenamento de imagens médicas, imagens obtidas de satélites, acervos, bibliotecas, etc, a qualidade da imagem recuperada deve estar o mais próximo possível da imagem original. Portanto, no processo de codificação de imagens pode-se reduzir o número de bits necessário para representá-las sem o comprometimento da qualidade da imagem recuperada.

Várias técnicas têm sido propostas para a codificação de sinais de imagens visando a redução da taxa de bits. Essas técnicas exploram a correlação entre as amostras do sinal, em geral, utilizando a correlação espacial e/ou temporal entre amostras vizinhas de uma mesma linha, de linhas adjacentes do mesmo campo, de campos de um mesmo quadro ou de quadros adjacentes. Algumas técnicas fazem uso do DPCM (*Differential Pulse Code Modulation*) outras utilizam os recursos chamados de "Transformadas", tais como *Haar, Hadamard, Slant, Seno (DST), Cosseno (DCT), Lapped Orthogonal Transform (LOT), Wavelet, Karhunen-Loève (KLT) etc.* **[2-11]**, e outras utilizam técnicas denominadas "*Compensação de Movimentos*" *(MC)* **[12]**, "*Codificação de Blocos*" *(BC)***,** "*Codificação de Sub-bandas*" *(SBC)*, "*Quantização Vetorial*" *(VQ)* **[13]**, "*Modulação com Predição e Quantização*" **[14-17]**, etc.

Com o objetivo de aumentar a eficiência da redução da taxa de bits e manter a qualidade da imagem recuperada em níveis aceitáveis, tem-se utilizado técnicas que usam a combinação de métodos como: "*DPCM + DCT*", "*DPCM + MC*", "*DCT + VQ*", etc. Obviamente, essas técnicas têm uma maior complexidade de implementação quando comparadas com técnicas não combinadas, além de exigirem grande quantidade de memória, e maior esforço computacional, visto que usam amostras de um ou mais quadros sucessivos. Devido à eficiência das técnicas combinadas, as mesmas são amplamente usadas quando o processamento envolve imagens em movimento. Para imagens estáticas, costuma-se usar no processamento as técnicas de transformada, explorando-se algumas características da imagem, conforme a exigência da aplicação.

1.2 Codificação Digital de Imagens por Transformada

Na codificação de imagens por transformada, as imagens primeiramente sofrem uma transformação de domínio para um espaço que seja mais apropriado para a aplicação e que possibilite a exploração eficiente das características da imagem resultante transformada. A Transformada Cosseno Discreta (DCT) é hoje a mais usada em aplicações de processamento digital de imagens, devido não apenas à sua maior eficiência no descorrelacionamento das amostras vizinhas, como também, pela existência de algoritmos rápidos e econômicos para implementação em circuitos integrados.

A aplicação da DCT em processamento de imagens tem como objetivo a extração de redundância existente entre pixels vizinhos em uma imagem, através da descorrelação entre os mesmos. Porém, à medida que os *pixels* se distanciam, fica mais difícil esse descorrelacionamento. Para contornar essa dificuldade, aplica-se a DCT sobre blocos de tamanhos pré-definidos, gerando blocos de coeficientes transformados, sendo que alguns coeficientes concentram maior energia que outros, e cada coeficiente apresenta sua própria variância. Sobre esses coeficientes são aplicadas as mais diversas operações com os mais diversos objetivos, tais como: interpolação, decimação, filtragem, compressão etc.

Quando o objetivo do processamento de imagem é a compressão, aplica-se sobre os coeficientes transformados operações de quantização que podem ser escalar, vetorial ou ainda quantização diferencial. Na quantização escalar, cada coeficiente transformado é representado por um número de bits de acordo com uma tabela de alocação de bits para aquele coeficiente. Na quantização vetorial, dois ou mais coeficientes são conjuntamente representados por um escalar, que representa um conjunto de seqüências previamente estabelecidas denominado dicionário (*codebook*). Na quantização diferencial, cada coeficiente é quantizado segundo uma lei de quantização previamente estabelecida, de acordo com a importância da informação contida em cada coeficiente. Como nem todos os coeficientes possuem informações imprescindíveis, os coeficientes que possuem informações irrelevantes podem ser descartados sem prejuízo significante da qualidade da imagem recuperada. Conclui-se, portanto, que uma imagem codificada com um número reduzido de coeficientes pode ser recuperada com boa qualidade.

Para que uma imagem processada possa ser representada com o menor número de bits possível, os coeficientes transformados e quantizados passam por um processo de mapeamento de equivalência, onde cada coeficiente quantizado é mapeado em um conjunto de palavras-código, que pode ser de tamanho fixo ou variável, e essas palavras-código representarão a informação codificada. O processo de equivalência normalmente é feito com base na estatística do coeficiente quantizado, usando-se para isso o código de Huffman. Normalmente em compressão de imagens, procura-se obter um número médio de bits por *pixel* o mais próximo possível da entropia que é definida como sendo a média estatística das informações de cada símbolo.

Para a recuperação da imagem original, a imagem codificada passa pelo processo inverso, onde os coeficientes quantizados são recuperados a partir das palavras código e os coeficientes transformados são recuperados através da quantização inversa, sendo que as amostras aproximadas dos blocos originais são recuperadas através da transformada inversa. Apresenta-se na **Fig. 1.1** um diagrama de blocos que representa a seqüência do processo de codificação e decodificação de imagem por transformada.

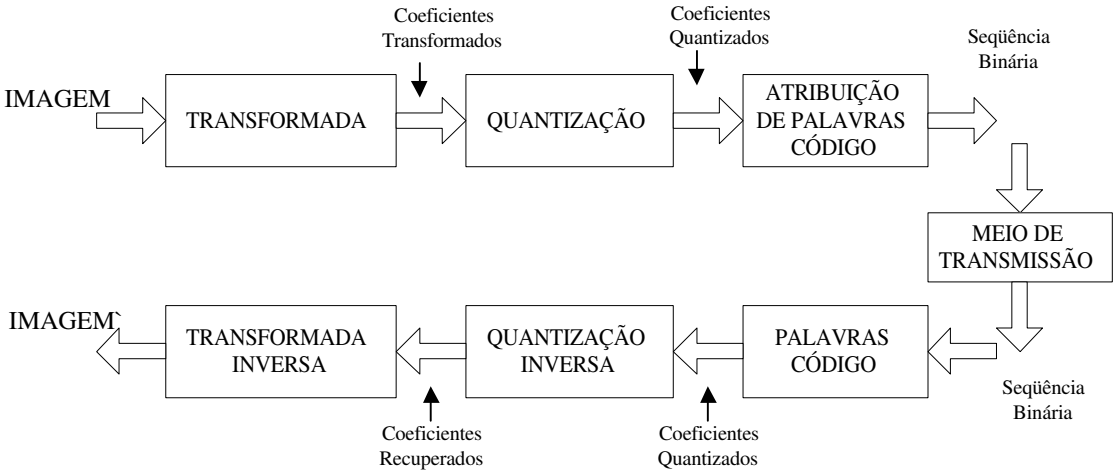


Fig.1.1 - Seqüência de codificação e decodificação de imagens por transformadas.

1.3 Contribuições e Organização deste Trabalho

A necessidade da codificação de imagens digitais com o objetivo da transmissão ou o armazenamento resultou no desenvolvimento de algoritmos de compressão rápidos, eficientes e que introduzissem o menor erro possível na imagem recuperada. Com esse propósito, várias técnicas foram desenvolvidas e, entre elas, a que tem apresentado um bom desempenho é aquela que usa a DCT. Como visto, na codificação de imagens por transformada, as imagens são processadas em blocos de tamanhos pré-definidos e cada bloco é processado independentemente. Devido a isso, alguns erros na codificação produzem descontinuidades entre os blocos da imagem recuperada. Essas descontinuidades devem-se ao fato de que as amostras da fronteira de um bloco não casam perfeitamente com as amostras da fronteira dos blocos adjacentes. Esses erros denominados efeitos de bloqueamento (ou efeito de blocos) aparecerão na imagem recuperada, causando uma sensação desagradável. Com a finalidade de reduzir esse efeito, muito se tem pesquisado e vários trabalhos têm sido desenvolvidos. Uma quantidade significante de algoritmos tem sido apresentada, visando a máxima compressão da imagem com o menor efeito de bloco possível **[9,10,18-20]** . A complexidade do algoritmo

depende fundamentalmente do propósito da aplicação e da qualidade requerida da imagem resultante. Para imagens em movimento, costuma-se usar processos que combinem técnicas diferentes, como DCT+DPCM, técnicas essas denominadas de codificação híbrida. Para imagens estáticas, costuma-se explorar características estatísticas da imagem. Como exemplo de aplicação que usa a DCT para compressão de imagens estáticas, destaca-se o padrão JPEG (*Joint Photographic Experts Group*) **[21]**. Para imagens em movimento, destaca-se o padrão MPEG-2 (*Moving Pictures Experts Group*) que usa a técnica híbrida DCT+DPCM com estimação e compensação de movimento *ME-MC* (*Motion Estimation-Motion Compensation*) **[22,23]**.

O presente trabalho tem como objetivo propor um algoritmo alternativo e eficiente de compressão de imagens estáticas ou em movimento, monocromáticas ou em cores, utilizando-se a transformada DCT–2D, e que apresente alta taxa de compressão com baixo efeito de bloqueamento. Esse algoritmo aborda uma codificação adaptativa eficiente para baixas taxas, mantendo a qualidade subjetiva da imagem. A codificação é feita baseada em um modelo de fonte que visa reduzir o efeito de bloco. Os blocos da imagem são analisados e classificados de acordo com seu nível de atividade, suas características perceptuais, orientação de bordas e tamanho. A adaptatividade é conseguida com a distribuição de bits entre as classes, favorecendo aquelas com maior nível de atividade. Para a avaliação comparativa de desempenho do algoritmo proposto foram realizadas simulações onde demonstramos que, com o método proposto, a qualidade subjetiva da imagem recuperada é comparável com as produzidas pelos codificadores analisados para taxas médias de 1 bit/pixel, porém, apresenta uma melhora na Relação Sinal/Ruído (*PSNR*) para as mesmas taxas, quando comparados com os mesmos codificadores. Para a realização das avaliações, foram desenvolvidos via *software* ambientes de simulação usando-se linguagem ***C***. Os programas são compostos de codificadores, decodificadores, quantizadores estatísticos, transformadas direta e inversa, programa de avaliação objetiva etc. Todos os *softwares* envolvidos foram desenvolvidos neste trabalho. Assim, este trabalho foi dividido da seguinte forma:

Capítulo 2: Nesse capítulo apresentam-se um estudo teórico da Transformada Discreta de Fourier (*DFT*) e suas propriedades, bem como sua evolução para a Transformada Cosseno discreta (*DCT*), que serão utilizadas no decorrer deste trabalho.

Capítulo 3: Nesse capítulo apresentam-se o processo que envolve a codificação de imagens por transformada DCT, sua aplicação em blocos da imagem, as considerações necessárias para quantização e escolha dos coeficientes para a transmissão ou armazenamento, projeto de quantizadores e técnicas de alocação de bits para os coeficientes mais significativos.

Capítulo 4: Nesse capítulo são analisados alguns trabalhos de aplicações de algoritmos que usam a transformada DCT em codificação para compressão de imagens.

Capítulo 5: Nesse capítulo apresentam-se o algoritmo proposto, os resultados das simulações das codificações de diversas imagens, as comparações com os resultados obtidos usando a DCT, LOT e DSTr, bem como a comparação com os resultados obtidos em outros trabalhos que visam a compressão de imagens.

Capítulo 6: Nesse capítulo apresentam-se as conclusões, analisando-se os resultados obtidos e vislumbrando-se as possibilidades de aprofundamento das contribuições deste trabalho.

Apêndice A: Nesse apêndice apresentam-se os quantizadores ótimos de Lloyd_Max para as funções densidade de probabilidade laplaciana e gaussiana usados para a quantização dos coeficientes transformados no processo de codificação proposto.

Apêndice B: Nesse apêndice apresentam-se os algoritmos rápidos para o cálculo da DCT-II usada neste trabalho.

Apêndice C: Nesse apêndice apresentam-se as listagens dos programas desenvolvidos para a simulação do sistema proposto.

CAPÍTULO 2

Transformada Cosseno Discreta - DCT

2.1 Introdução

A transformada é um mecanismo de operação matemática baseada na decomposição de sinais no domínio temporal, ou espacial, para o domínio do espectro de frequência, portanto, mais complexo do ponto de vista de cálculo. Devido a sua eficiência, o uso das transformadas tem crescido consideravelmente nos últimos anos em aplicações de processamento digital de sinais de voz, áudio e principalmente de imagens digitalizadas. Isso se deve ao aparecimento de novas técnicas de transformação **[2,9,10,24]**.

A operação por transformada pode ser vista como uma operação que tenta originar uma seqüência de coeficientes espectrais não correlacionados, reduzindo ao máximo a redundância de informação. Em sinais digitalizados, a correlação entre as amostras é tanto maior quanto mais próximas estiverem as mesmas. Portanto, para aumentar a eficiência do processamento por transformada, de uma seqüência discreta de comprimento N, dada por $x[n], n = 0, 1, 2, \ldots, N-1$, de sinais digitalizados, esses sinais são divididos em blocos de comprimento M tal que $N = kM$ e a transformada é aplicada sobre cada bloco, independentemente.

Para a aplicação da transformada no sinal cuja seqüência é definida por $x[n]$, define-se um vetor coluna X como um bloco de comprimento M extraído do sinal $x[n]$ tal que:

$$X = \{ x[kM - M + 1], x[kM - M + 2], \ldots, x[kM - 1], x[kM] \}^T \qquad (2.1)$$

onde T indica transposição do vetor X, e k representa a ordem do bloco dentro do sinal $x[n]$.

Se a transformada é definida por uma matriz de transformação C_k cujas linhas são formadas pelos vetores que compõem a base da transformada, então a transformação direta do vetor X é dada por:

$$\boldsymbol{Y} = \boldsymbol{C_k}\, \boldsymbol{X} \tag{2.2}$$

onde $\boldsymbol{Y}$ é o vetor transformado de $\boldsymbol{X}$. Se os elementos de $\boldsymbol{C_k}$ são complexos, então $\boldsymbol{T}$ indica a transposta da matriz conjugada. Sob a forma matricial, (2.2) é representada como:

$$\begin{bmatrix} y(0) \\ y(1) \\ \vdots \\ y(M-2) \\ y(M-1) \end{bmatrix} = \begin{bmatrix} C_0(0) & C_0(1) & \cdots & C_0(M-2) & C_0(M-1) \\ C_1(0) & C_1(1) & \cdots & C_1(M-2) & C_1(M-1) \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ C_{M-2}(0) & C_{M-2}(1) & \cdots & C_{M-2}(M-2) & C_{M-2}(M-1) \\ C_{M-1}(0) & C_{M-1}(1) & \cdots & C_{M-1}(M-2) & C_{M-1}(M-1) \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ \vdots \\ x(M-2) \\ x(M-1) \end{bmatrix}$$

A transformação inversa é dada por:

$$\boldsymbol{X} = [\boldsymbol{C_k}]^{-1}\, \boldsymbol{Y} \tag{2.3}$$

quando $[\boldsymbol{C_k}]^{-1} = [\boldsymbol{C_k}^*]^T$ diz-se que a transformada é unitária. Então (2.3) resulta em:

$$\boldsymbol{X} = [\boldsymbol{C_k}^*]^{\mathbf{T}}\, \boldsymbol{Y} \tag{2.4}$$

Essa equação é válida para qualquer matriz de transformação inversível (não singular e unitária).

Sob a forma matricial, (2.4) é representada como:

$$\begin{bmatrix} x(0) \\ x(1) \\ \vdots \\ x(M-2) \\ x(M-1) \end{bmatrix} = \begin{bmatrix} C_0^*(0) & C_1^*(0) & \cdots & C_{M-2}^*(0) & C_{M-1}^*(0) \\ C_0^*(1) & C_1^*(1) & \cdots & C_{M-2}^*(1) & C_{M-1}^*(1) \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ C_0^*(M-2) & C_1^*(M-2) & \cdots & C_{M-2}^*(M-2) & C_{M-1}^*(M-2) \\ C_0^*(M-1) & C_1^*(M-1) & \cdots & C_{M-2}^*(M-1) & C_{M-1}^*(M-1) \end{bmatrix} \begin{bmatrix} y(0) \\ y(1) \\ \vdots \\ y(M-2) \\ y(M-1) \end{bmatrix}$$

e sob a forma de somatório (2.3) e (2.4) são dadas por:

$$y[k] = \sum_{m=0}^{M-1} C_k[m]\, x[m] \qquad \text{onde} \quad k = 0, 1, 2, \ldots, M\text{-}1 \tag{2.5}$$

$$x[m] = \sum_{k=0}^{M-1} \left[C_k^*[m]\right]^T y[k] \qquad \text{onde} \quad m = 0,1, 2, \ldots, M\text{-}1. \tag{2.6}$$

onde os $C_k[\,m\,]$ representam as funções bases da transformada direta e $Y[\,k\,]$ a transformada da seqüência discreta $x[m]$.

Em processamento de sinais por transformadas os sinais são geralmente divididos em blocos e a transformada é aplicada independentemente em cada bloco. Além disso, cada coeficiente transformado é quantizado individualmente sendo esses processamentos independentes. Esses fatos geralmente dão origem a descontinuidades entre blocos adjacentes do sinal recuperado, devido aos erros de quantização. Essas descontinuidades são conhecidas como efeitos de bloqueamento **[10]**.

2.2. Transformada de Fourier (FT)

A Transformada de Fourier (***FT***) de uma função contínua ***x(t)*** definida no intervalo $-\infty < t < \infty$ é dada em **[9, 25]** por:

$$X(\varpi) \equiv F[x(t)] = \left(\tfrac{1}{2\pi}\right)^{1/2} \int_{-\infty}^{\infty} x(t) e^{-j\varpi t} dt \qquad \begin{matrix} \varpi = 2\pi f \\ j = \sqrt{-1} \end{matrix} \tag{2.7}$$

onde ϖ é a freqüência em radianos e f é a freqüência em Hertz. Sua transformada inversa é definida como:

$$x(t) \equiv F^{-1}[X(\varpi)] = \left(\tfrac{1}{2\pi}\right)^{1/2} \int_{-\infty}^{\infty} X(\varpi) e^{j\varpi t} d\varpi \tag{2.8}$$

As Equações (2.7) e (2.8) são definidas como o par de transformada de Fourier direta e inversa, respectivamente.

2.3. Transformada Cosseno de Fourier (FCT)

Se ***x(t)*** é definida somente para $t \geq 0$, então, tomando-se uma função ***y(t)*** que seja a extensão par de ***x(t)*** definida sobre toda a parte real, tal que:

$$y(t) = x(|t|) \qquad t \in R \tag{2.9}$$

onde $y(.)$ é uma função par do argumento (.), ou seja:

$$y(t) = \begin{cases} x(t) & ; \quad t \geq 0, \\ x(-t) & ; \quad t \leq 0. \end{cases} \tag{2.10}$$

a Transformada de Fourier de ***y(t)*** é dada por:

$$Y(\varpi) = F[y(t)] = \left(\frac{1}{2\pi}\right)^{1/2} \int_{-\infty}^{\infty} y(t)\, e^{-j\varpi t}\, dt \tag{2.11}$$

A integral em (2.11) pode ser avaliada em duas partes, de $(-\infty, 0]$ e $(\infty, 0]$. Então, usando (2.10) em (2.11), obtém-se:

$$Y(\varpi) = \left(\frac{1}{2\pi}\right)^{1/2} \int_{-\infty}^{0} x(-t)\, e^{-j\varpi t} dt + \left(\frac{1}{2\pi}\right)^{1/2} \int_{0}^{\infty} x(t)\, e^{-j\varpi t} dt \tag{2.12}$$

Trocando a variável de integração no intervalo $(-\infty, 0]$ de ***t*** para ***-t*** obtém-se:

$$Y(\varpi) = \left(\frac{1}{2\pi}\right)^{1/2}\left[\int_0^{\infty} x(t)\, e^{j\varpi t}\, dt + \int_0^{\infty} x(t)\, e^{-j\varpi t}\, dt\right]$$

$$Y(\varpi) = \left(\frac{1}{2\pi}\right)^{1/2}\left[\int_0^{\infty} x(t)\left(e^{j\varpi t} + e^{-j\varpi t}\right) dt\right] \quad ,$$

onde

$$\frac{e^{j\varpi t} + e^{-j\varpi t}}{2} \;=\; Re\left[e^{j\varpi t}\right] \;=\; cos(\varpi t) \tag{2.13}$$

logo:

$$Y(\varpi) = 2\left(\frac{1}{2\pi}\right)^{1/2}\int_0^{\infty} x(t)\cos(\varpi t)\, dt \tag{2.14}$$

A Equação (2.14) é definida como sendo a Transformada Cosseno de Fourier (***FCT***) de ***x(t)*** sendo representada por:

$$X_C(\varpi) \;\equiv\; F_C[x(t)] \;=\; \left(\frac{2}{\pi}\right)^{1/2}\int_0^{\infty} x(t)\cos(\varpi t)\, dt \tag{2.15}$$

Nota-se que $X_C(\varpi)$ é uma função par de ϖ, portanto, pode-se aplicar a Transformada Inversa de Fourier em (2.14) ou (2.15) para $t \geq 0$, obtendo-se:

$$y(t) = x(t) \;\equiv\; F_C^{-1}[X_C(\varpi)] = \left(\frac{2}{\pi}\right)^{1/2}\int_0^{\infty} X_C(\varpi)\cos(\varpi t)\, d\varpi; \qquad t \geq 0 \tag{2.16}$$

As Equações (2.15) e (2.16) definem o **Par de Transformada Cosseno de Fourier (*FCT*)** de ***x(t)*** (algumas propriedades da *FCT* podem ser vistas em **[9]**).

Considerando que a função cosseno é uma parte real de uma função exponencial de argumento puramente imaginário, verifica-se que existe um relacionamento próximo entre a **Transformada de Fourier** e a **Transformada Cosseno** dado por:

$$F[y(t)] \;=\; 2F_C[x(t)] \qquad se \qquad y(t) \;=\; x(|t|) \tag{2.17}$$

2.4. Transformada Discreta de Fourier (DFT)

Segundo **[24]**, a definição da DFT tem duas interpretações: uma é que a DFT é a representação de uma seqüência ***x[n]*** como sendo uma combinação linear de exponenciais complexas harmonicamente relacionadas, a outra é que a DFT corresponde à discretização da Transformada de Fourier de um seqüência discreta ***x[n]***.

A Transformada Discreta de Fourier (***DFT***) de uma seqüência ***x[n]*** finita de comprimento ***N,*** com ***n = 0, 1, 2, ..., N-1***, é uma representação no domínio da freqüência de ***x[n]***, e é definida em **[2-4,24]** como:

$$X[k] = \sum_{n=0}^{N-1} x[n]\, e^{-j\frac{2\pi k}{N}n}, \qquad k = 0, 1, 2, ..., N-1 \tag{2.18}$$

A Transformada Inversa Discreta de Fourier ***(IDFT)*** é definida como:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k]\, e^{j\frac{2\pi n}{N}k}, \qquad n = 0, 1, 2, ..., N-1 \tag{2.19}$$

2.5. Transformada Cosseno Discreta (DCT)

A aplicação da DFT em processamento digital de sinais (PDS) teve um fundamental crescimento em 1965 quando Cooley & Tukey **[26]** desenvolveram o primeiro algoritmo rápido para a implementação eficiente da DFT . O rápido crescimento da tecnologia de dispositivos digitais foi seguido de várias modificações, melhorias e enriquecimento do algoritmo básico de Cooley & Tukey, que passaram a ter adicional importância na área de aplicação da DFT. De maneira semelhante, o descobrimento da Transformada Cosseno Discreta (DCT – *Discrete Cosine Transform*) em 1974 **[27]** proporcionou um significante impacto na área de processamento digital de sinais. O algoritmo original da DCT está baseado

na FFT (*Fast Fourier Transform*). Porém, o principal ponto de partida na implementação eficiente da DCT foi baseado em um algoritmo recursivo em aritmética real desenvolvido por Chen, Smith & Fralick em 1977 **[28]**. Posteriormente, outros algoritmos foram desenvolvidos, concentrando-se na redução da complexidade computacional e/ou na simplificação estrutural.

A Transformada Cosseno Discreta tem como principal característica a decomposição de um sinal em suas componentes de freqüências, agrupando os maiores valores nas componentes de baixa freqüência, e diminuindo de valor mais rapidamente com o crescimento da frequência. Essa característica foi, primeiramente, usada de forma eficiente em projetos de filtros e, posteriormente, pode também ser usada na compressão de imagens.

Existem outras transformadas discretas que também podem ser usadas em processamento de imagens digitalizadas, tal como Walsh-Hadamar (WFT), Slant (ST), Haar (HT), seno discreto (DST), etc, e também transformadas híbridas como Slant-Haar (SHT), Hadamard-Haar (HHT), etc. Essas transformadas também possuem propriedades similares àquelas da DCT. Existe, entretanto, uma transformada que tem melhor capacidade de compactação de energia que as outras transformadas. Tal transformada denominada Transformada Karhunen-Loève (KLT) é considerada como ótima. Essa transformada descorrelaciona completamente uma seqüência de sinais **[3,9,11,24,29]**, entretanto, não garante por si só esquemas computacionais rápidos e eficientes, visto que necessita do conhecimento prévio da estatística do sinal, o que é praticamente impossível para sinais aleatórios. A transformada DCT mostrou ser assintoticamente equivalente à transformada ótima KLT para a descorrelação dos sinais, além de poder ser computada eficientemente de maneira similar à TF **[27]**.

A DCT é considerada uma transformada ligeiramente sub-ótima na sua eficiência de compactação de energia e mostrou ser superior às demais transformadas, na compressão de largura de faixa (redução de redundância) de uma ampla variedade de sinais, tais como: voz, sinais de TV, imagens infra vermelho, fotovideotexto, textura de superfície, etc. A DCT tem suas próprias áreas de atuação, como em análise espectral de seqüências reais, em soluções de alguns problemas de valores limites e em processamento de sinais no domínio da transformada. Em particular, sua versão discreta tem encontrado grande preferência na comunidade de processamento digital de sinais. Entretanto, devido à sua grande eficiência

computacional, ela tornou-se a transformada mais utilizada em processamento de imagens que envolvem codificação por transformadas, e é utilizada nos padrões JPEG (*Joint Photographic Experts Group*) e MPEG (*Moving Picture Experts Group*).

Em processamento de imagens, a codificação por transformada não é aplicada sobre a imagem como um todo, mas sim em blocos de tamanho fixo 4x4, 8x8, 16x16, etc. A razão para isso deve-se aos seguintes fatos:

1- A correlação entre pixels é menor entre pixels distantes do que entre *pixels* vizinhos.
2- A transformada de pequenos blocos é mais fácil de ser computada do que de toda a imagem.

2.5.1. Transformada Cosseno Discreta do Tipo I (DCT-I)

Da Equação (2.15), chamando o núcleo da FCT de:

$$K_C(\varpi, t) = cos(\varpi t) \tag{2.20}$$

então, (2.15) e (2.16) resultam respectivamente em:

$$X_C(\varpi) = \left(\frac{2}{\pi}\right)^{1/2} \int_0^{\infty} x(t)\, K_C(\varpi, t)\, dt \tag{2.21}$$

e

$$y(t) = x(t) = \left(\frac{2}{\pi}\right)^{1/2} \int_0^{\infty} X_C(\varpi)\, K_C(\varpi, t)\, d\varpi; \qquad t \geq 0 \tag{2.22}$$

Considerando,

$$\begin{aligned} \varpi_m &= 2\pi\, m\, \Delta f \\ t_n &= n\, \Delta t \end{aligned} \tag{2.23}$$

onde Δf é uma unidade de intervalo de amostra da frequência f e Δt é uma unidade de intervalo de amostra no tempo t. Os ω_m e t_n representam as amostras angulares da frequência e do tempo respectivamente, onde m e n são inteiros. Assim, o núcleo pode ser reescrito na forma:

$$K_C(\varpi_m, t_n) = K_C(m,n) = \cos(2\pi\, m\Delta f\, n\Delta t) \tag{2.24}$$

Em aplicações práticas, a computação da Transformada Cosseno é feita com dados amostrados de duração finita. Portanto, fazendo:

$$\Delta f\, \Delta t = \frac{1}{2N} \tag{2.25}$$

onde N é um número inteiro, tem-se portanto:

$$K_C(m,n) = \cos\left(\frac{\pi\, m\, n}{N}\right) \qquad com \quad m,n = 0,1,\cdots N \tag{2.26}$$

onde (2.26) representa o **Núcleo da Transformada Cosseno de Fourier Discretizada**.

Considerando o núcleo discreto $K_C(m,n)$ como sendo elementos em uma matriz de transformação com $(N+1)(N+1)$ elementos, então, denotando essa matriz por $[M]$, os mn-ésimos elementos dessa matriz são dados por:

$$[M]_{mn} = \cos\left(\frac{\pi\, m\, n}{N}\right) \qquad m,n = 0,1,2,...N \tag{2.27}$$

A discretização de um tempo de duração finita é feita dividindo-o em N intervalos de tempo, com duração de Δt cada.

Incluindo os pontos extremos, existem $N+1$ pontos de amostras para serem considerados. Se os $N+1$ pontos de amostras discretas forem representados por um vetor

coluna $\boldsymbol{x} = [x(0),x(1),...x(N)]^T$, quando $\boldsymbol{[M]_{mn}}$ for aplicado nesse vetor, obter-se-á um vetor coluna $\boldsymbol{X} = [X(0),X(1),...X(N)]^T$ tal que:

$$X = \begin{bmatrix} X(0) \\ X(1) \\ \vdots \\ X(N) \end{bmatrix} = [M]_{mn} * \begin{bmatrix} x(0) \\ x(1) \\ \vdots \\ x(N) \end{bmatrix} \tag{2.28}$$

o qual, na forma de elemento a elemento, é dado por:

$$X(m) = \sum_{n=0}^{N} cos(\frac{\pi\, m\, n}{N})\, x[n] \qquad m = 0,1,...N \tag{2.29}$$

O vetor $\boldsymbol{x[n]}$ é dito ter sofrido uma transformação discreta. A Equação (2.29) é uma **Transformada Cosseno Discreta**. Ela foi primeiramente reportada por Kitajiman, em 1980 **[30]**, e foi denominada de **Transformada Cosseno Simétrica** (***SCT***).

Aplicando-se fatores de normalização apropriados em (2.26), obtém-se uma matriz de transformação $\boldsymbol{[C]}$ dada por:

$$[C]_{mn} = \sqrt{\frac{2}{N}} \left\{ K_m K_n\, cos(\frac{\pi\, m\, n}{N}) \right\} \qquad m,n = 0,1,\cdots N \tag{2.30}$$

onde,

$$K_j = \begin{cases} \dfrac{1}{\sqrt{2}} & ; \quad j = 0 \text{ ou } N \\ 1 & ; \quad j \neq 0 \text{ ou } N \end{cases} \qquad j = m \text{ ou } n \tag{2.31}$$

Pode-se mostrar que (2.30) é uma matriz unitária **[9]**. Pela propriedade da unitariedade pode-se mostrar que $\boldsymbol{[C] = [C]^T = [C]^{-1}}$ **[30]**.

Segundo a classificação feita por Wang **[31]**, (2.30) representa o **Núcleo da Transformada Cosseno Discreta do tipo I (DCT-I)**. Como em (2.29), na forma de elemento a elemento $X_c[m]$ resulta em:

$$X_c[m] = \left(\frac{2}{N}\right)^{\frac{1}{2}} K_m \sum_{n=0}^{N} K_n \, x[n] \, cos\left(\frac{\pi \, m \, n}{N}\right) \qquad m = 0,1,...,N \tag{2.32}$$

e sua inversa em:

$$x[n] = \left(\frac{2}{N}\right)^{\frac{1}{2}} K_n \sum_{m=0}^{N} K_m \, X_c[m] \, cos\left(\frac{\pi \, m \, n}{N}\right) \qquad n = 0,1,...,N \tag{2.33}$$

Os vetores $X_c[m]$ e $x[n]$ são ditos serem o **Par de Transformada Cosseno Discreta do Tipo I (DCT-I)**. Esses vetores são compostos de $N+1$ amostras, devido ao fato de que suas amostras extremas são diferentes de zero.

2.5.2. Transformada Cosseno Discreta do Tipo II (DCT-II) via Transformada Discreta de Fourier (DFT)

Segundo Wang **[31]** e Rao **[9]**, a Transformada Cosseno Discreta foi classificada em quatro tipos: ***DCT-I***, ***DCT-II***, ***DCT-III*** e ***DCT-IV***. A Transformada Discreta Cosseno do Tipo II (***DCT-II***) foi primeiramente reportada por Ahmed, Natarajan e Rao **[27]**, e é atualmente a mais usada em processamento digital de sinais (imagens).

Na derivação dessa transformada, Ahmed et. al. **[27]** e Kitajima **[30]** consideram o problema da diagonalização da matriz de covariância e asseguram a unitariedade da matriz de transformação, uma vez que, na diagonalização, elas são matrizes transformadas similares.

Na verificação da Equação (2.15) observa-se que a *Transformada de Fourier de uma função par resulta na definição da Transformada Cosseno de Fourier (**FCT**)*. Essa

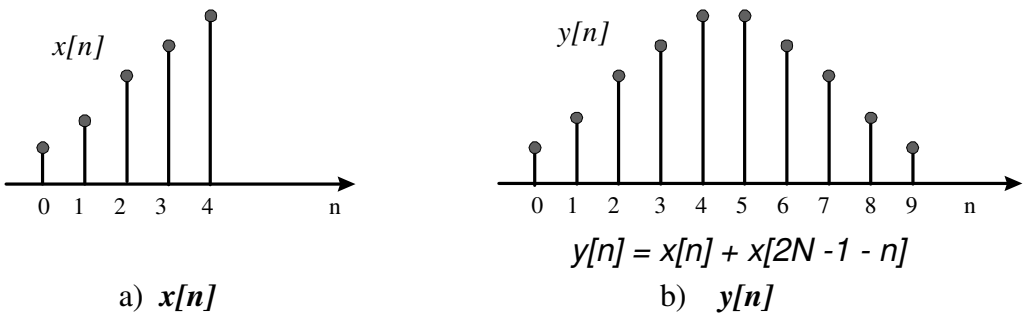
propriedade pode ser explorada para a computação da Transformada Cosseno Discreta tipo II (***DCT-II***), considerando um vetor (seqüência) ***x[n]*** de comprimento ***N***, com ***n*** *= 0,1,2,...N-1* e zero fora desse intervalo. Então, define-se uma seqüência estendida ***y[n]*** com ***2N*** pontos **[3]**, simétrica em torno do ponto ***(2N-1)/2*** de tal modo que {***y(n)***} possa ser escrita na forma:

$$y[n] = \begin{cases} x[n] & ; \quad n = 0,1,2,\ldots,N-1 \\ x[2N-1-n] & ; \quad n = N, N+1,\ldots,2N-1 \end{cases} \tag{2.34}$$

onde

$$y[n] = x[n] + x[2N-1-n] \tag{2.35}$$

É mostrado nas **Figs. 2.1a)** e **2.1b)** um exemplo das seqüências ***x[n]*** e ***y[n]*** para o caso de ***N = 5***. Como pode ser visto, a seqüência ***y[n]*** apresenta uma simetria em relação ao ponto ***n = (2N-1)/2***.



a) ***x[n]*** b) ***y[n]***

Fig. 2.1 – Seqüências ***x[n]*** e ***y[n]***

A seqüência ***y[n]*** é usada como o passo intermediário para a definição da Transformada Cosseno Discreta de ***x[n]*** via DFT. Portanto, a ***DFT*** da seqüência ***y[n]*** de ***2N*** pontos é dada por:

$$Y[k] = \sum_{n=0}^{2N-1} y[n]\, e^{-j\frac{2\pi}{2N}nk} \tag{2.36}$$

Fazendo $e^{-j\frac{2\pi}{2N}} = W_{2N}$ tem-se:

$$Y[k] = \sum_{n=0}^{2N-1} y[n] W_{2N}^{nk} \tag{2.37}$$

Substituindo a Equação (2.34) na Equação (2.37) tem-se:

$$Y[k] = \sum_{n=0}^{N-1} x[n] W_{2N}^{nk} + \sum_{n=N}^{2N-1} x[2N-1-n] W_{2N}^{nk} \tag{2.38}$$

No segundo somatório, fazendo:

$$m = 2N\text{-}1\text{-}n \quad para \;\; n = N \;\Rightarrow\; m = N\text{-}1 \tag{2.39}$$

e

$$n = 2N\text{-}1\text{-}m \quad para \;\; n = 2N\text{-}1 \;\Rightarrow\; m = 0$$

tem-se portanto,

$$Y[k] = \sum_{n=0}^{N-1} x[n] W_{2N}^{nk} + \sum_{m=0}^{N-1} x[m] W_{2N}^{(2N-1-m)k} \tag{2.40}$$

A substituição de ***m*** por ***n*** resulta em:

$$Y[k] = \sum_{n=0}^{N-1} x[n] W_{2N}^{nk} + \sum_{n=0}^{N-1} x[n] W_{2N}^{(2N-1-n)k} \tag{2.41}$$

Como:

$$W_{2N}^{(2N-1-n)} = W_{2N}^{2N} W_{2N}^{-(n+1)} \qquad e \qquad W_{2N}^{2N} = e^{-j\frac{2\pi}{2N}2N} = 1 \tag{2.42}$$

logo

$$W_{2N}^{(2N-1-n)} = W_{2N}^{-(n+1)} \tag{2.43}$$

e portanto,

$$Y[k] = \sum_{n=0}^{N-1} x[n] W_{2N}^{nk} + \sum_{n=0}^{N-1} x[n] W_{2N}^{-(n+1)k} \tag{2.44}$$

$$Y[k] = \sum_{n=0}^{N-1} x[n] \left[W_{2N}^{nk} + W_{2N}^{-nk} W_{2N}^{-k} \right] \tag{2.45}$$

$$Y[k] = \sum_{n=0}^{N-1} x[n] W_{2N}^{-\frac{k}{2}} \left[W_{2N}^{nk} W_{2N}^{\frac{k}{2}} + W_{2N}^{-nk} W_{2N}^{-\frac{k}{2}} \right] \tag{2.46}$$

$$Y[k] = W_{2N}^{-\frac{k}{2}} \sum_{n=0}^{N-1} x[n] \left[W_{2N}^{(n+\frac{1}{2})k} + W_{2N}^{-(n+\frac{1}{2})k} \right] \tag{2.47}$$

$$Y[k] = 2W_{2N}^{-\frac{k}{2}} \sum_{n=0}^{N-1} x[n] \cos\left[\frac{(2n+1)\pi k}{2N} \right] \tag{2.48}$$

onde ***k = 0,1,...,N-1*** e ***n = 0,1,...,N-1***.

A Equação (2.48) é a Transformada Cosseno Discreta (***DCT***) da seqüência ***x[n]*** de ***N*** pontos exceto por um fator k_m , enquanto que a Equação (2.36) é a Transformada Discreta de Fourier (***DFT***) da seqüência ***y[n]*** de ***2N*** pontos. Portanto, de (2.48) tem-se:

$$W_{2N}^{\frac{k}{2}} Y[k] = \sum_{n=0}^{N-1} 2\, x[n] \cos\left[\frac{(2n+1)\pi k}{2N} \right] \tag{2.49a}$$

sendo, ***k = 0,1,...,N-1***. Portanto,

$$X[k] = \sum_{n=0}^{N-1} 2\, x[n] \cos\left[\frac{(2n+1)\pi k}{2N} \right] \tag{2.49b}$$

onde:

$$X[k] = \begin{cases} W_{2N}^{\frac{k}{2}} Y[k] & ; \quad 0 \le k \le N-1 \\ 0 & ; \quad fora \end{cases} \tag{2.49c}$$

As Equações (2.49) provêm um meio de computar a ***DCT*** de ***x[n]*** via uma ***DFT*** de ***2N*** pontos da seqüência ***y[n]***.

Quando a seqüência ***x[n]*** é real, ***y[n]*** é real e simétrica. Nesse caso, Haralick **[32]** mostrou que ***Y[k]*** pode ser obtido via duas ***FFTs*** de ***N*** pontos, além de poder ser obtida por uma simples ***FFT*** de ***2N*** pontos.

Uma vez que a ***FFT*** de ***N*** pontos requer $N\log_2 N$ operações complexas em geral, tal aproximação representa uma economia de ***2N*** operações complexas. Tseng e Miller **[33]**, entretanto, mostraram que quando a seqüência de entrada ***x[n]*** é real, a ***DFT radix-2*** da seqüência ***y[n]*** real, par, de ***2N*** pontos, requer apenas $\left(\frac{N}{2}\log_2 N + \frac{N}{2}\right)$ operações complexas. O termo $\frac{N}{2}$ surge do fator de escala requerido como em (2.49). Isto representa uma significante redução na complexidade da computação de duas ***DFT$_s$*** de ***N*** pontos.

Das Equações (2.48) e (2.49), observa-se que o núcleo discreto dessa ***DCT*** é dado por:

$$K_C(m,n) = \cos\left[\frac{\pi(2n+1)m}{2N}\right] \qquad m,n = 0,1,\cdots,N-1 \tag{2.50}$$

portanto, esse núcleo é considerado como sendo os coeficientes de uma matriz de transformação com ***(N)x(N)*** elementos.

Aplicando-se um fator de normalização apropriados como em (2.30), obtém-se uma matriz de transformação ***[C]*** dada por:

$$[C]_{mn} = \sqrt{\frac{2}{N}} \left\{ K_m \cos\left[\frac{\pi\,(2n+1)m}{2N} \right] \right\} \qquad m,n = 0,1,\cdots,N-1 \tag{2.51}$$

onde

$$K_m = \begin{cases} \dfrac{1}{\sqrt{2}} & ; \quad m = 0 \\ 1 & ; \quad m \neq 0 \end{cases} \tag{2.52}$$

Segundo classificação feita por Wang **[31]**, (2.51) representa o ***Núcleo da Transformada Cosseno Discreta do tipo II (DCT-II)***. Analogamente a (2.29), na forma de elemento a elemento, $\boldsymbol{X_c[m]}$ resulta em:

$$X_c[m] = \left(\frac{2}{N} \right)^{\frac{1}{2}} K_m \sum_{n=0}^{N-1} x[n] \cos\left[\frac{\pi\,(2n+1)m}{2N} \right] \quad m = 0,1,\cdots,N-1 \tag{2.53}$$

Da mesma maneira que em (2.30), (2.51) é uma matriz unitária **[3]**. Portanto, a transformada inversa é dada por:

$$x[n] = \left(\frac{2}{N} \right)^{\frac{1}{2}} \sum_{m=0}^{N-1} K_m\, X_c[m] \cos\left[\frac{\pi\,(2n+1)m}{2N} \right] \quad n = 0,1,\cdots,N-1 \tag{2.54}$$

Os vetores $\boldsymbol{X_c[m]}$ e $\boldsymbol{x[n]}$ são compostos de $\boldsymbol{N}$ amostras e ditos serem o Par de Transformada Cosseno Discreta do tipo II (***DCT-II***).

Uma outra maneira de se representar o par de transformada (2.53) e (2.54) é dada por:

$$X_c[m] = \left(\frac{1}{N} \right)^{\frac{1}{2}} K_m \sum_{n=0}^{N-1} x[n] \cos\left[\frac{\pi\,(2n+1)m}{2N} \right] \qquad m = 0,1,\cdots,N-1 \tag{2.55}$$

e

$$x[n] = \left(\frac{1}{N}\right)^{\frac{1}{2}} \sum_{m=0}^{N-1} K_m \, X_c[m] \, cos\left[\frac{\pi\,(2n+1)m}{2N}\right] \qquad n = 0,1,\cdots,N-1 \tag{2.56}$$

onde

$$K_m = \begin{cases} \dfrac{1}{\sqrt{2}} & ; \quad m \neq 0 \\ 1 & ; \quad m = 0 \end{cases} \tag{2.57}$$

Pode-se mostrar que os fatores $\sqrt{\left(\frac{2}{N}\right)}$ ou $\sqrt{\left(\frac{1}{N}\right)}$ que aparecem em ambas as transformadas (direta e inversa) podem ser fundidos como (***2/N***) ou (***1/N***) e movidos para uma das transformadas. Então, outra maneira de se representar as transformadas usando esse último recurso e as Equações (2.53) e (2.54) é:

$$X_c[m] = \left(\frac{2}{N}\right) K_m \sum_{n=0}^{N-1} x(n) \, cos\left[\frac{\pi\,(2n+1)m}{2N}\right] \qquad m = 0,1,\cdots,N-1 \tag{2.58}$$

e

$$x[n] = \sum_{m=0}^{N-1} K_m \, X_c[m] \, cos\left[\frac{\pi\,(2n+1)m}{2N}\right] \qquad n = 0,1,\cdots,N-1 \tag{2.59}$$

É evidente que os termos $\boldsymbol{X_c[m]}$ e ***x[n]*** em (2.58) e (2.59) são relacionados pelo fator de escala $\sqrt{\left(\frac{2}{N}\right)}$.

Pela fusão do fator de normalização em uma das equações, a família de matrizes de transformação da ***DCT*** não é mais ortonormal. Ela, entretanto, continua ainda a ser ortogonal.

Os pares de equação (2.53, 2.54), (2.55, 2.56) e (2.58, 2.59) são observados em diferentes literaturas, e eles são utilizados na implementação dos diversos algoritmos rápidos

de implementação da ***DCT*** **[3,8,10]**, sendo que as versões (2.53) e (2.54) são as formas mais usadas na prática **[34]**.

2.5.3. Transformada Cosseno Discreta do Tipo III e IV

Existem outras formas de apresentação da DCT as quais apresentam um deslocamento de fase igual para todas as funções da base. São definidas como DCT dos tipos III e IV **[3,8-10]**. Os pares das transformadas DCT-III e DCT-IV são mostrados nas equações seguintes.

Par de DCT-III:

$$X_c[m] = \left(\frac{2}{N}\right)^{\frac{1}{2}} \sum_{n=0}^{N-1} K_n \, x[n] \cos\left[\frac{\pi (2m+1)n}{2N}\right] \qquad m = 0,1,\cdots,N-1 \tag{2.60}$$

$$x[n] = \left(\frac{2}{N}\right)^{\frac{1}{2}} K_n \sum_{m=0}^{N-1} X_c[m] \cos\left[\frac{\pi (2m+1)n}{2N}\right] \qquad n = 0,1,\cdots,N-1 \tag{2.61}$$

Observa-se que a DCT-III é a transposta da DCT-II.

Par de DCT-IV:

$$X_c[m] = \left(\frac{2}{N}\right)^{\frac{1}{2}} \sum_{n=0}^{N-1} x[n] \cos\left[\frac{\pi (2n+1)(2m+1)}{2N}\right] \quad m = 0,1,\cdots,N-1 \tag{2.62}$$

$$x[n] = \left(\frac{2}{N}\right)^{\frac{1}{2}} \sum_{m=0}^{N-1} X_c[m] \cos\left[\frac{\pi (2n+1)(2m+1)}{2N}\right] \quad n = 0,1,\cdots,N-1 \qquad (2.63)$$

Observa-se que a DCT-IV é uma versão deslocada da DCT-I.

2.5.4. Visualização do efeito da Transformada 1D

Na formulação apresentada para a ***DFT*** e para a ***DCT***, é suposto implicitamente que as seqüências ***x[n]*** e ***y[n]*** se repetem infinitamente, formando uma seqüência periódica $\tilde{x}[n]$ e $\tilde{y}[n]$, respectivamente.

No caso da ***DFT***, quando a seqüência ***x[n]*** de *N* pontos se repete para formar uma seqüência periódica infinita $\tilde{x}[n]$, sã observadas descontinuidades na seqüência $\tilde{x}[n]$, devido à junção do início e do final da seqüência *x[n]* no processo de repetição, como mostra a **Fig. 2.2**.

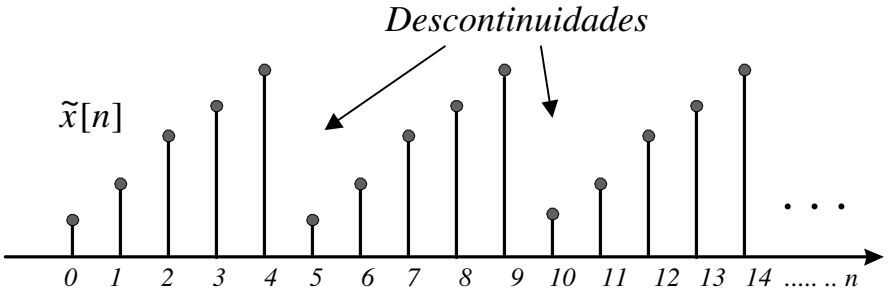


Fig. 2.2 – Seqüência periódica $\tilde{x}[n]$ obtida pela repetição de *x[n]*

Essas descontinuidades provocam o aparecimento de componentes de alta frequência com energia considerável na seqüência $\tilde{x}[n]$, que não podem ser descartadas no processo de quantização, sob pena de provocarem erros no sinal recuperado (como exemplo os efeitos de blocos em imagens). Isso dificulta sua aplicação no processo de codificação e compressão de sinais digitalizados, como no caso de imagens.

No caso da ***DCT***, quando a seqüência ***y[n]*** de comprimento ***2N*** se repete para formar a seqüência periódica $\tilde{y}[n]$, as descontinuidades mencionadas anteriormente não ocorrem, como pode ser visto na **Fig. 2.3**, ficando as transições mais suaves. A eliminação dessas descontinuidades significa que os coeficientes da ***DCT***, associados com as freqüências mais altas (k maior), tem menos energia que os coeficientes da ***DFT*** de mesma ordem.

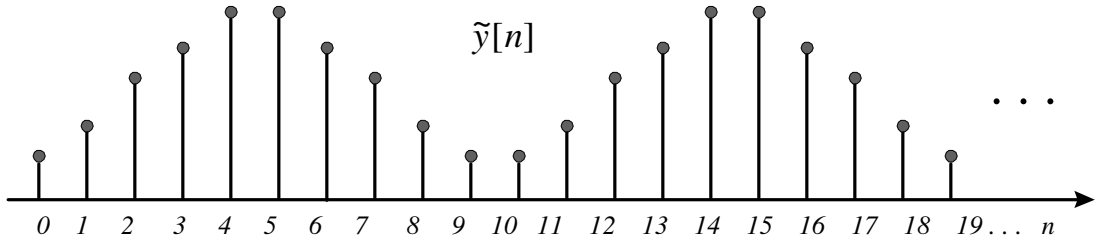


Fig. 2.3 – Seqüência periódica $\tilde{y}[n]$ obtida pela repetição de *y[n]* com *2N* pontos.

Portanto, a compactação de energia é maior no caso da ***DCT*** do que no caso da ***DFT***. Essa propriedade é muito importante e, por isso, muito explorada na codificação de imagens **[3]**. Esse fato faz com que a DCT seja uma das transformadas mais utilizadas nos métodos de codificação e compressão de imagens.

2.5.5. Transformada de Fourier Bidimensional (DFT-2D)

A Transformada Discreta de Fourier de uma seqüência bidimensional ***x[m,n]***, discreta e finita, de comprimentos [***M , N***] (***m*** = *0, 1, 2, ..., M*-1 e ***n*** = *0, 1, 2, ..., N-1*), é uma representação no domínio da freqüência em ***2D***, de ***x[m,n]***, e é definida em **[2,3,24]** como ***DFT-2D***. A *DFT-2D* é considerada uma transformada separável e sua versão corresponde à aplicação de duas transformada ***1D***, uma nas linhas e outra nas colunas da seqüência ***x[m,n]***. A ***DFT-2D*** é definida como:

$$X[k_1,k_2] = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x[n_1,n_2]\, e^{-j\frac{2\pi k_1}{N_1}n_1}\, e^{-j\frac{2\pi k_2}{N_2}n_2} \tag{2.64}$$

onde, $k_1 = 0, 1, 2, \ldots, N_1-1$ e $k_2 = 0, 1, 2, \ldots, N_2-1$

A Transformada de Fourier Discreta Inversa (***IDFT-2D***) é dada por:

$$x[n_1,n_2] = \frac{1}{N_1 N_2} \sum_{k_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} X[k_1,k_2]\, e^{j\frac{2\pi n_1}{N_1}k_1}\, e^{j\frac{2\pi n_2}{N_2}k_2} \tag{2.65}$$

onde $n_1 = 0, 1, 2, \ldots, N_1-1$ e $n_2 = 0, 1, 2, \ldots, N_2-1$

A partir das equações (2.64) e (2.65) da ***DFT-2D*** e ***IDFT-2D***, pode-se definir outras transformadas como a ***DCT-2D*** e ***DST-2D***.

2.5.6. Transformada Cosseno Discreta Bidimensional (DCT-2D)

A Transformada cosseno Discreta Bidimensional pode ser derivada da DFT-2D ou diretamente pela redução em DCT-1D.

2.5.6.1. DCT-2D via DFT-2D

Para a obtenção da DCT-2D via DFT-2D, toma-se uma seqüência bidimensional ***x[m , n]***, com $m = 0, 1, 2, ..., M-1$ e $n = 0, 1, 2, ..., N-1$, deriva-se uma nova seqüência auxiliar ***y[m , n]***, com tamanho ***2N x2M*** pontos, e calcula-se a DFT-2D de ***y[n , m]***, resultando na seqüência transformada $Y[k_1, k_2]$. A seqüência ***y[n , m]*** está relacionada à seqüência ***x[m , n]*** na forma:

$$y[m,n] = x[m,n] + x[2M-1-m,n] + x[m,2N-1-n] + x[2M-1-m,2N-1-n] \tag{2.66}$$

A DCT-2D da seqüência ***x[m , n]*** é conseguida a partir da DFT-2D da seqüência ***y[m , n]*** com os seguintes passos de processamento:

$$\underset{[M,N]}{\boldsymbol{x[m,n]}} \Rightarrow \underset{[2M,2N]}{\boldsymbol{y[m,n]}} \Rightarrow \text{DFT-2D} \Rightarrow \underset{[2M,2N]}{\boldsymbol{Y[k_1, k_2]}} \Rightarrow \text{DCT-2D} \Rightarrow \underset{[M,N]}{\boldsymbol{X[k_1, k_2]}} \qquad (2.67)$$

A recuperação da seqüência ***x[m,n]*** é conseguida com os seguintes passos de processamento:

$$\underset{[M,N]}{\boldsymbol{X[k_1, k_2]}} \Rightarrow \text{IDCT-2D} \Rightarrow \underset{[2M,2N]}{\boldsymbol{Y[k_1, k_2]}} \Rightarrow \text{IDFT-2D} \Rightarrow \underset{[2M,2N]}{\boldsymbol{y[m,n]}} \Rightarrow \underset{[M,N]}{\boldsymbol{x[m,n]}} \qquad (2.68)$$

Para a obtenção da DCT-2D da seqüência ***x[m , n]*** segundo a Equação (2.67) procede-se da seguinte forma: sobre as linhas e colunas da seqüência ***y[m , n]*** é aplicada a DFT-1D, obtendo-se assim a seqüência transformada 2D, ***Y[k₁ , k₂]***. Em seguida, toma-se [*M,N*] pontos da seqüência ***Y[k₁ , k₂]***, obtendo-se a seqüência ***X[k₁ , k₂]***, como mostrado na Equação (2.69) a seguir:

$$X[k_1,k_2] = \begin{cases} W_{2N_1}^{\frac{k_1}{2}} W_{2N_2}^{\frac{k_2}{2}} Y[k_1,k_2]; & 0 \le k_1 \le N_1 - 1, \quad 0 \le k_2 \le N_2 - 1 \\ 0 & ; \quad fora \end{cases} \qquad (2.69)$$

Seguindo o mesmo procedimento do Item ***2.5.2.***, obtém-se o par de transformadas DCT-2D para a seqüência ***x[m , n]***:

$$X[k_1,k_2] = \frac{2}{\sqrt{N_1 N_2}} \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x[n_1,n_2] \cos\left[\frac{\pi(2n_1+1)k_1}{2N_1}\right] \cos\left[\frac{\pi(2n_2+1)k_2}{2N_2}\right] \qquad (2.70)$$

com $0 \le k_1 \le N_1 - 1$, $0 \le k_2 \le N_2 - 1$, e

$$x[n_1,n_2] = \frac{2}{\sqrt{N_1 N_2}} \sum_{k_1=0}^{N_1-1} \sum_{k_2=0}^{N_2-1} \alpha_1(k_1)\alpha_2(k_2)\, X[k_1,k_2] \cos\left[\frac{\pi(2n_1+1)k_1}{2N_1}\right] \cos\left[\frac{\pi(2n_2+1)k_2}{2N_2}\right] \tag{2.71}$$

com $0 \le n_1 \le N_1 - 1$, $0 \le n_2 \le N_2 - 1$, onde:

$$\alpha_1[k_1] = \begin{cases} \frac{1}{\sqrt{2}}, & k_1 = 0 \\ 1, & 1 \le k_1 \le N_1 - 1 \end{cases} \qquad \alpha_2[k_2] = \begin{cases} \frac{1}{\sqrt{2}}, & k_2 = 0 \\ 1, & 1 \le k_2 \le N_2 - 1 \end{cases} \tag{2.72}$$

2.5.6.2. DCT-2D pela redução em DCT-1D

Seja ***[x]*** uma matriz de dimensões (***M x N***), onde ***x[m,n]*** pode ser interpretado como sendo o nível de cinza de um elemento de imagem digital (pixel) na posição (***m,n***). Sejam ***[T_M]*** e ***[T_N]*** duas matrizes reais de transformação com dimensões (***M x M***) e (***N x N***), respectivamente. A matriz transformada, bidimensional ***[X]*** da matriz ***[x]*** e sua inversa são definidas respectivamente por:

$$[X] = [T_M][x][T_N]^T \tag{2.73a}$$

e

$$[x] = [T_M]^T [X][T_N] \tag{2.73b}$$

onde:

$[T_M]^T = [T_M]^{-1}$ *transformada unitária*
$[T_N]^T = [T_N]^{-1}$ *transformada unitária*

As formas de (2.73) assumem que a transformada bidimensional pode ser implementada por uma série de transformada unidimensional. Essa propriedade é válida para qualquer seqüência separável. Uma seqüência bidimensional $v_{k1,k2}(m,n)$ é dita separável se:

$$v_{k1,k2}(m,n) = v_{k1}(m)v_{k2}(n) = v(k_1,m)v(k_2,n) \tag{2.74}$$

Fazendo ***[x]*** ser a matriz bidimensional (***M x N***) representando os dados e ***[X]*** a matriz bidimensional representando a Transformada Cosseno Discreta do tipo II (DCT-II), então, o ***k_1k_2-ésimo*** elemento da matriz ***[X]*** é dado por:

$$X(k_1,k_2) = \frac{2\alpha_1(k_1)\alpha_2(k_2)}{\sqrt{MN}} \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} x(m,n)\cos\frac{k_1\pi(2m+1)}{2M}\cos\frac{k_2\pi(2n+1)}{2N} \tag{2.75}$$

onde $k_1 = 0,1,....,M-1$, $k_2 = 0,1,....,N-1$, e

$$\alpha_1(k_1) = \begin{cases} \frac{1}{\sqrt{2}} & , \quad k_1 = 0 \\ 1 & , \quad k_1 \neq 0 \end{cases} \quad ; \quad \alpha_2(k_2) = \begin{cases} \frac{1}{\sqrt{2}} & , \quad k_2 = 0 \\ 1 & , \quad k_2 \neq 0 \end{cases} \tag{2.76}$$

De maneira similar, o ***mn-ésimo*** elemento de ***[x]*** é dado pela Transformada Inversa Cosseno Discreta do tipo II (IDCT-II) definida como:

$$x(m,n) = \frac{2}{\sqrt{MN}} \sum_{k_1=0}^{M-1}\sum_{k_2=0}^{N-1} \alpha_1(k_1)\alpha_2(k_2)X(k_1,k_2)\cos\frac{k_1\pi(2m+1)}{2M}\cos\frac{k_2\pi(2n+1)}{2N} \tag{2.77}$$

onde $m = 0,1,....,M-1$ e $n = 0,1,....,N-1$.

Usando a propriedade da separabilidade em (2.74) e fazendo ***$v(k_1,m)$*** e ***$v(k_2,n)$*** serem:

$$v(k_1,m) = \sqrt{\frac{2}{M}}\alpha_1(k_1)\cos\left(\frac{k_1\pi}{2M}(2m+1)\right) \tag{2.78}$$

e

$$v(k_2,n)=\sqrt{\frac{2}{N}}\alpha_2(k_2)\cos\left(\frac{k_2\pi}{2N}(2n+1)\right) \tag{2.79}$$

$$0\le m,k_1\le M-1;\quad 0\le n,k_2\le N-1$$

então, a DCT bidimensional como indicado em (2.73) pode ser definida como:

$$X(k_1,k_2)=\alpha_1(k_1)\left(\frac{2}{M}\right)^{\frac{1}{2}}\sum_{m=0}^{M-1}\left\{\alpha_2(k_2)\left(\frac{2}{N}\right)^{\frac{1}{2}}\sum_{n=0}^{N-1}x(m,n)\cos\frac{k_2\pi(2n+1)}{2N}\right\}\cos\frac{k_1\pi(2m+1)}{2M} \tag{2.80}$$

O somatório interno de (2.80) é uma DCT-II unidimensional de ***N*** pontos das linhas de ***[x]***, enquanto que o somatório externo representa a DCT-II unidimensional de ***M*** pontos das colunas da matriz semi-transformada. Isso implica que a DCT-II bidimensional (***M x N***) pode ser implementada por ***M DCT*** de ***N*** pontos ao longo das linhas de ***[x]*** seguida por ***N DCT*** de ***M*** pontos ao longo das colunas da matriz obtida depois das linhas transformadas. A ordem de transformação de linhas ou colunas é teoricamente irrelevante, podendo-se inverter a ordem das DCT's como ilustrado a seguir:

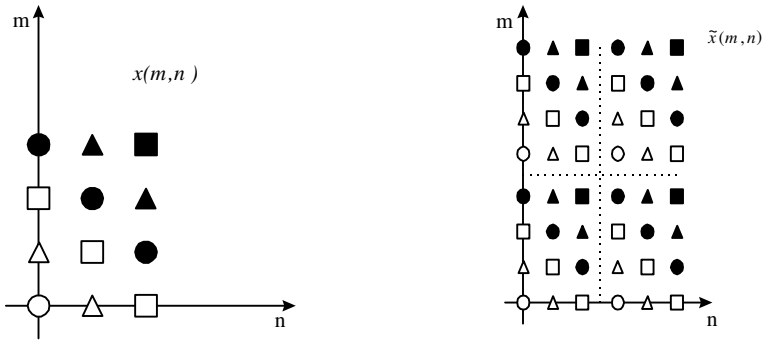
$$X(k_1,k_2)=\alpha_2(k_2)\left(\frac{2}{N}\right)^{\frac{1}{2}}\sum_{n=0}^{N-1}\left\{\alpha_1(k_1)\left(\frac{2}{M}\right)^{\frac{1}{2}}\sum_{m=0}^{M-1}x(m,n)\cos\frac{k_1\pi(2m+1)}{2M}\right\}\cos\frac{k_2\pi(2n+1)}{2N} \tag{2.81}$$

Na análise de (2.75) e (2.77), observa-se que a propriedade de separabilidade pode ser igualmente aplicada para a ***IDCT-II***. Na realidade, essa propriedade pode ser estendida para dimensões superiores a ***2***. Como um exemplo, para uma seqüência de imagens 3D, a DCT pode ser implementada por uma série de ***DCT's 1D*** ao longo de cada dimensão.

2.5.7. Visualização do efeito da Transformada 2D

Na formulação apresentada para a ***DFT-2D*** e para a ***DCT-2D***, é também suposto que as seqüências ***x[m , n]*** e ***y[m , n]*** se repetem infinitamente, formando uma seqüência periódica $\tilde{x}[m,n]$ **e** $\tilde{y}$**[m,n] com períodos***MxN*, respectivamente.

No caso da ***DFT-2D***, por exemplo, quando a seqüência ***x[m , n]*** de (*M,N*) = (*4,3*) pontos se repete para formar uma seqüência periódica infinita $\tilde{x}[m,n]$ com período *4x3*, são observadas descontinuidades na seqüência $\tilde{x}[m,n]$, devido à junção do início e do final da seqüência *x[m,n]* no processo de repetição (**Figs. 2.4a e 2.4b**).



(a) (b)

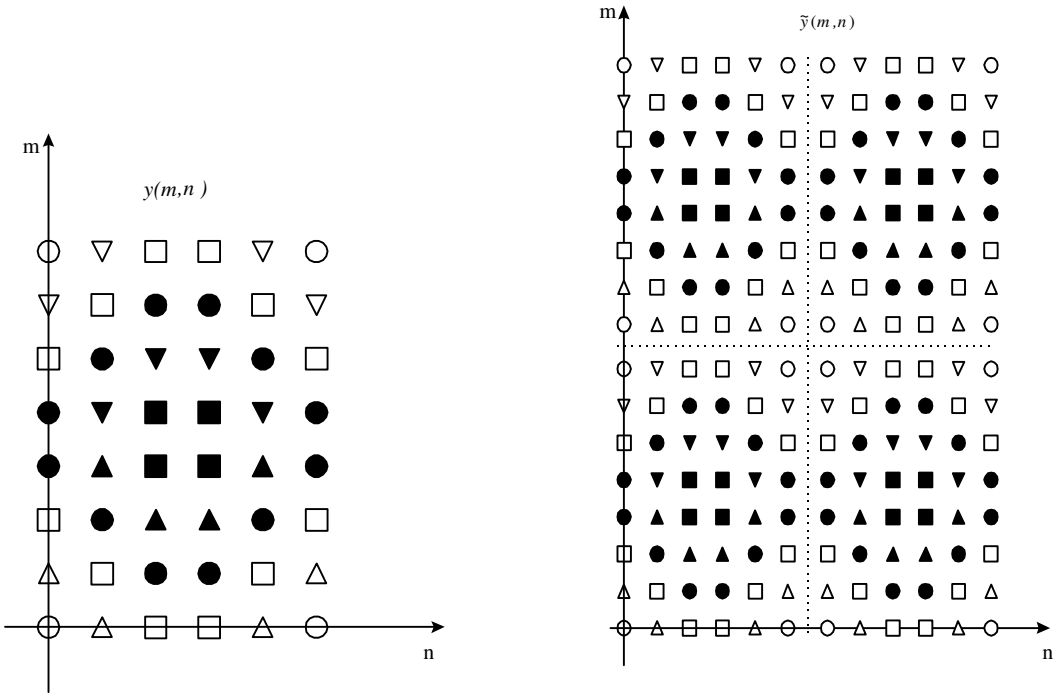
Fig. 2.4 – (a) Seqüência ***x[m,n]*** e (b) Seqüência periódica $\tilde{x}[m,n]$

Como no caso **1D**, essas descontinuidades provocam o aparecimento de componentes de alta freqüência nas direções ***m*** e ***n***, com energia considerável na seqüência $\tilde{x}[m,n]$, que não podem ser descartadas no processo de quantização.

No caso da ***DCT-2D***, a seqüência ***y[m , n]*** (**Fig. 2.5a**), é usada como passo intermediário para a obtenção da ***DCT-2D*** da seqüência ***x[m , n]***.

Quando a seqüência ***y[m,n]*** de comprimento (***2M,2N***) se repete para formar a seqüência periódica $\tilde{y}[m,n]$**, as descontinuidades mencionadas anteriormente não ocorrem, como pode ser visto naFig. 2.5b**, ficando as transições mais suaves. A eliminação dessas

descontinuidades significa que os coeficientes da ***DCT-2D***, associados às freqüências mais altas (m,n maior), têm menos energia que os coeficientes da ***DFT-2D*** de mesma ordem.



(a) (b)

Fig. 2.5: (a) Seqüência *y[m,n] = x[m,n]+x[2M-1-m,n]+x[m,2N-1-n]+x[2M-1-m,2N-1-n];*
(b) Seqüência periódica $\tilde{y}[m,n]$ obtida de *x[m,n]* e *y[m,n]*.

CAPÍTULO 3

Codificação de Imagens por Transformada DCT

3.1 Introdução

Desde 1974 **[35]**, quando a DCT-II foi desenvolvida, ela tem atraído a atenção da comunidade científica que trabalha na área de processamento digital de sinais. Desde então a DCT tem sido a transformada mais amplamente usada em codificação de imagens. Isso se deve a sua alta capacidade de compactação de energia o que a aproxima da transformada estatisticamente ótima KLT (*Karhunen-Loève Transform*) na descorrelação de sinais baseado em um processo de Markov. Sua popularidade cresceu graças ao desenvolvimento de algoritmos rápidos para a implementação da DCT envolvendo apenas aritmética com números reais, o que contribuiu com a redução da complexidade computacional desses algoritmos e com a estrutura recursiva, resultando na simplicidade de *hardware*. Como já visto, as DCTs podem ser ortogonais e separáveis. A ortogonalidade implica que a energia do sinal (da informação) é preservada no domínio da transformada. A separabilidade indica que uma DCT multidimensional pode ser implementada por uma série de DCTs unidimensionais. O benefício dessa propriedade é que algoritmos rápidos desenvolvidos para DCT-1D podem ser diretamente estendidos para DCTs multidimensionais, além de ser vantajoso do ponto de vista da simulação e da realização de *hardware*. Os comitês de padronização como **CCITT** (*International Telegraph and Telephone Consultative Committee*)**, CCIR** (*International Radio Consultative Committee*), **CMTT** (*Committee for Mixed Telephone and Television*) e **ISO** (*International Standard Organization*) recomendam a DCT como um dos principais componentes na compressão de dados para aplicações em teleconferência, videofone, transmissão progressiva de imagens, videotexto, etc. O esquema de codificação para o HIVITS (*High-quality Videofone and High-quality Television System*) projetado pela RACE (*Research in Advanced Communication Technologies for Europe*) é baseado na técnica híbrida

"*Predictive*/DCT". O objetivo da HIVITS é oferecer serviços de comunicações em multimídia variando de videofone a taxa de 64 Kbps a Televisão de Alta Definição – HDTV (*High-Defini tion Television*) a taxa no intervalo de 70 a 140 Mbps **[36].** Também, a DCT está sendo usada como uma importante ferramenta de compressão de sinais de HDTV para transmissão no intervalo de 70 a 140 Mbps **[37-40]**.

Vários codificadores baseados em DCT foram desenvolvidos para os mais diversos propósitos. Codificadores para sinais de TV NTSC (*National Television System Comittee*) com qualidade de radiodifusão com taxa de 44,736 Mbps (3^a hierarquia digital do padrão americano), codificadores para sinais de TV PAL (*Phase Alternating by Line*) a taxa de 35,368 Mbps (3^a hierarquia digital européia) **[41-45]**, codificadores de sinais de TV digital padrão 4:2:2 Rec. 601 do CCIR **[46]**, a taxas no intervalo de 32,768 a 44,736 Mbps (nível H2 da hierarquia do CCITT **[47]**) e a taxa de 15 Mbps **[48]**, etc. Todos esses codificadores mostraram resultados promissores. A DCT tem sido também usada como ferramenta principal no desenvolvimento de técnicas de compressão de dados para armazenamento e recuperação de imagens em mídias de armazenamento digital (DSM – *Digital Storage Media*) como os vídeos domésticos (VTRs – *Vídeo Tape Records*), discos óticos, CDROM (*Compact Disk Read Only Memory*), vídeo disco, dados de áudio, discos rígidos (*Winchester*), redes de telecomunicações, etc. **[49]**. Além de ser usada também como a principal ferramenta de compressão em codificação de sinais de TV em componentes Y, U e V amostradas em 10,125 MHz e 3,35 MHz respectivamente, correspondente a qualidade de vídeo na categoria 3 do CCIR 500-2 a taxa de 15 Mbps **[50]**. As aplicações em DSM são voltadas para bancos de dados de imagem de acesso aleatório, sistemas de multimídia interativos e sistemas onde vídeo, áudio, banco de dados e programas de computadores são armazenados na mesma mídia. Com o objetivo de propor um padrão de codificação para representação de imagens em movimento para DSM tendo taxa máxima de 1.5 Mbps, foi montado um grupo de especialistas em imagem em movimento MPEG (*Moving Pictures Experts Group*) sob o WG8 do SC2 do comitê técnico conjunto ISO/IEC (*International Standard Organization/International Engineering Consortium*) **[51]**. O algoritmo a ser proposto, deveria ser projetado para manter alta qualidade de vídeo e áudio durante várias operações, tais como: acesso aleatório, imagem congelada, vídeo reverso, vídeo rápido etc. Mais recentemente, a DCT tem sido usada como

ferramenta fundamental nos sistemas de codificação e compressão de imagens nos padrões JPEG (*Joint Photographic Experts Group*) e MPEG.

As aplicações da DCT incluem filtragem, filtros espelhados em quadratura, transmultiplexadores, varredura multiespectral de dados **[52]**, codificação de voz, codificação de imagens (quadro parado, seqüência de quadros monocromáticos ou em cores, armazenamento de imagens **[49]**), reconhecimento de padrões, melhoramento de imagens, codificação de imagens de radar **[53]**, análise de textura de superfície **[54]**, codificação de imagens infravermelho e codificação de imagens para impressão **[55]**, etc. Portanto, a DCT se estabeleceu como uma ferramenta significante em áreas específicas de aplicações de DSP (*Digital Signal Processing*).

Melhorias significativas com o uso da DCT na codificação de imagens foram conseguidas com modificações nas operações básicas como: uso de vários esquemas de varredura, características adaptativas, quantização diferencial, ponderação pelo sistema visual humano (HVS - *Human Visual System*), e transmissão progressiva dos coeficientes transformados. Houve também ganho de desempenho com a suplementação de outras operações tais como: predição, quantização vetorial (VQ – *Vector Quantization*), codificação por sub-banda, etc. Dessa forma, a DCT se fortaleceu firmemente no arsenal das ferramentas para DSP.

3.2. Codificação de Imagens

Embora outros aspectos do processamento de imagens, tais como filtragem, realce, classificação, reconhecimento de padrões e transmissão progressiva, sejam discutidos em outros trabalhos, o objetivo no momento é apresentar os fundamentos da codificação por transformada DCT para compressão de sinais digitais. Portanto, os objetivos incluem pesquisas que enfatizem: como uma imagem no domínio espacial, ou uma seqüência de imagens no domínio espaço-temporal, pode ser mapeada no domínio da transformada, tal que a largura de faixa para transmissão, ou capacidade de memória para armazenamento, possa ser reduzida com subsequente recuperação da imagem, ou da seqüência de imagens, por

transformada inversa, com distorção imperceptível ou desprezível. A minimização da distorção é significativa no sentido subjetivo. Embora sejam definidas várias medidas quantitativas, estas fornecem apenas um discernimento preliminar e parcial da eficiência e efetividade da codificação por transformada. Porém, o espectador humano é o último juiz relativo à qualidade das imagens processadas. Portanto, em geral, são introduzidas características adaptáveis baseadas em aspectos psicovisuais ou perceptuais, tais que as distorções das imagens recuperadas não sejam percebidas ou sejam toleráveis.

Invariavelmente, o esquema de codificação completo envolve muitas outras operações como quantização, codificação (de fonte e de canal), *buffer*, controle de *buffer*, detecção e correção de erros, encriptação, multiplexação (vídeo, áudio e dados), interfaces, vários sincronismos, sinais de alarmes e controles, conversores A/D e D/A, filtros, composição e decomposição de sinais (quando imagens e cores são codificadas na forma de componentes), etc. Essas operações constituem parte integral dos *codecs*. Outros fatores não menos significantes são: potência, tamanho, custos (comercializáveis), versatilidade, expansibilidade e compatibilidade com outros codificadores.

Na codificação digital de imagens para compressão com ou sem perdas, o objetivo é representar uma imagem como uma seqüência de dígitos binários com o menor número de bits possível, preservando os níveis de qualidade e inteligibilidade necessários para uma dada aplicação **[2, 3]**. Isso deve-se ao fato de que, em geral, a quantidade de dados de imagens digitalizadas é bastante considerável, o que acarretaria no aumento de custo para transmissão e/ou armazenamento das informações em seu estado natural (sem processamento).

Várias técnicas com diversos algoritmos são utilizadas para codificação de imagens, sendo que cada técnica explora uma característica da imagem **[3]**. Existem algoritmos que usam modelos de uma imagem, ou de parte dela, e os parâmetros desses modelos são codificados. A imagem é sintetizada a partir da recuperação desses modelos. Existem algoritmos que são implementados no domínio espacial para explorar as características de variação de intensidade do sinal e codificar essas variações. Esses codificadores são conhecidos como codificadores de forma de onda. Entre esses codificadores, destacam-se o DM (*Delta Modulation*), PCM (*Pulse Code Modulation*) e o DPCM (*Differential Pulse Code Modulation*). Existem também os algoritmos de codificadores por transformada (que é o

objeto de apreciação). Nesses algoritmos, as amostras das imagens que estão no domínio espacial são transformadas para um outro domínio por uma transformada, dando origem aos coeficientes transformados. Dentre as transformadas mais conhecidas destacam-se as: DFT (*Discrete Fourier Transform*), Hadamard, Haar, SLT (*Slant Transform*), WLT (*Wavelet Transform*), LOT (*Lapped Orthogonal Transform*), DST (*Discrete Sine Transform*), DSTr (*Discrete Sine Transform with Axis Rotation*), DCT (*Discrete Cosine Transform*) e KLT (*Karhunen-Loève Transform*).

As técnicas de codificação por transformadas têm como objetivo a redução da correlação de intensidade existente entre *pixels* vizinhos de uma imagem, eliminando ou reduzindo, dessa forma, as informações redundantes presentes na mesma. Exploram também o fato de que, nos coeficientes transformados, grande parte da energia se concentra em um pequeno número de coeficientes. Essa característica, denominada de compactação de energia, é a propriedade mais importante para uma transformada ser utilizada na codificação de imagens, visto que ela permite que uma imagem seja recuperada a partir de um pequeno número de coeficientes transformados. Essa é a razão, pela qual, a transformada DCT é uma das transformadas mais utilizadas em processamento de codificação e compressão de imagens. Outras transformadas que vêm se destacando na utilização de codificação de imagens são, por exemplo, a transformada *wavelet* **[56, 57]**, a LOT, DST e a DSTr.

Para que uma transformada possa ser utilizada na codificação de imagens, a mesma deve apresentar algumas propriedades que influirão no seu bom desempenho. Assim, dada uma seqüência bidimensional ***x[m,n]*** de tamanho ***MxN*** (que pode representar uma imagem digitalizada) e as funções bases da transformada, representadas por ***a[m,n ; u,v]*** e ***b[m,n ; u,v]***, as propriedades apresentadas pela transformada são as seguintes:

1 – LINEARIDADE - As transformadas devem ser lineares, de modo que o par de transformadas (direta e inversa) pode ser escrito na forma:

$$X[u,v] = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x[m,n]\, a[m,n;u,v] \quad ; \quad \begin{matrix} 0 \le u \le M-1, \\ 0 \le v \le N-1 \end{matrix} \tag{3.1}$$

$$x[m,n] = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} X[u,v]\, b[m,n;u,v] \quad ; \quad \begin{matrix} 0 \le m \le M-1, \\ 0 \le n \le N-1 \end{matrix} \tag{3.2}$$

onde ***X[u,v]*** em (3.1) é uma seqüência bidimensional de tamanho ***MxN***, representando os coeficientes transformados pela combinação linear da seqüência ***x[n,m]*** com as funções bases ***a[m,n ; u,v]***, e ***x[m,n]*** em (3.2) pode ser visto como uma combinação linear das funções bases ***b[m,n ; u,v]***, cujos coeficientes transformados ***X[u,v]*** representam as amplitudes dessas funções bases.

2 – SEPARABILIDADE - As funções bases devem ser separáveis, de modo que as transformadas direta e inversa possam ser aplicadas sobre as linhas e as colunas de uma seqüência bidimensional ou imagem, como se fossem transformadas unidimensionais. Reduzindo dessa forma, não só o número de operações aritméticas necessárias para se calcular a transformada, como também o esforço computacional. Portanto, (3.1) e (3.2) podem ser reescritas da seguinte forma:

$$X[u,v] = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x[m,n]\, a_1[m,u]\, a_2[n,v] \quad ; \quad \begin{matrix} 0 \le u \le M-1, \\ 0 \le v \le N-1 \end{matrix} \tag{3.3}$$

$$x[m,n] = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} X[u,v]\, b_1[m,u]\, b_2[n,v] \quad ; \quad \begin{matrix} 0 \le m \le M-1, \\ 0 \le n \le N-1 \end{matrix} \tag{3.4}$$

3 – CONCENTRAÇÃO DE ENERGIA - A transformada deve ser capaz de concentrar grande parte da energia em um número reduzido de coeficientes, de modo que se possa recuperar uma imagem a partir de desse pequeno número de coeficientes transformados, obedecendo a determinados critérios de fidelidade, que dependem da aplicação, e tendo como referência a transformada KLT (melhor transformada em termos de compactação de energia **[3]**).

4 – REDUÇÃO DE REDUNDÂNCIA - A transformada deve ser capaz de reduzir a correlação entre os coeficientes transformados.

A codificação de imagens por transformada é realizada normalmente em três etapas. No primeiro momento, a imagem sofre o processo de transformação de domínio, onde os valores de seus *pixels* são mapeados para o domínio da transformada através de uma transformação linear inversível. No segundo momento, os coeficientes transformados passam por um processo de quantização, que pode ser escalar ou vetorial. Na quantização escalar, cada coeficiente transformado é representado de maneira independente por um número de bits de acordo com uma tabela de alocação, previamente estabelecida ou determinada com base nas estatísticas dos coeficientes transformados. Na quantização vetorial, vários coeficientes (no mínimo dois) são representados de maneira conjunta por um escalar (código) que pertence a um conjunto de seqüências previamente estabelecidas e armazenadas em um dicionário denominado ***codebook***. No terceiro momento, os níveis quantizados (na saída do quantizador) são associados (representados) às palavras-código que podem ter comprimentos fixos ou variáveis, tendo sempre como objetivo a minimização do comprimento médio das palavras-código, que representam os coeficientes quantizados.

A **Fig. 3.1** mostra o diagrama de blocos simplificado de um sistema de codificação de imagens usando transformada. Na entrada do sistema, tem-se a seqüência ***x[n,n]*** que pode representar uma imagem. Essa seqüência sofre o processo de transformação tendo na saída do processo seus coeficientes transformados ***X[u,v]***. Esses coeficientes são quantizados, resultando em uma versão aproximada $\hat{X}[u,v]$ dos coeficientes ***X[u,v]***. Os coeficientes $\hat{X}[u,v]$ são posteriormente codificados, atribuindo-se uma palavra código para cada coeficiente. As palavras-código resultantes formam uma seqüência de bits que é transmitida ou armazenada. Para a recuperação de uma imagem codificada, a seqüência de bits é separada em palavras-código que são decodificadas, resultando nos coeficientes quantizados $\hat{X}[u,v]$. Sobre os coeficientes $\hat{X}[u,v]$ é aplicada a transformada inversa, resultando nas amostras $\hat{x}[m,n]$ (versão aproximada da seqüência original ***x[m,n]***).

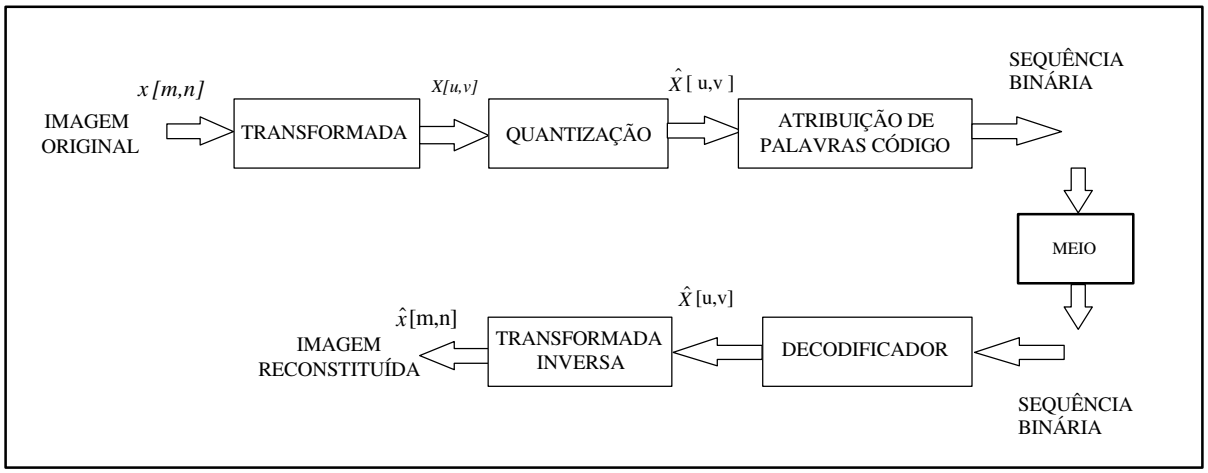


Fig. 3.1 – Diagrama de blocos simplificado de um codificador de imagens por transformada.

3.3. Transformadas de Imagens por Blocos

Em codificação de imagens por transformada, uma imagem de tamanho *C x L* geralmente é dividida em blocos de tamanho *NxN* com *N* sendo um submúltiplo inteiro de *C* e *L*, como mostrado na **Fig. 3.2**. Em geral, são usados blocos de tamanho 8x8 e 16x16. Blocos desses tamanhos possibilitam a introdução de características adaptativas baseadas na atividade ou detalhes dentro dos mesmos, além de reduzir consideravelmente a complexidade de *hardware* (tamanho de memória e lógica) e o esforço computacional, comparado com a aplicação da DCT em um quadro inteiro da imagem. Embora possa existir alguma correlação entre blocos vizinhos, isso é insignificante para justificar blocos maiores que 16x16, visto que essa propriedade não é explorada **[58,59]**. Modestino, Daut e Vickers **[60]**, através de estudos que combinam codificação de fonte e canal, concluíram que blocos de 16x16 apresentam melhor compromisso, quando se leva em conta a distribuição estatística das imagens, quantizadores e canais com ruído.

Na realidade, codificadores para videoconferência, videofone e processadores de imagens paradas foram desenvolvidos pela indústria baseados em DCT com blocos 8x8 e 16x16. Apesar do tamanho do bloco tornar mais eficiente a implementação de algoritmos

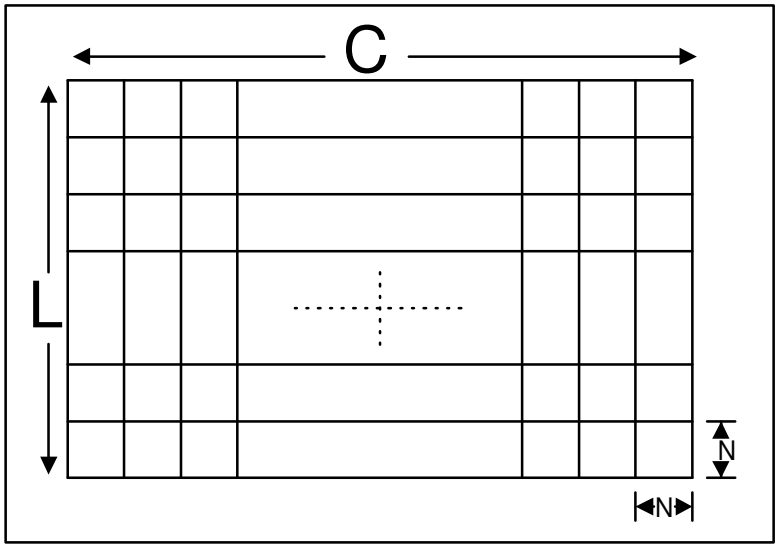


Fig. 3.2 – Uma imagem *C x L* dividida de blocos *N x N*

rápidos para computação dos coeficientes transformados, o tamanho dos blocos não pode ser reduzido indefinidamente, pois à medida que o tamanho do bloco decresce, diminui a correlação explorada entre as intensidades dos *pixels* vizinhos. Como a correlação entre os *pixels* vizinhos é uma das principais características exploradas na codificação de imagens, a redução do tamanho do bloco aumenta a correlação entre os blocos vizinhos. Como essa correlação não é explorada, o desempenho da codificação por transformada é reduzido. Portanto, os tamanhos típicos de blocos usados na codificação de imagens são 8x8 e 16x16. Entretanto, em codificação para baixas taxas de bits, o processamento por transformada em blocos pode resultar em efeito de bloqueamento nas imagens recuperadas. Para eliminar ou aliviar esses artefatos, várias técnicas têm sido desenvolvidas e apresentadas **[61-80]**. A DCT-2D com blocos de tamanho 8x8 foi recomendada pelo CCITT SG/XV **[79,81-87]** para vídeo teleconferência na taxa de *n x 384 Kbps* (*n = 0,1,..5*), e posteriormente trocada para *p x 64 Kbps* (p=1,2,...,30) **[81]**, e pelo JPEG para transmissão progressiva de fotovideotexto, quadros parados de TV, etc **[88-100]**. Também, o algoritmo de codificação para imagens paradas foi estendido para imagens em movimento, com possível transmissão em rede de pacotes **[101]** e ISDN (*Integrated Services Digital Network*) faixa estreita e faixa larga **[46]**. A proposta feita pelo grupo 1 do T1, Y1 para codificadores de sinais de TV NTSC com qualidade de

radiodifusão na taxa de 44,735 Mbps, 3ª hierarquia padrão americano também está baseada na DCT-2D com blocos 8x8 **[41,42]**.

Como a imagem ***x[C,L]*** é dividida em blocos *NxN*, em cada bloco ***x[m,n]***, ***m = 0,1,...,N*** e ***n = 0,1,...,N***, é aplicada a transformada, resultando em blocos transformados com coeficientes ***X[u,v]***. Esses coeficientes transformados passam por um processo de seleção, onde alguns coeficientes serão descartados de maneira seletiva ou prefixada, restando os coeficientes selecionados $X_s[u,v]$. Esses coeficientes são em seguida quantizados, resultando nos coeficientes $\hat{X}_{sq}[u,v]$, e posteriormente codificados, seguindo-se os mesmos procedimentos vistos anteriormente. A **Fig. 3.3** mostra o diagrama de blocos simplificado de um sistema de codificação de imagens por blocos ***NxN***, usando-se transformada DCT. Embora esse exemplo mostre o uso da DCT, esse esquema é igualmente válido para qualquer transformada ortogonal.

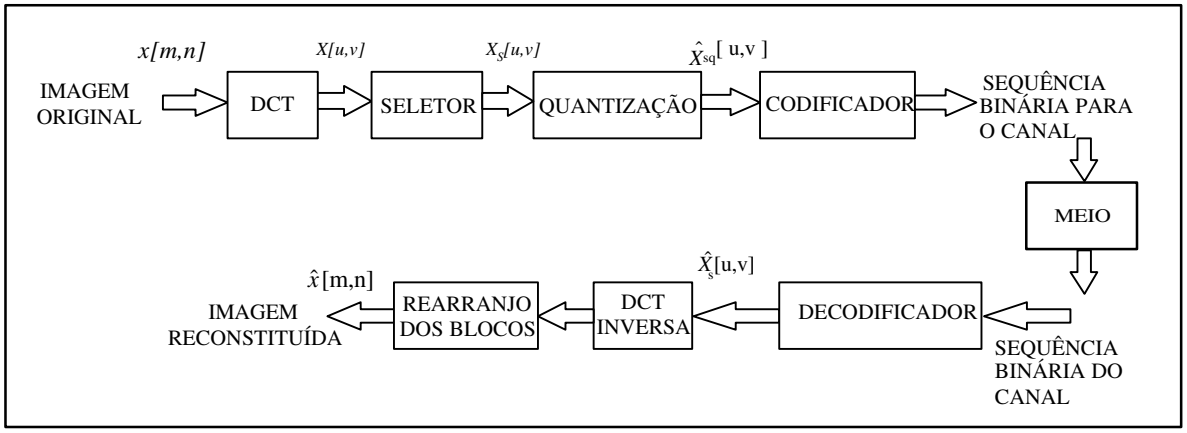


Fig. 3.3 – Diagrama simplificado de codificador de imagens por blocos usando DCT.

3.3.1. A transformada DCT

Embora a transformada e sua inversa possam ser qualquer par da família de transformadas discretas unitárias (DFT, DCT, DST, WHT, ST, HT, DHT, DLT e CMT **[102,103]**), este trabalho será limitado à discussão da DCT e, em particular, da DCT-2D tipo

II, visto que esta é a mais utilizada em aplicações de codificação de imagens por apresentar melhor desempenho, quando comparada com as versões DCT-I, DCT-III e DCT-IV. A DCT-II é aplicada em blocos de tamanho ***NxN***, sendo ***N***, em geral um submúltiplo do tamanho da imagem. Dessa forma, em termos quantitativos, as operações de codificação de imagens por transformada podem ser descritas por:

$$X(u,v) = \frac{2\alpha_1(u)\alpha_2(v)}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x(m,n) \cos\frac{u\pi(2m+1)}{2M} \cos\frac{v\pi(2n+1)}{2N} \tag{3.5}$$

onde $u = 0,1,....,M-1$, $v = 0,1,....,N-1$,

$$\alpha_1(u) = \begin{cases} \frac{1}{\sqrt{2}} & , \quad u = 0 \\ 1 & , \quad 1 \le u \le M-1 \end{cases} \quad ; \quad \alpha_2(v) = \begin{cases} \frac{1}{\sqrt{2}} & , \quad v = 0 \\ 1 & , \quad 1 \le v \le N-1 \end{cases} \tag{3.6}$$

e

$$x(m,n) = \frac{2}{\sqrt{MN}} \sum_{k_1=0}^{M-1} \sum_{k_2=0}^{N-1} \alpha_1(k_1)\alpha_2(k_2) X(k_1,k_2) \cos\frac{k_1\pi(2m+1)}{2M} \cos\frac{k_2\pi(2n+1)}{2N} \tag{3.7}$$

onde $m = 0,1,....,M-1$ e $n = 0,1,....,N-1$. Pode-se observar em (3.5) e (3.7) que as funções bases são separáveis e, portanto, a transformada é separável. Enquanto ***X[0,0]*** representa a intensidade média do bloco da imagem (componente DC), os coeficientes ***X[u,v]*** são componentes de freqüência crescente nas direções horizontal e vertical (componentes AC). A **Fig. 3.4** mostra o mapeamento de um bloco original ***NxN*** para o domínio da DCT-2D tipo-II.

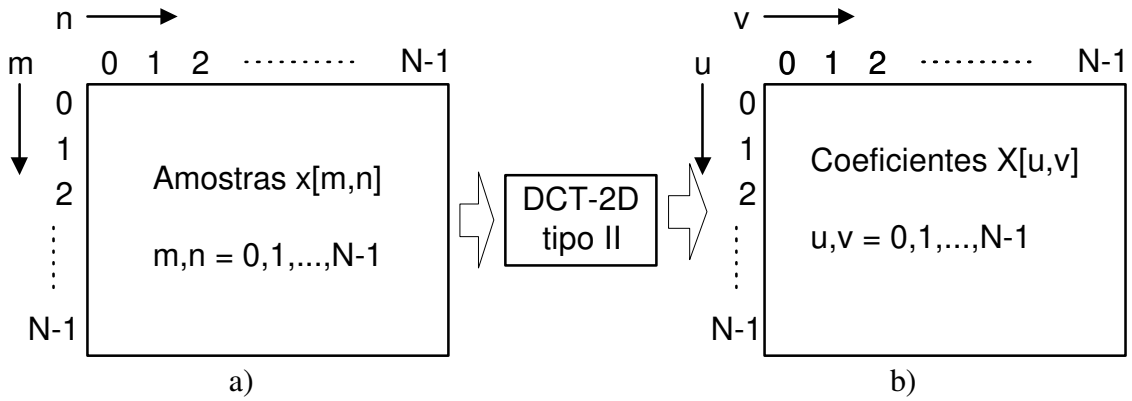


Fig. 3.4 – a) Bloco da imagem original, b) Bloco transformado

3.3.2. Seleção dos coeficientes da Transformada.

A interpretação física para o processo de seleção dos coeficientes transformados pode ser observada a partir da imagem das funções bases da DCT-2D 8x8 mostrada na **Fig. 3.5.** O bloco superior esquerdo tem intensidade uniforme, representando a intensidade média de um bloco da imagem. A progressão da esquerda para a direita representa o crescimento no número de bordas verticais. A progressão de cima para baixo representa o crescimento do número de bordas na horizontal. O bloco inferior direito representa a máxima combinação de bordas verticais e horizontais (esse bloco tem o padrão de um xadrez).

O descarte de coeficientes de alta freqüência do domínio da DCT-2D implica na eliminação das correspondentes imagens bases da imagem original. Como a DCT-2D é um processo de decomposição no mapeamento do domínio espacial em imagens bases DCT, essa decomposição estrutural pode ser utilizada na seleção adaptativa dos coeficientes transformados em bloco-por-bloco base para posterior processamento, associando cada bloco a um número finito de classes baseado na distribuição dos coeficientes. As **Figs. 3.6, 3.7** e **3.8** mostram algumas formas de seleção dos coeficientes transformados.

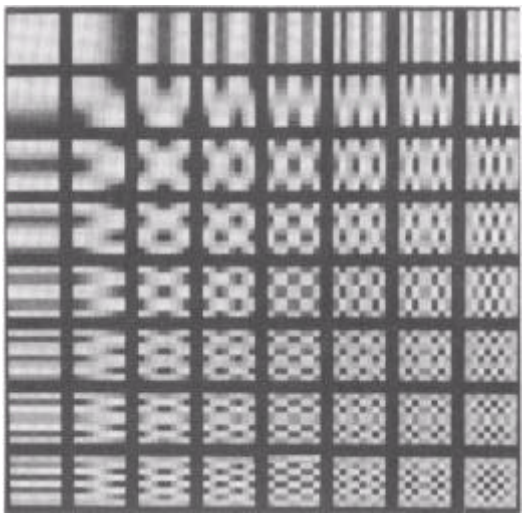


Fig. 3.5 – Imagens base 8x8 da DCT

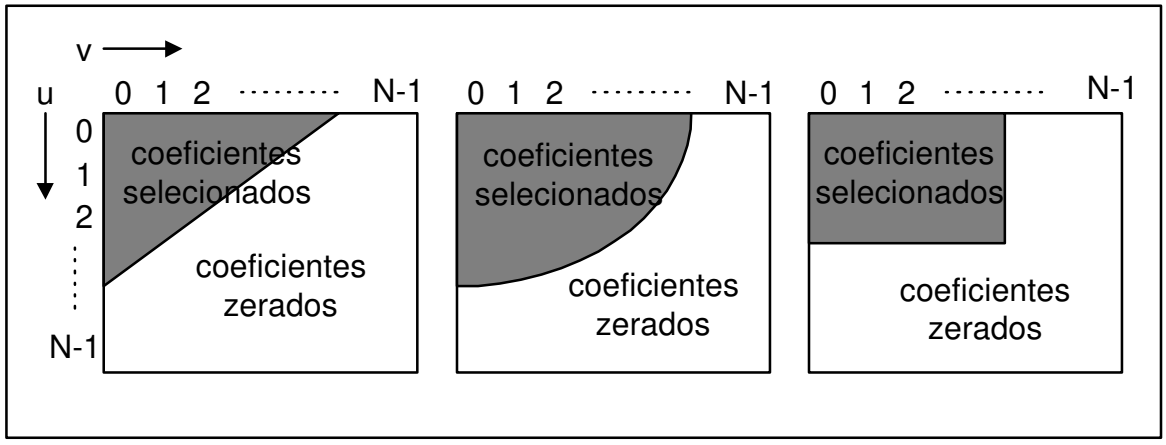


Fig. 3.6 – Amostragem por zona geométrica

No processo de seleção mostrado na **Fig. 3.6**, apenas os coeficientes dentro das zonas especificadas são processados. Se apenas coeficientes de baixas freqüências são selecionados, esse processo é equivalente a uma filtragem passa-baixas no domínio da freqüência. Algumas informações de alta freqüência, relacionadas às bordas ou aos contornos, podem ser perdidas nesse processo de seleção. A classificação por atividade baseada na estrutura dominante, mostrada na **Fig. 3.7**, foi incorporada na codificação MC DPCM/DCT (*Motion Compesation*

Differential Pulse Code Modulation/Discrete Cosine Transform) para desenvolvimento de *codecs* para teleconferência a taxas de 64 – 320Kbps **[104]**. Nesse esquema, a energia *AC* total em cada região é usada para classificar o bloco em estruturas vertical, horizontal ou diagonal. Essa classificação seguida por adicionais subdivisões de classe em cada uma dessas três regiões, juntamente com um casamento do objeto com um gabarito, respondeu efetivamente na melhoria de qualidade das imagens processadas.

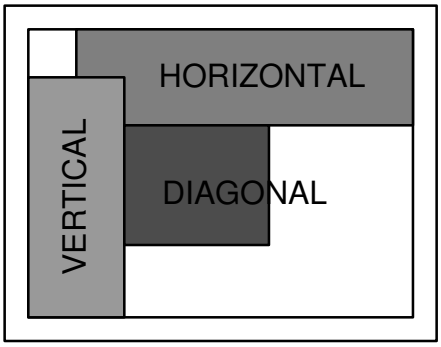


Fig. 3.7 – Classificação por regiões de acordo com a estrutura dominante no bloco

A **Fig. 3.8** mostra a distribuição de freqüência dos coeficientes e as características dos blocos que eles representam **[105]**. Na aplicação da filtragem por zona MVZS (*Maximum Variance Zonal Sampling*) **[106]** no domínio da DCT-2D, apenas aqueles coeficientes com maior variância serão posteriormente processados (os restantes serão zerados). Dessa forma, não será necessário gerar nenhum bit de *overhead* para o receptor, visto que a distribuição de variância poderá ser montada em uma tabela (*look-up table*) na memória do receptor. Entretanto, em amostragem adaptativa, onde apenas os coeficientes acima de um determinado limiar serão processados e transmitidos, serão requeridos bits de *overhead* de transmissão, caso o receptor necessite conhecer a localização dos coeficientes codificados. Embora bits de *overhead* sejam necessários para indicar a classificação do bloco, ou a localização dos coeficientes, ainda assim pode ser alcançada uma redução na taxa de bits e uma melhoria subjetiva na qualidade das imagens recuperadas.

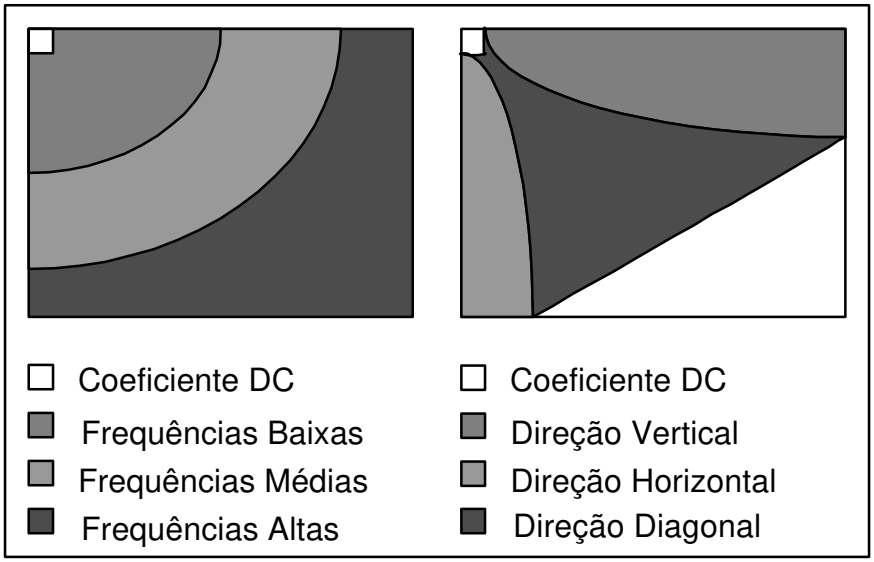


Fig. 3.8 – Distribuição de freqüência dos coeficientes DCT-2D e suas características

Tescher e Cox **[58]** mostraram que as variâncias dos coeficientes da DCT são altamente correlacionadas com a varredura em zig-zag. A varredura em zig-zag tornou-se então o mecanismo mais popular de seleção dos coeficientes, após a qual é feita a seleção daqueles coeficientes que estão acima de um determinado limiar. A **Fig. 3.9** mostra o esquema de varredura em zig-zag e a seqüência de transmissão dos coeficientes para um bloco 8x8 **[107]**.

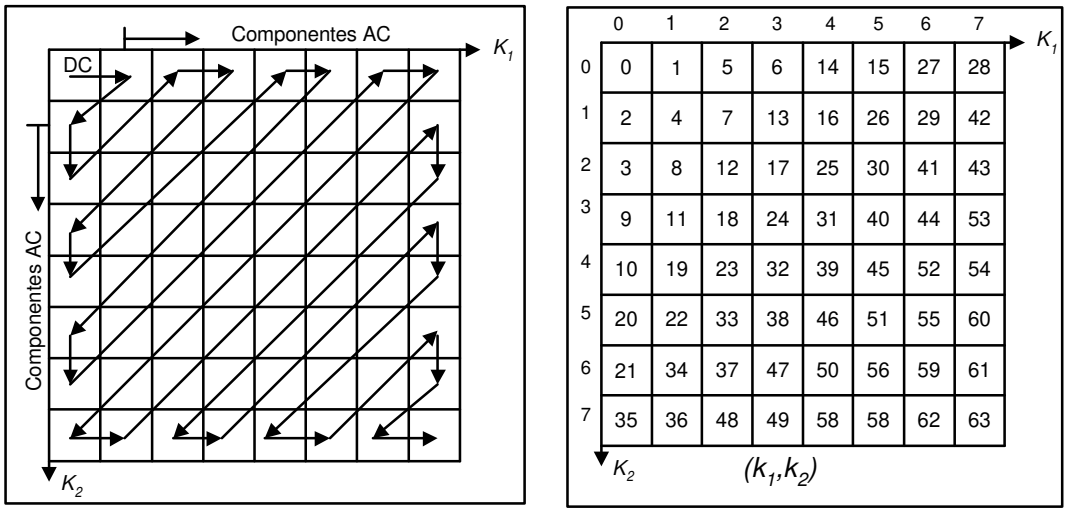


Fig. 3.9 – Varredura em zig-zag e seqüência de transmissão dos coeficientes

3.3.3. Quantização

Normalmente, os sinais que queremos processar se encontram na forma analógica, contínua, variando no tempo ou na freqüência entre dois limiares (intervalo de variação), limite inferior e limite superior. Portanto, representam quantidades escalares. Esses escalares podem assumir infinitos valores diferentes dentro desse intervalo de variação. Para viabilizar o processamento desses sinais, faz-se necessário que esses infinitos possíveis valores sejam representados por um número limitado (finito) de valores, usando um número finito de bits para representá-los. Assumindo que um total de L níveis sejam utilizados para representar o sinal contínuo x, o processo de atribuição de um dos L níveis a cada valor de x é conhecido como ***Quantização de Amplitude*** ou simplesmente ***Quantização*** **[108,109]**. Como um exemplo, tem-se uma imagem cujos níveis de cinza que representam a intensidade dos *pixels* podem assumir infinitos valores, em geral, esses níveis são representados por 8 bits, assumindo valores inteiros entre 0 e 255. Um outro exemplo, quando uma imagem passa por um processo de transformada, os coeficientes resultantes são números reais, podendo assumir uma infinidade de valores dentro de um dado intervalo. Esses coeficientes precisam ser representados por um número finito de bits, ou seja, precisam ser quantizados. Se cada escalar for quantizado individualmente, o procedimento é denominado ***Quantização Escalar***. Se dois ou mais escalares forem quantizados conjuntamente, o procedimento é denominado ***Quantização Vetorial*** ou ***Quantização por Bloco***.

3.3.3.1. Quantização Escalar

Existem vários métodos de quantização escalar, podendo ser categorizados por: Quantizadores Uniformes e Não Uniformes, Simétricos e Não Simétricos, Com Memória e Sem Memória. Na quantização Sem Memória, implica que a quantização da amostra presente é independente da história prévia do quantizador. Simétrico, implica que os níveis de entrada (decisão) e de saída (reconstrução) são simétricos para ambos os intervalos positivo e negativo. Para os quantizadores simétricos, portanto, os níveis positivos (negativos)

descrevem completamente o quantizador. O Quantizador Uniforme pode ser descrito pelo número de níveis e tamanho do passo de quantização. O Quantizador Não Uniforme é caracterizado pela variação do passo de quantização. Em geral, para pequenos valores de entrada no quantizador a quantização é fina, tornando-se grosseira à medida que os valores de entrada crescem.

Na quantização escalar sem memória, cada amostra do conjunto de dados de entrada é individualmente mapeada para um dado nível de quantização, de acordo com as características do quantizador, e em seguida, é representada por um dado número finito de bits previamente estabelecido.

Em processamento de codificação de imagens por transformadas, são utilizados alguns métodos de quantização escalar dos coeficientes transformados. Alguns desses métodos serão apresentados a seguir.

Seja um conjunto infinito $X = \{ x \in \mathbb{R} \}$ ($\mathbb{R}$ = conjunto dos reais) de amostras na entrada de um quantizador $Q[\ .\ ]$ e um conjunto finito $Y = \{ y_0, y_1, \ldots, y_{L-1} \mid y_i \in \mathbb{R} \}$ de amostras na saída do mesmo. O conjunto X é dividido em L intervalos X_i não sobrepostos $0 \leq i \leq L-1$ tal que $\{ X_i \} = (x_{i-1}, x_i)$. Os limites dos intervalos x_{i-1} e x_i são denominados de ***níveis de decisão***. Os intervalos X_i devem satisfazer as seguintes condições:

$$\bigcup_{i=0}^{L-1} X_i = X \tag{3.8}$$

e

$$X_i \cap X_j = 0 \qquad i \neq j \tag{3.9}$$

Cada intervalo X_i é associado a um nível y_i denominado de ***nível de representação*** do quantizador ou ***níveis de quantização***. As **Figs. 3.10 a), b)** e **c)** mostram alguns mapeamentos de quantizadores escalares. Os valores do eixo horizontal correspondem aos níveis de decisão $\{ X_i \}$ e os valores do eixo vertical, correspondem aos níveis de representação $\{ y_i \}$ do quantizador.

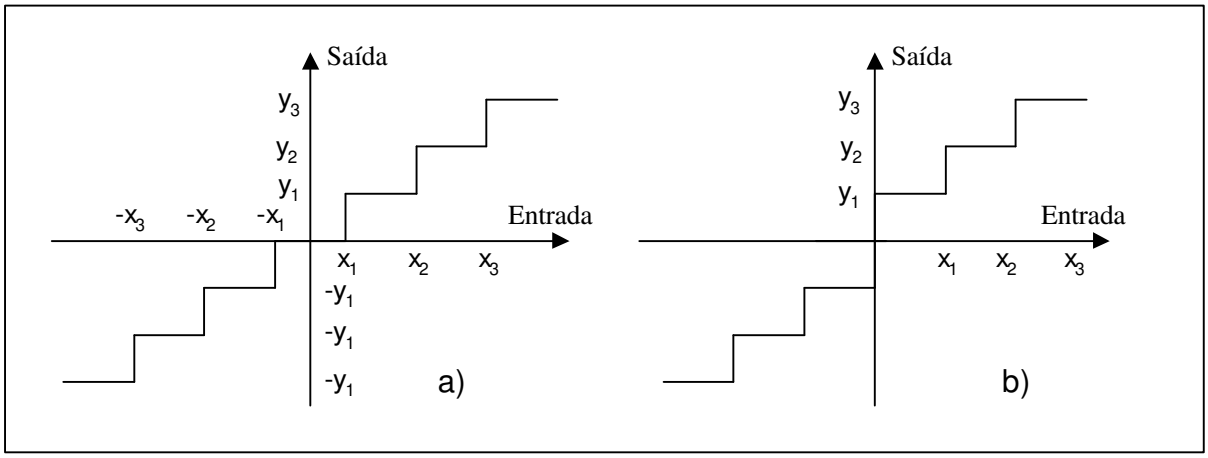


Fig. 3.10 – Quantizador Simétrico Uniforme a) *midtread*; b) *midriser*

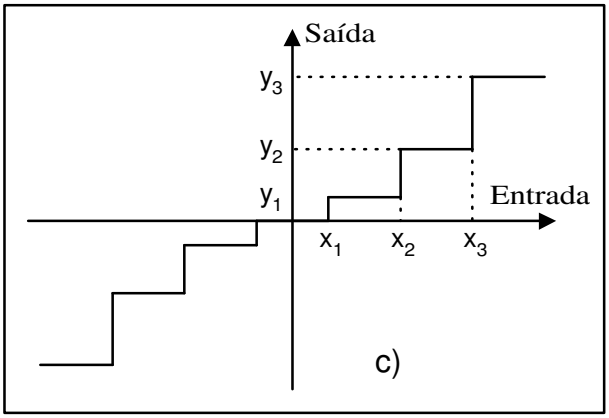


Fig. 3.10 c) – Quantizador Simétrico Não Uniforme

Seja, portanto, um quantizador $Q[\ .\ ]$ cujo sinal de entrada é denotado por x e as amostras de saída denotadas por $\hat{x}$, sendo o valor quantizado de x como mostrado na **Fig. 3.11** que representa a função de entrada-saída para um quantizador genérico. As amostras de x devem ser representadas por um dos L níveis de quantização. Se $x \in X_i$, então, por definição, $\hat{x}$ pode ser expresso por:

$$x \in \Re \ \Rightarrow \ Q[x] = \hat{x} = r_i \qquad \text{para} \qquad \begin{matrix} d_{j-1} < x \le d_j \\ i = 0, 1, \ldots, L-1 \end{matrix} \tag{3.10}$$

onde ***Q[.]*** representa a operação de quantização, r_i representa o ***i-ésimo*** nível de reconstituição, para $0 \leq i \leq L-1$, e d_j representa o ***j-ésimo*** nível de decisão, para $1 \leq j \leq L$.

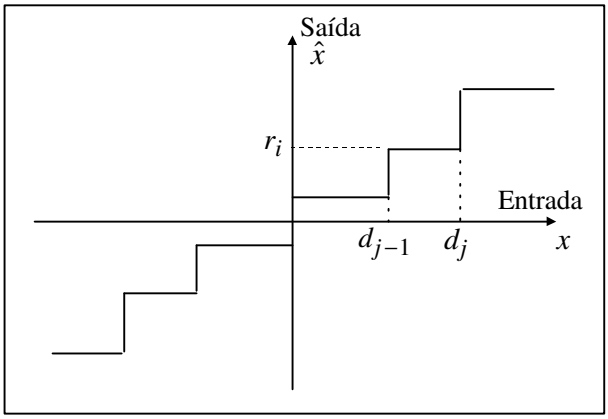


Fig. 3.11 – Relação entrada-saída de um quantizador genérico

Se o valor de entrada cair no intervalo entre d_{j-1} e d_j, o nível de reconstituição de saída será mapeado para r_i (razão pela qual a quantização é considerada um processo determinístico).

Se para representar L níveis de quantização, forem utilizados R bits em média por amostra, então pode-se dizer que:

$$L = 2^R$$

e

$$\textbf{Taxa de bits} = R = \log_2 L \quad \text{bits/amostra} \tag{3.11}$$

O desempenho de um quantizador é medido através do erro introduzido pelo processo de quantização. Esse erro é denominado de erro de quantização, ou ruído de quantização, e é definido por:

$$e_Q[n] = \hat{x}[n] - x[n] \tag{3.12}$$

então $\hat{x}$ pode ser expresso por:

$$\hat{x}[n] = Q[x] \;=\; x[n] + e_Q[n] \tag{3.13}$$

A quantidade e_Q^2 pode ser vista como um caso especial de medida de distorção $d(x, \hat{x})$, que é uma medida da distância ou discrepância entre $\boldsymbol{x}$ e $\hat{x}$. Outros exemplos que podem ser usados como medida de distorção são: $|\hat{x} - x|$ e $\left| |\hat{x}|^p - |x|^p \right|$.

Os níveis de decisão e de representação são normalmente determinados através da minimização de algum critério de erro baseado na medida da distorção $d(x, \hat{x})$, tal como a distorção média $\boldsymbol{D}$ definida por:

$$D = E[\, d(x, \hat{x})] = \int_{x_0=-\infty}^{x_0=+\infty} d(x_0, \hat{x})\, p_x(x_0)\, dx_0 \tag{3.14}$$

onde $\boldsymbol{p_x(.)}$ é a função densidade de probabilidade de $\boldsymbol{x}$ e $\boldsymbol{E[\,.\,]}$ é a esperança do argumento.

3.3.3.1.1. Quantizador Uniforme

O quantizador escalar mais simples de implementação é o quantizador uniforme. Nesse tipo de quantizador, os níveis de decisão $\boldsymbol{d_j}$ e de reconstrução $\boldsymbol{r_i}$ são uniformemente espaçados e sua função densidade de probabilidade $\boldsymbol{p_x(x_0)}$ é uniforme. Esse tipo de quantizador é definido pelas seguintes expressões:

$$\Delta \;=\; d_j - d_{j-1} \quad , \quad 1 \le j \le L \tag{3.15}$$

$$r_i = \frac{d_j + d_{j-1}}{2} \quad , \quad 0 \le i \le L-1 \tag{3.16}$$

onde Δ é o tamanho do espaçamento entre dois níveis de decisão consecutivos ou dois níveis de reconstrução consecutivos. Como um exemplo, é mostrado na Fig. 3.12 um quantizador

uniforme, de 8 níveis de quantização (R = 3 bits de representação), onde é mostrado apenas a parte positiva e $\boldsymbol{x}$ variando no intervalo ($\boldsymbol{a}$, $\boldsymbol{b}$) = (–1 , +1).

Para o projeto de um quantizador uniforme, são necessários os parâmetros: passo de quantização (Δ) e o número de intervalos ($\boldsymbol{L}$), ou os limites de variação do sinal ($\boldsymbol{a}$, $\boldsymbol{b}$) que correspondem à região de sobrecarga. Esses limites são, normalmente, determinados como múltiplos do desvio padrão (σ_x) da função densidade de probabilidade do sinal de entrada.

Supondo-se que o sinal $\boldsymbol{x}$ na entrada do quantizador seja uma variável aleatória com média $\mu_{\boldsymbol{x}}$ igual a zero (se $\boldsymbol{x}$ não tiver média nula, deve-se subtrair seus dados da média e então a entrada terá média zero), variância $\sigma_{\boldsymbol{x}}^{\boldsymbol{2}}$ e que apresenta uma função densidade de probabilidade $\boldsymbol{p_x(x)}$, a variância de $\boldsymbol{x}$ é definida por:

$$\sigma_x^{\,2} = E[\,x^2\,] - E[\,x\,]^2 = E[\,x^2\,] \qquad \textit{dado que} \quad E[\,x\,] = 0 \tag{3.17}$$

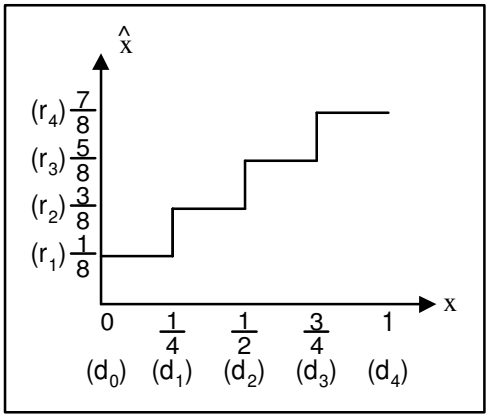


Fig. 3.12 – Quantizador uniforme de 8 níveis.

O quantizador uniforme apresenta um erro de quantização $\boldsymbol{e_Q}$ Equação (3.12), que é uma função da variável aleatória $\boldsymbol{x}$ cuja representação é mostrada na **Fig. 3.13**. A variância do erro de quantização é dada por:

$$\sigma_q^{\,2} = E[\,e_q^{\,2}\,] = \int_{-\infty}^{\infty} e_q^{\,2}\, p_q(q)\,dq \quad = \quad \int_{-\infty}^{\infty} (\hat{x} - x)^2\, p_x(x)\,dx \tag{3.18}$$

Como $\hat{x}$ é um dos $\boldsymbol{L}$ níveis de reconstrução obtido por (3.10), a integração em (3.18) é realizada nos $\boldsymbol{L}$ intervalos diferentes, correspondentes aos $\boldsymbol{L}$ níveis de reconstituição. Portanto, (3.18) pode ser escrita da forma:

$$\sigma_q^{\,2} = E[\,e_q^{\,2}\,] = \sum_{i=0}^{L-1} \int_{d_{i-1}}^{d_i} (r_i - x)^2\; p_x(x)\,dx \tag{3.19}$$

Para um sinal $\boldsymbol{x}$ variando entre os limites ($\boldsymbol{a}$, $\boldsymbol{b}$) na entrada de um quantizador uniforme, o valor do passo de quantização, conforme a **Fig. 3.12**, é dado por:

$$\Delta \;=\; x_i - x_{i-1} \;=\; \frac{(b-a)}{L} \tag{3.20}$$

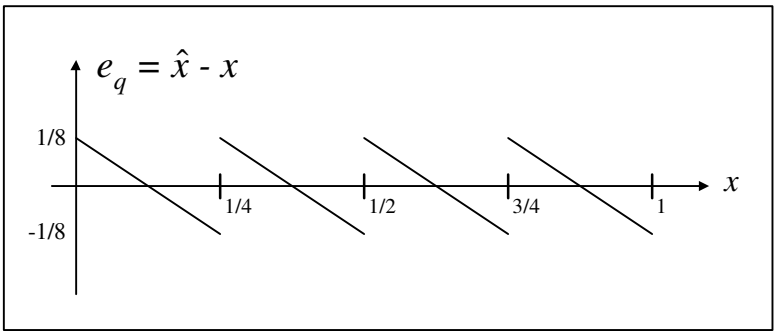


Fig. 3.13 – Erro de quantização do quantizador da Fig. 3.12.

Se forem usados $\boldsymbol{R}$ bits para representar os $\boldsymbol{L}$ níveis de quantização, então (3.20) pode ser representada por:

$$\Delta \;=\; \frac{(b-a)}{L} \quad = \quad \frac{(b-a)}{2^R} \tag{3.21}$$

Conforme **Fig. 3.13**, a distribuição do erro de quantização para esse quantizador é uniformemente distribuído no intervalo considerado. Então, o erro de quantização e sua função densidade de probabilidade são definidos pelas equações:

$$-\frac{\Delta}{2} \leq e_q \leq \frac{\Delta}{2} \tag{3.22}$$

e

$$p_q(q) = \begin{cases} \frac{1}{\Delta}, & |q| \leq \frac{\Delta}{2} \\ 0, & para\ \ outros\ \ valores \end{cases} \tag{3.23}$$

A variância σ_q^2 para o erro de quantização de um quantizador uniforme é obtida substituindo (3.23) em (3.18), considerando que o intervalo de integração é dado por (3.22). Portanto, σ_q^2 resulta em:

$$\sigma_q^2 = \frac{\Delta^2}{12} \tag{3.24}$$

Substituindo (3.21) em (3.24) obtém-se:

$$\sigma_q^2 = \frac{(b-a)^2}{12} 2^{-2R} \tag{3.25}$$

Se for considerado que $\boldsymbol{x}$ apresenta variações simétricas, onde |b| = |a|, então (3.25) resulta em:

$$\sigma_q^2 = \frac{1}{3} b^2\ 2^{-2R} \tag{3.26}$$

É observado na Equação (3.26) que o desvio padrão σ_q (valor ***RMS***) do ruído de quantização de um quantizador uniforme cresce linearmente com o passo de quantização Δ e decresce exponencialmente com a taxa R em bit/amostra.

3.3.3.1.2. Quantizador Não Uniforme

Embora o quantizador uniforme possa ser implementado de maneira direta e simples, na maioria das aplicações seu desempenho pode não ser ótimo. Por exemplo, em compressão de imagens por transformada, é muito mais provável que os coeficientes transformados $x_c(i,j)$ na entrada do quantizador se apresentem mais em uma determinada região do que em outras. A quantização uniforme, nesses casos, significa que os níveis de reconstrução que raramente ocorrem e os níveis que ocorrem com maior freqüência serão quantizados com o mesmo número de níveis. Certamente, o desempenho do quantizador poderá ser melhorado se for atribuído maior número de níveis de decisão e de reconstituição nas regiões onde houver maior ocorrência de dados (intervalos de decisão mais próximos), e menor número de níveis de decisão e de reconstituição nas regiões de menor ocorrência de dados (intervalos de decisão mais afastados), fazendo dessa forma uma quantização não uniforme como mostrado na **Fig. 3.10c**.

Como visto na **Fig. 3.10c**, nesse tipo de quantização, os níveis de decisão d_i e d_{i-1} e de reconstituição r_i e r_{i-1} não mais estão igualmente espaçados entre si, seus espaçamentos estão, portanto, relacionados com a estatística do sinal (dados) na entrada do quantizador. Entretanto, a determinação ótima dos valores dos d_i e r_i depende do critério de erro adotado. Em geral, um critério de distorção freqüentemente usado é o do *mínimo erro quadrático médio* ***MMSE*** (*Minimum Mean Square Error*).

Suponha que x é uma variável aleatória com função densidade de probabilidade $p_x(x)$. Então, o uso do critério ***MMSE*** para determinar r_k e d_k que minimizam a distorção média D resulta em:

$$D = E\left[d(\hat{x} - x)\right] = E[\, e_q^{\,2}\,] = E\left[(\hat{x} - x)^2\right] = \int_{x=-\infty}^{\infty} (\hat{x} - x)^2 \; p_x(x)\, dx \qquad (3.27)$$

Considerando que $\hat{x}$ é um dos L níveis de reconstrução obtido por (3.10), a integral em (3.27) é realizada nos L intervalos diferentes. Portanto, (3.27) pode ser escrita na forma:

$$D = E[\, e_q^2 \,] = E[\, (\hat{x} - x)^2 \,] \;=\; \sum_{i=0}^{L-1} \int_{x=d_{i-1}}^{d_i} (r_i - x)^2 \; p_x(x)\, dx \tag{3.28}$$

Para minimizar ***D***, deve ser considerado que:

$$\begin{aligned} d_0 &= -\infty \\ d_L &= \infty \\ \frac{\partial D}{\partial r_k} &= 0, \qquad 1 \le k \le L \\ \frac{\partial D}{\partial d_k} &= 0, \qquad 1 \le k \le L-1 \end{aligned} \tag{3.29}$$

De (3.28) e (3.29) obtendo-se:

$$r_k = \frac{\int_{x=d_{k-1}}^{d_k} x\, p_x(x)\, dx}{\int_{x=d_{k-1}}^{d_k} p_x(x)\, dx} \tag{3.30}$$

$$d_k = \frac{r_k + r_{k+1}}{2} \tag{3.31}$$

$$d_0 = -\infty \;\; e \;\; d_L = \infty \tag{3.32}$$

A Equação (3.30) estabelece que o nível de reconstrução r_k é o centro de massa (centróide) da função densidade de probabilidade $p_x(x)$ no intervalo $d_{k-1} \le x \le d_k$. A parte superior de (3.30) representa o momento de massa no intervalo considerado em relação à origem, e a parte inferior representa a massa total da área de $p_x(x)$ no intervalo considerado. A Equação (3.31) estabelece que os níveis de decisão d_k (exceto d_0 e d_L) são dados pelo ponto médio entre dois níveis de reconstituição r_k e r_{k+1}. As Equações (3.30) a (3.32) são necessárias para a solução ótima de certas classes de função densidade de probabilidade. Para a função

densidade de probabilidade Uniforme, Gaussiana e Laplaciana, são também suficientes. A resolução dessas equações envolve um problema não linear. Os quantizadores não lineares baseados no critério ***MMSE*** são referidos na literatura como quantizadores de Lloyd-Max, uma vez que Lloyd e Max **[110]**, independentemente, apresentaram as soluções ótimas para essas equações quando o sinal (dados) $\boldsymbol{x}$ na entrada do quantizador apresenta uma função densidade de probabilidade Uniforme, Gaussiana ou Laplaciana. A Fig. 3.14 mostra a relação entrada-saída (parte positiva) para um quantizador ótimo de Lloyd-Max de 8 níveis, quando $\boldsymbol{p_x(x)}$ é distribuição laplaciana. A parte negativa tem simetria impar em relação à origem.

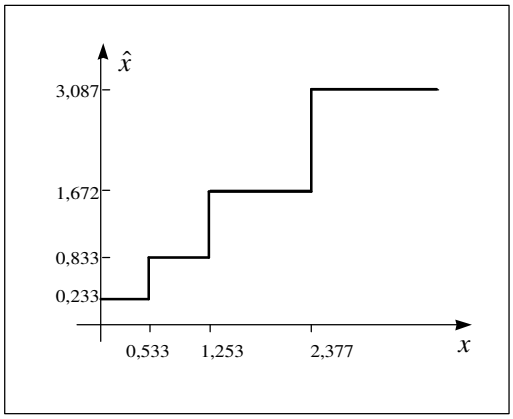


Fig. 3.14 – Relação entrada-saída de 8 níveis para um quantizador ótimo de Lloyd-Max, com distribuição Laplaciana.

No **Apêndice *A*** são mostradas várias tabelas para vários quantizadores ótimos Lloyd-Max quando na entrada do quantizador encontra-se uma variável aleatória $\boldsymbol{x}$ com funções densidades de probabilidade $\boldsymbol{p_x(x)}$ gaussiana e laplaciana, com média zero e variância unitária, para as taxas de 1 a 8 bit/amostra.

3.3.3.2. Quantização Vetorial

Na quantização vetorial, o sinal na entrada do quantizador é dividido em blocos de escalares e cada bloco é visto como uma unidade. Então, os escalares são conjuntamente quantizados na unidade. Na codificação de imagens por transformada, os coeficientes transformados são agrupados em blocos e cada bloco é conjuntamente quantizado, e representado por um escalar pertencente a um conjunto de seqüências previamente estabelecidas denominado de dicionário (*codebook*).

Fazendo **x** denotar um vetor ***N*** dimensional, $\mathbf{x} \in \mathbb{R}^N$ tal que $\mathbf{x} = [x_1, x_2, \ldots, x_N]^T$ é constituído de ***N*** valores escalares reais, de amplitude contínua. Em quantização vetorial, o quantizador ***Q[.]*** mapeia **x** para um conjunto finito ***N*** dimensional $\boldsymbol{Y} = [y_1, y_2, \ldots, y_N \mid \mathbf{y}_i \in \mathbb{R}^N]^T$ onde ***Y*** é escolhido de ***L*** possíveis níveis de reconstrução ou quantização. Portanto, fazendo $\hat{x}$ denotar o valor quantizado de **x**, então, $\hat{x}$ pode ser expresso como:

$$\hat{x} = Q[x] = y_i \quad , \quad x \in C_i \tag{3.33}$$

onde ***Q[.]*** representa a operação de quantização vetorial, $\boldsymbol{y_i}$ para $0 \leq i \leq L-1$ correspondem aos vetores de níveis de quantização (reconstrução), $\boldsymbol{C_i}$ representam as células correspondentes aos níveis de decisão, podendo ser interpretadas como polígonos sólidos no espaço ***N*** dimensional $\mathbb{R}^N$. Se **x** se encontra na célula $\boldsymbol{C_i}$, **x** é mapeado para $\boldsymbol{y_i}$. A **Fig. 3.15** mostra um exemplo de quantização vetorial para $\boldsymbol{N=2}$ e $\boldsymbol{L=9}$ **[38]**.

Como na quantização escalar, o desempenho do quantizador é realizado por uma medida de distorção $\boldsymbol{d(\mathbf{x}, \hat{x})}$, que indica o grau de discrepância entre **x** e $\hat{x}$. O critério de fidelidade ou de distorção adotado na medida é realizado de maneira indireta. Assim, $\mathbf{x} \in \boldsymbol{C_i}$ se a distorção de **x** em relação a $\mathbf{y}_i$ for menor ou igual à distorção em relação a qualquer $\mathbf{y}_j$ para $\boldsymbol{i \neq j}$. Essa análise pode ser representada como:

$$x \in C_i \quad \Rightarrow \quad Q[x] = y_i \quad \Leftrightarrow \quad d(x, y_i) \leq d(x, y_j) \quad , \quad i, j = 0,1,\ldots,L-1 \tag{3.34}$$

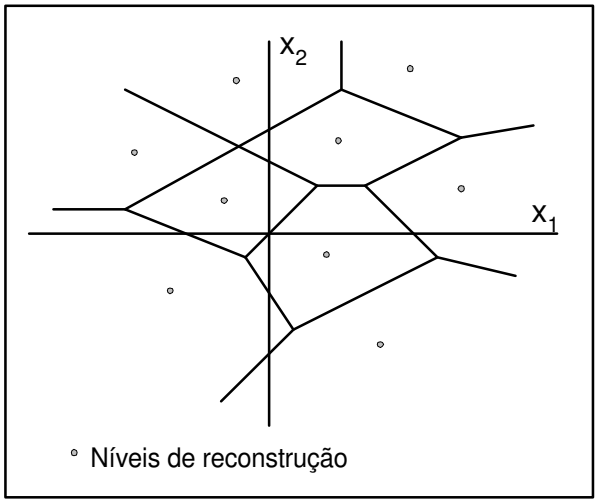


Fig. 3.15 – Quantização vetorial com 2 escalares por vetor e 9 níveis de reconstrução.

Quando a menor aproximação é encontrada, o índice $\boldsymbol{i}$ correspondente a essa aproximação (representando o vetor ótimo) é codificado e transmitido. No receptor, $\mathbf{y}_i$ pode ser reconstruído simplesmente procurando no *codebook* o número $\boldsymbol{i}$ da célula correspondente. Codificações desse tipo (quando $\boldsymbol{i}$ é codificado diretamente) apresentam uma taxa de bits/amostra da ordem de $\frac{1}{N}\log_2 L$.

A codificação vetorial apresenta resultados superiores em relação a quantização escalar. Porém, aumenta significativamente a complexidade do sistema, visto que faz-se necessária a implementação de cálculos para a construção do *codebook*, o que torna o sistema mais pesado. Melhores detalhes podem ser encontrados em diversas literaturas sobre o assunto **[111]**. A quantização escalar é a mais usada porque é mais simples de ser implementada, apesar da quantização vetorial apresentar um melhor desempenho.

3.3.4. Projeto do Quantizador e Alocação de Bits para os Coeficientes Transformados

A base para a compressão de fonte em codificadores baseados na DCT é o quantizador. A quantização é uma das operações básicas fundamentais da codificação de imagens para a

compressão. Na codificação por transformada, a imagem de tamanho ***LxC*** é dividida em blocos ***MxN*** (onde *M* e *N* são submúltiplos de *L* e *C*, respectivamente) e a transformada é aplicada sobre esses blocos, extraindo as redundâncias existentes entre os *pixels* vizinhos dentro de cada bloco, gerando coeficientes transformados com diferentes variâncias. Cada coeficiente transformado deverá ser quantizado individualmente. Portanto, é requerida uma matriz de quantização com o mesmo tamanho do bloco usado na transformada. Cada elemento da matriz de quantização poderá ter qualquer valor entre 1 e 255, definindo os passos de quantização correspondentes aos coeficientes da DCT. Uma matriz de quantização com todos os elementos iguais a 1 resulta em uma compressão quase sem perda (toda perda será devido ao erro de arredondamento). Em geral, quanto maiores forem os elementos, maior será a perda e melhor será a compressão obtida.

Uma quantização agressiva aumentará a compressão, porém, poderá introduzir artefatos na imagem (tal como o efeito de bloqueamento) e causar uma distorção na imagem recuperada. Assim, uma boa matriz de quantização deverá ter o balanço entre a necessidade de compressão e uma boa qualidade da imagem recuperada. As técnicas para o projeto do quantizador podem ser divididas em duas classes: 1 – aquelas que são baseadas na percepção humana e experimentos psicovisuais e 2 – aquelas que são baseadas na teoria taxa-distorção e controle da taxa de bits.

3.3.4.1. Técnicas baseadas em experimentos psicovisuais

As técnicas baseadas na percepção humana definem os elementos da matriz de quantização pelos limiares de visibilidade para as funções bases da DCT, tal que qualquer perda devido à quantização seja perceptualmente irrelevante. Esses limiares de visibilidade são determinados por uma série de experimentos psicovisuais que pode ser descritos analiticamente por:

$$u_r = u_o + \gamma\, u_t \tag{3.35}$$

onde u_o é uma imagem uniforme de fundo, u_t uma função estimulante, γ uma variável e u_r uma imagem visível resultante. O teste de percepção consiste em determinar um valor para γ tal que torne visível a diferença entre u_o e u_t. Na prática, as funções bases da DCT são superpostas a um fundo uniforme e a intensidade γ destas funções são aumentadas até tornarem-se visíveis. Lohscheller **[112]** foi um dos primeiros a determinar esses limiares para as funções bases da DCT. Seus trabalhos tiveram como resultado a definição das matrizes de quantização que são usadas no JPEG, listadas no **Anexo K** desse padrão e mostradas nas **Figs. 3.16a)** e **b)**.

A matriz da **Fig. 3.16a)** pode ser usada em imagens com níveis de cinza ou em imagens de componentes de luminância. Já a matriz da **Fig. 3.16b)** pode ser usada em imagens de componentes de crominância. O uso dessas matrizes é opcional. Entretanto, experimentos têm mostrado que tal uso é muito robusto e aplicável para um amplo conjunto de imagens. A qualidade da imagem recuperada ou a taxa de compressão pode não ser satisfatória para certas aplicações. Nesses casos, pode-se buscar um balanço entre a qualidade da imagem recuperada e a taxa de compressão, escalonando uniformemente os elementos da matriz original por um fator de escala denominado *fator de qualidade* q.

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Matriz de quantização para luminância
(a)

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Matriz de quantização para crominância
(b)

Fig. 3.16 – Matrizes de quantização usadas no JPEG (Anexo K):
a) para Luminância e b) para Crominância

Considerando que ***x[m,n]*** são os elementos do sinal original e ***X[u,v]*** os coeficientes transformados, então, os coeficientes quantizados resultantes $X_q[u,v]$ são calculados pela expressão:

$$X_q[u,v] = \left\lfloor \frac{X[u,v]}{Q[u,v]} \right\rfloor \cong \left\lfloor \frac{X[u,v] + \left\lfloor \frac{Q[u,v]}{2} \right\rfloor}{Q[u,v]} \right\rfloor \tag{3.36}$$

onde $\lfloor \, . \, \rfloor$ indica o maior inteiro do argumento. Outras matrizes podem também ser usadas para a quantização dos coeficientes transformados.

As **Figs. 3.17a)** e **b)** mostram as matrizes usadas no padrão MPEG-2 para a quantização de macroblocos intracodificados e macroblocos preditos e/ou interpolados, respectivamente.

8	16	19	22	26	27	29	34
16	16	22	24	27	29	34	37
19	22	26	27	29	34	34	38
22	22	26	27	29	34	37	40
22	26	27	29	32	35	40	48
26	27	29	32	35	40	48	58
26	27	29	34	38	46	56	69
27	29	35	38	46	56	69	83

Matriz de quantização para macroblocos intracodificados
(a)

16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16

Matriz de quantização para macroblocos preditos/interpolados
(b)

Fig. 3.17 – Matrizes de quantização usadas no MPEG-2
a) para macroblocos intracodificados
b) para macroblocos preditos

3.3.4.2. Técnica baseada em controle da taxa de bits

As técnicas baseadas em controle de taxa de bits geram matrizes de quantização cujos elementos são baseados nas propriedades estatísticas da imagem. Embora os resultados da teoria da percepção humana possam influenciar os resultados finais do projeto, a ênfase, entretanto, é o controle da taxa de bits.

Após a imagem ter sido passada completamente pelo processo de transformação, tem-se como resultado uma imagem ***LxC*** com ***B*** blocos ***MxN*** de coeficientes transformados, sendo ***B*** determinado por:

$$B = \left(\frac{L}{M}\right) x \left(\frac{C}{N}\right) \tag{3.37}$$

Para se obter a compressão da imagem transformada, define-se ***R*** como sendo a taxa média de bits de compressão requerida (essa taxa é considerada antes do codificador de *Huffman*). Se forem alocados ***b[i,j]*** bits por coeficiente DCT quantizado, então, ***R*** será dado por:

$$R = \frac{1}{MxN} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} b[\,i, j\,] \tag{3.38}$$

Considera-se que a DCT tem a propriedade de compactação de energia, onde alguns coeficientes transformados retêm maior energia que outros. Considera-se também que a imagem transformada é constituída de ***B*** blocos, sendo que cada bloco é constituído de ***y[i,j]*** coeficientes transformados. Além disso, considera-se que cada bloco transformado tem comportamento diferente, dependendo de sua posição dentro da imagem. A questão é: como alocar os bits para cada coeficiente dentro do bloco transformado, visto que a taxa ***R*** de bits/coeficiente é fixada para todos os blocos? Certamente, os coeficientes que retém maior energia deverão ser contemplados com um número maior de bits, e os de menor energia com um número menor de bits. A solução para esse impasse é usar a variância $\sigma[i,j]^2$ de cada

coeficiente transformado como parâmetro de alocação do número de bits para cada coeficiente. A variância $\sigma[i,j]^2$ de cada coeficiente transformado é obtida pela expressão:

$$\sigma^2[i,j] = Var(y[i,j]) = \frac{1}{B}\sum_{k=0}^{B-1}\{y_k[i,j] - \mu[i,j]\}^2 \tag{3.39}$$

onde $y_k[i,j]$ denota o coeficiente $[i,j]$ da DCT do ***k-ésimo*** bloco, e $\mu[i,j]$ denota a média desse coeficiente, dada por:

$$\mu[i,j] = mean(y[i,j]) = \frac{1}{B}\sum_{k=0}^{B-1} y_k[i,j] \tag{3.40}$$

Deve-se, portanto, determinar o número de bits ***b[i,j]*** para cada coeficiente transformado de modo que cada bloco satisfaça a seguinte condição:

$$\sum_{i=0}^{M-1}\sum_{j=0}^{N-1} b(i,j) = (MxN)R \tag{3.41}$$

A idéia chave do método de controle da taxa de bits é alocar maior número de bits para os coeficientes transformados que apresentam maior variância. Essa estratégia visa minimizar o erro quadrático médio. Da teoria da taxa de distorção, a solução ótima de alocação de bits para os coeficientes é dada por:

$$b[i,j] = \alpha + \frac{1}{2}log_2\frac{\sigma^2[i,j]}{\sigma_g^2} \quad , \quad \begin{matrix} i = 0,1,2,\cdots,M-1 \\ j = 0,1,2,\cdots,N-1 \end{matrix} \tag{3.42}$$

Essa solução é conhecida como regra do log-variância **[39, 43]**, onde $\sigma^2[i,j]$ é a variância do coeficiente transformado ***y[i,j]***, e σ_g^2 é a média geométrica das variâncias dos coeficientes transformados, dada por:

$$\sigma_g^2 = \left(\prod_{i=0}^{M-1} \prod_{j=0}^{N-1} \sigma^2[i,j] \right)^{1/(MxN)} \tag{3.43}$$

Na Equação (3.43), α é um multiplicador de *Lagrange*, cuja escolha deve ser tal que satisfaça as condições de (3.41). Quando as taxas são relativamente altas, α pode ser escolhido aproximado por ***R***. Em (3.42), se ocorrer de ***b[i,j]*** ser negativo para algum coeficiente transformado ***y[i,j],*** então nenhum bit será alocado a esse coeficiente, isto é, ***b[i,j] = 0***. Como resultado desse processo, é montada uma matriz de alocação de bits para a quantização dos coeficientes transformados. Nesse caso, se for realizada uma quantização escalar baseada em *Lloyd-Max*, então, essa matriz será usada para quantizar individualmente cada coeficiente, de acordo com o número de bits alocados para aquela posição, obtendo-se, assim, uma imagem transformada e quantizada, para uma taxa média de *R* bits/amostra.

Conforme as **Figs. 3.1** e **3.3**, as amostras quantizadas passarão em seguida por um processo de codificação, onde é realizado (através de mapeamento) uma correspondência de cada amostra quantizada a uma palavra código correspondente (que pode ter tamanho fixo ou variável), pertencente a um dicionário de códigos. O objetivo desse mapeamento é reduzir tanto quanto possível o número médio de bits necessários para representar uma imagem codificada. Isso é conseguido utilizando-se códigos que extraem as redundâncias presentes na imagem. O método mais comumente usado para esse tipo de compressão é denominado de *"codificação por entropia"*. D. A. Huffman em 1952 desenvolveu um método eficiente de construção de códigos por entropia, que hoje é denominado pela literatura como *"código de Huffman"*. Outros métodos de codificação são também usados como: *Lempel-Ziv*, Aritméticos, etc.

Após a imagem ter passado pelo processo de codificação, os dados serão então armazenados para posterior utilização ou transmitidos via algum meio para um ponto distante. Em ambos os casos, para reconstruir a imagem, é necessário que a mesma passe por um processo inverso ao sofrido quando de sua codificação, como mostrado nas **Figs. 3.1** e **3.3**. A imagem recuperada é uma réplica da imagem original. O desempenho do sistema usado para a

codificação da imagem é analisado através de avaliações objetivas e subjetivas. As objetivas são realizadas através de medidas de distorção entre as imagens original e recuperada, em geral, pela relação sinal-ruído SNR (*Signal-to-Noise Ratio*) e pela relação sinal-ruído de pico PSNR (*Peak Signal-to-Noise Ratio*). As subjetivas são realizadas através de avaliações por parte de observadores (em geral peritos) da qualidade visual da imagem recuperada com relação à imagem original.

CAPÍTULO 4

Aplicações da DCT em Compressão de Imagens

4.1 Introdução

Como mencionado em capítulos anteriores, a DCT pode ser suplementada ou complementada com outros métodos de redução de redundância, tais como: DPCM, codificação por Sub-Banda, Quantização Vetorial ***(VQ)***, etc. Outras características adicionais podem também ser incorporadas em todo o processo de codificação, tais como: compensação de movimento, preditores e/ou quantizadores chaveados, ponderação por HVS, múltiplos processos de varredura, onde são priorizados os coeficientes transformados, etc. A compressão de dados em aplicações de codificação de voz, imagens, imagens multi-espectrais, imagens coloridas para impressão, codificação de textura, etc, usando quantização vetorial, tornou-se possível graças ao algoritmo rápido e eficiente de construção de *codebook* proposto por Lind, Buzo e Gray **[113]**, popularmente conhecido como algoritmo ***LBG***. Posteriormente, várias modificações e melhorias foram propostas **[68,114-119]**, como em outras técnicas híbridas de codificação, tais como: DCT/DPCM, DCT/Sub-Banda, etc. O objetivo dessas diversas variações de VQ, em geral, é reduzir o tamanho da memória e/ou a complexidade da codificação da VQ básica ou da VQ adaptativa, baseada nas propriedades do sinal de entrada (local ou global).

4.2. DCT/VQ

O casamento da DCT com a quantização vetorial (***DCT/VQ***) tem como objetivo suavizar suas desvantagens e ao mesmo tempo reforçar seus pontos fortes. A combinação

DCT/VQ apresenta hoje muitas variações, adaptações e opções, como também existe para outras técnicas híbridas.

O processo de codificação por quantização vetorial envolve os seguintes passos:

1 – Desenvolvimento do *codebook* baseado em uma grande seqüência de treinamento que seja representativa da seqüência de teste;

2 – Formatação do sinal de entrada como um vetor de entrada;

3 – Pesquisa pelo vetor código mais próximo dentro do *codebook*, que seja mais representativo do vetor de entrada. O vetor código selecionado deve ter a menor distorção ou a melhor correspondência com o vetor de entrada. A distorção deve ser pré-definida ;

4 – Geração de um rótulo para o vetor código selecionado sobre o canal de comunicação (dito código binário).

A dimensão do vetor (número de elementos no vetor de entrada), tamanho do *codebook* (número de vetores códigos no *codebook*), critério de distorção, algoritmo de pesquisa (para alocar a melhor correspondência entre o vetor de entrada e o vetor código) e como o *codebook* é dividido são algumas das variáveis que devem ser consideradas no projeto de VQ. Assume-se que o vetor de entrada é de dimensão $\boldsymbol{K}$, dado por $\mathbf{x} = \{x_1, x_2, ..., x_K\}$, e o *codebook* é de tamanho $\boldsymbol{M}$, cujos membros são dados por: $\boldsymbol{y1, y2, ..., y_M}$. A taxa de bit por vetor é dada por $\boldsymbol{\log_2 M}$ e a taxa de bit por componente é dada por $\boldsymbol{R = (\log_2 M)/K}$.

Para obter alto desempenho em VQ, o vetor representativo $\boldsymbol{y_i}$ selecionado deve ser o mais próximo possível de **x**, o tamanho do *codebook* deve ser grande e a dimensão do vetor de entrada deve ser grande o suficiente para que a correlação entre vetores adjacentes seja mínima. O aumento do tamanho do *codebook* e/ou da dimensão do vetor de entrada implica em um grande aumento da capacidade de memória e em um crescimento exponencial na complexidade computacional. Para propósitos práticos, tanto $\boldsymbol{M}$ como $\boldsymbol{K}$ devem ser finitos. Para aplicações em processamento de imagens são usados: $\boldsymbol{M}$ = (64, 128, 256 e 512) e $\boldsymbol{K}$ = (4,9,16 e 25). Para determinar a escolha de $\boldsymbol{y_i}$, foram definidas várias medidas de distorção (não negativas). As mais significantes são: MSE (*Mean Square Error*), MAE (*Mean Absolute Error*), $\boldsymbol{l_n}$ ou *Holder Norm*, ***nth*** *Law Distortion*, *Weighted Square Distortion*, *General*

Quadratic Distortion, etc **[119]**. A definição de algumas dessas medidas são apresentadas a seguir:

$$\textbf{MSE =>}\ d(x, y_i) = \frac{1}{K}\sum_{m=1}^{K}(x_m - y_{im})^2 \tag{4.1}$$

$$\textbf{MAE =>}\ d(x, y_i) = \frac{1}{K}\sum_{m=1}^{K}|x_m - y_{im}| \tag{4.2}$$

$$\textbf{Holder Norm (} l_n \textbf{) =>}\ d(x, y_i) = \|x - y_i\|_n = \left[\sum_{m=1}^{K}|x_m - y_{im}|^n\right]^{\frac{1}{n}} \tag{4.3}$$

$$\textbf{nth Law Distortion =>}\ d(x, y_i) = \left(\|x - y_i\|_n\right)^n = \sum_{m=1}^{K}|x_m - y_{im}|^n \tag{4.4}$$

$$\textbf{Weighted Square Distortion =>}\ d(x, y_i) = \sum_{m=1}^{K} w_m (x_m - y_{im})^2 \tag{4.5}$$

$$\textbf{General Quadratic Distortion =>}\ d(x, y_i) = (x - y_i)[W](x - y_i)^T \tag{4.6}$$

Nas Equações (4.1) a (4.6), **x** é assumido ser real, $w_1, w_2, \ldots, w_k$ em (4.5) são os pesos e ***[W]*** em (4.6) é uma matriz simétrica positiva ***KxK*** e $y_i = \{ y_{i1}, y_{i2}, \ldots, y_{ik} \}$ é o i-ésimo membro do alfabeto (*codebook*). Esses requisitos garantem que a distorção seja não negativa. A escolha apropriada de ***[W]*** em (4.6) permite que as propriedades perceptíveis da imagem (ou voz) possam ser incorporadas no codificador VQ.

A **Fig. 4.1** mostra o esquema básico de um codificador VQ.

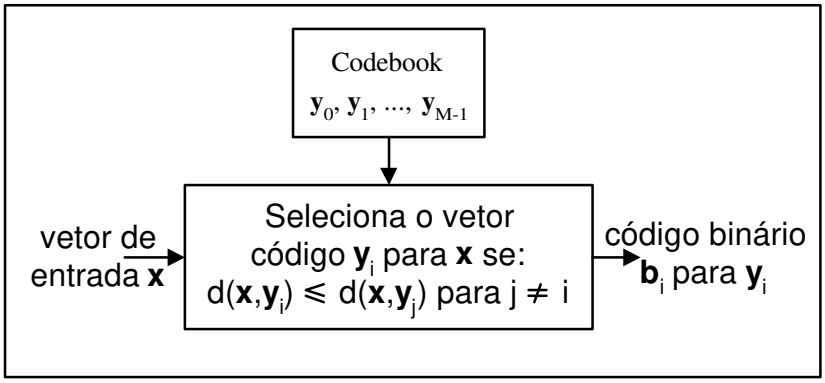


Fig. 4.1 – Codificador Vetorial básico

Uma combinação possível DCT/VQ é aplicar a DCT sobre as linhas (ou colunas) dos blocos de uma imagem, seguida da aplicação da VQ sobre as colunas (ou linhas) dos blocos semitransformados **[120]**. Outra variação da combinação DCT/VQ é usada em seqüência de quadros onde a DCT-2D é aplicada sobre os blocos de cada quadro e a VQ é aplicada ao longo da direção temporal dos quadros sucessivos. Essa combinação é caracterizada por uma combinação DCT/VQ de um arranjo 3D obtida via DCT-2D de cada quadro. A **Fig. 4.2** ilustra essa combinação.

Na **Fig. 4.2** é mostrado que a DCT-2D é aplicada sobre os blocos de cada quadro e a quantização vetorial VQ é aplicada sobre os ***M x N*** vetores, cada vetor composto dos elementos transformados dos ***K*** quadros.

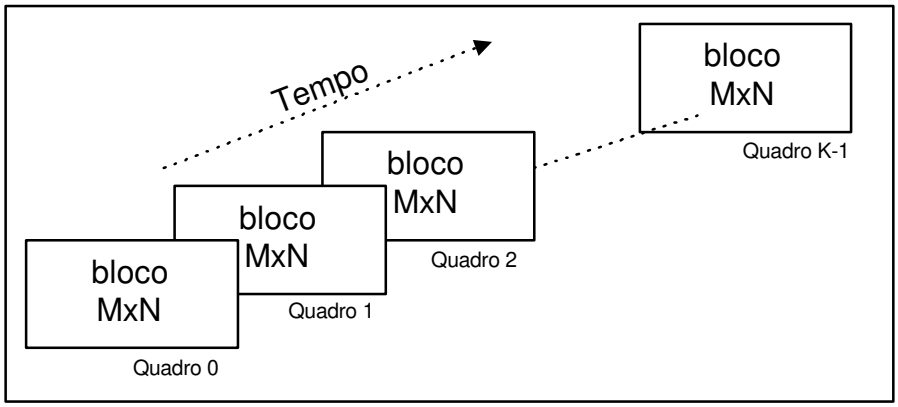


Fig. 4.2 – Combinação DCT/VQ de um arranjo 3D

O desempenho desse tipo de combinação pode ser melhorado pela introdução da filtragem por zona geométrica como mostrado no capítulo anterior, onde alguns coeficientes podem ser adaptativamente descartados. Também, a dimensão ***K*** do vetor pode ser feita adaptativa baseado na correlação temporal entre os quadros **[114]**. Assim, em uma seqüência de imagens com pouco movimento, ***K*** pode ser dimensionado um pouco grande. Já para uma seqüência de imagens com muito movimento, ***K*** deve ser pequeno. Vários *codebooks* de dimensões ***M*** e contendo diferentes vetores de dimensões ***K*** podem ser projetados adaptativamente baseado no movimento, mudança da cena, etc, como ilustrado na **Fig. 4.3**.

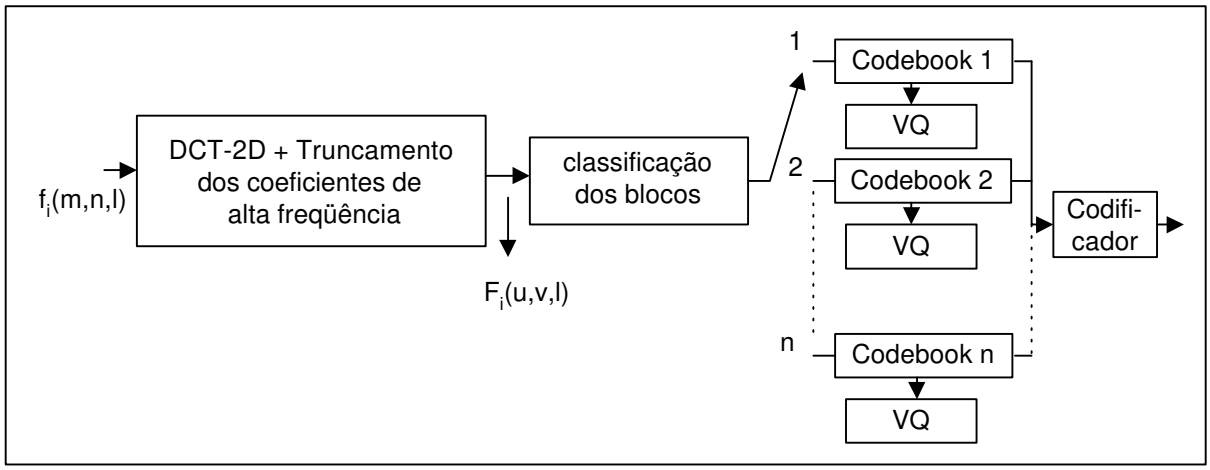


Fig. 4.3 – Sistema adaptativo de quantização vetorial com transformada DCT

4.2.1. Classificação de VQ com DCT (CVQ)

Com o objetivo de preservar as características perceptuais de uma imagem, Ramamurthi e Gersho **[117]** propuseram um esquema de codificação vetorial combinado DCT/VQ, denominado "*Classified VQ (CVQ)*". Nesse esquema, a imagem é dividida em blocos 4x4, ou 5x5, e cada bloco é classificado baseado nas bordas (orientação, localização e polaridade), área de sombra, range médio e outras categorias, como mostrado na **Fig. 4.4**. Para

cada classe, são gerados *codebooks*, denominados *subcodebook* (podendo ter tamanhos diferentes), baseado no vetor de treinamento pertencente àquela classe.

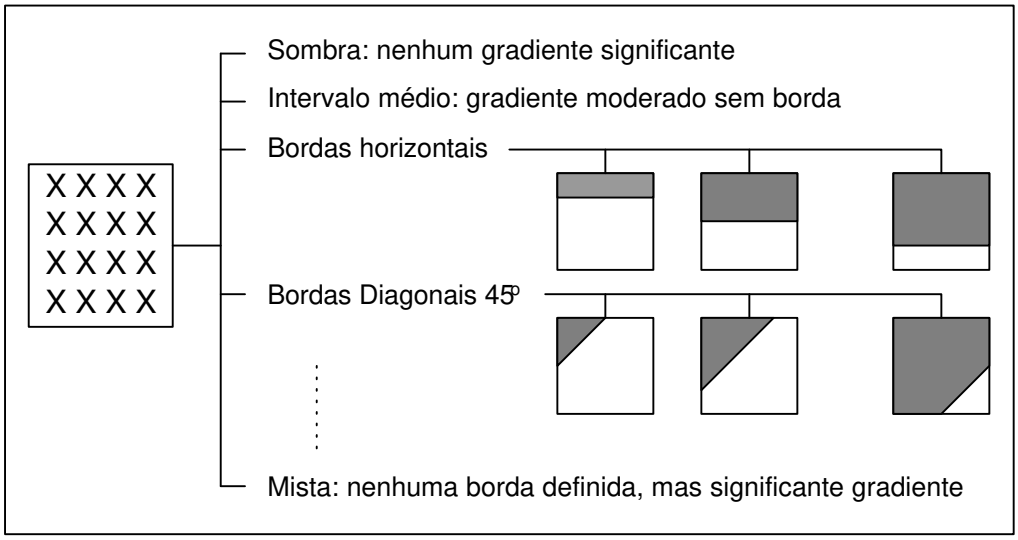


Fig. 4.4 – Classificação dos blocos para CVQ

Nesses sistemas existem ***M*** classes. Inicialmente um classificador seleciona a classe de cada bloco e o processo de VQ é baseado no *subcodebook* de cada classe. Se a entrada **x** pertence a classe ***i***, o i-ésimo *subcodebook* $\boldsymbol{C_i}$ de tamanho $\boldsymbol{N_i}$ será explorado para codificar **x**, usando uma medida de distorção $\boldsymbol{d_i(.\ ,\ .\ )}$. Em geral, a medida de distorção para diferentes classes pode ser diferente. O número total de vetores códigos é dado por: $N_T = \sum_{i=1}^{M} N_i$. O índice do vetor código mais próximo é transmitido para o receptor. O decodificador simplesmente recupera o correspondente vetor código do seu *codebook*, gerando o bloco de saída $\hat{x}$. A **Fig. 4.5** mostra o esquema básico do codificador CVQ.

Uma vez que existe um quantizador vetorial para cada classe, adotou-se estruturas de codebook diferentes para classes diferentes com o objetivo de reduzir a complexidade e melhorar o desempenho do codificador. Para a classe de intervalo médio, foi proposta a transformação de domínio para explorar as propriedades espectrais da classe e obter vantagens como: redução na complexidade de codificação e amoldamento espectral do ruído. Os blocos de intervalo médio de entrada são transformados pela DCT-2D e os coeficientes transformados

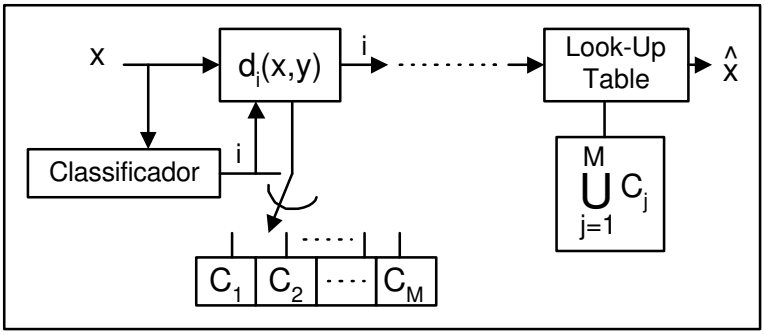


Fig. 4.5 – Codificador CVQ

passam por uma amostragem zonal, onde os coeficientes menos significativos são zerados. O *codebook* transmitido C_t contém vetores códigos no domínio da ***DCT***. Assim, o vetor código em C_t mais próximo a ***S*** (**Fig. 4.6**) é achado usando o ***MSE*** e seu índice ***i*** é transmitido. Os vetores códigos no *codebook* do receptor C_r são o ***DCT*** inverso dos vetores códigos correspondente em C_t. Na **Fig. 4.6** é mostrado o codificador para os blocos de intervalo médio. Para a recuperação dos blocos, é necessária a transmissão de bits de *overhead* que identificam a classe do bloco, além do índice para o modelo selecionado.

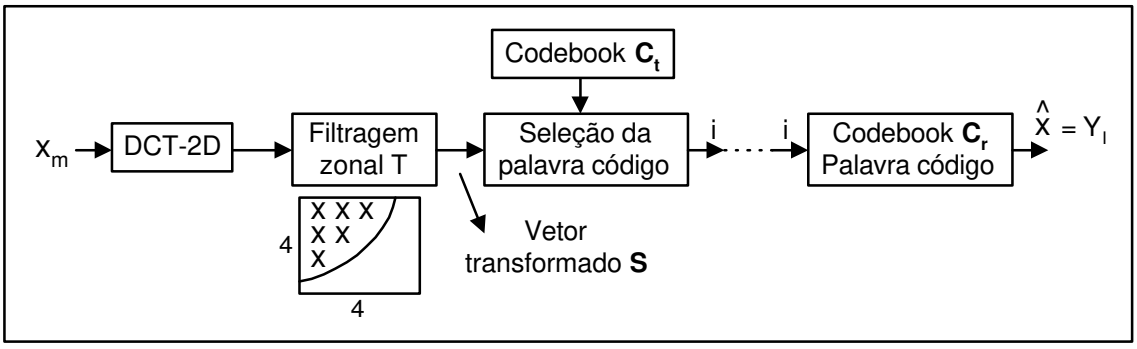


Fig. 4.6 – Codificador de blocos de classe de intervalo médio

Uma variação para o CVQ foi proposto por Labit e Marescq **[118]**, cujo esquema é mostrado na **Fig. 4.7**. A classificação dos blocos é realizada no domínio da DCT e baseada na energia ***ac*** das diferentes regiões (**Fig. 4.8**).

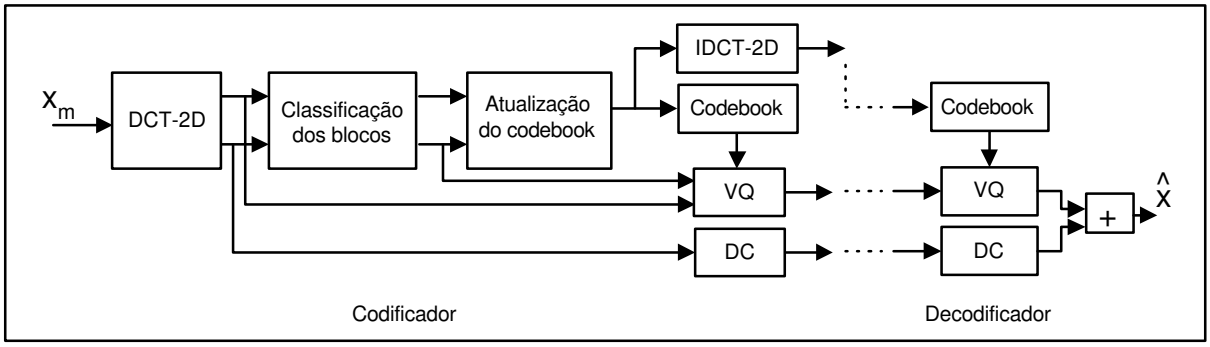


Fig. 4.7 – Codificador CVQ no domínio da DCT aplicado a seqüência de imagens

A adaptatividade é conseguida pela atualização interativa dos *subcodebooks* com os termos ***dc*** de todos os blocos codificados separadamente. Com esse sistema DCT/CVQ adaptativo, segundo Labit, obteve-se melhorias na qualidade visual das imagens processadas para uma específica complexidade de VQ e taxa de bits. Outros esquemas DCT/CVQ foram também propostos **[121]**.

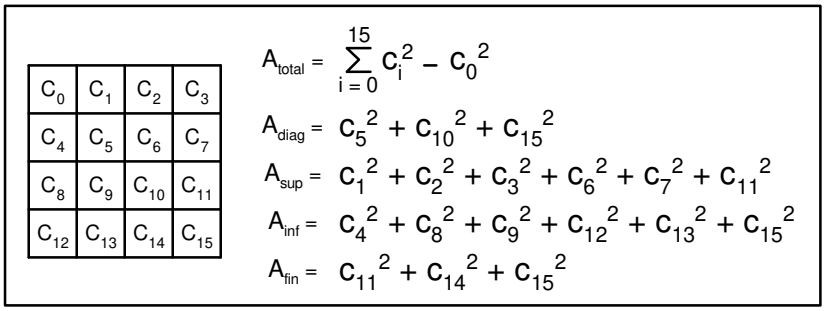


Fig. 4.8 – Atividades parciais de um bloco transformado 4x4

4.2.2. ADCT/VQ de Imagens Monocromáticas e Coloridas

A codificação de imagens com aplicação de quantização vetorial (VQ) apresenta algumas dificuldades na sua implementação. Entre elas, encontra-se a redução da complexidade da quantização vetorial devido a sua tarefa computacional ser relativamente

grande. Outra dificuldade é que normalmente o quantizador vetorial é projetado no domínio espacial original usando o algoritmo ***LBG*** **[113]**. Uma vez que esse procedimento usa o treinamento de um conjunto de dados derivados de imagens específicas, o quantizador vetorial apresenta considerável degradação de qualidade quando a imagem processada não estiver incluída no projeto do *codebook*. Isso torna difícil o uso de quantizador vetorial universal no domínio espacial original. Uma solução para evitar essa desvantagem é quantizar vetorialmente os sinais transformados que idealmente não apresentem nenhuma variação estatística. Se uma função distribuição de probabilidade de um sinal transformado poder ser modelado com uma distribuição padrão bem definida, então um quantizador universal poderá ser projetado desse modelo. Aizawa *et al.* **[122-124]** apresentaram um esquema de quantização vetorial adaptativo no domínio da transformada DCT denominado ***ADCT-VQ***, em que é usado um quantizador vetorial universal independente de imagens especificas. Nesse esquema, o sinal no domínio da DCT é particionado em vetores, os quais são normalizados e quantizados vetorialmente. Uma vez que os coeficientes ***ac*** normalizados são bem modelados como uma distribuição Laplaciana com média zero e variância unitária, eles podem ser eficientemente quantizados usando o quantizador vetorial universal projetado com a distribuição multidimensional Laplaciana com média zero e variância unitária. A **Fig. 4.9** mostra o diagrama em blocos do sistema proposto por Aizawa *et al.*

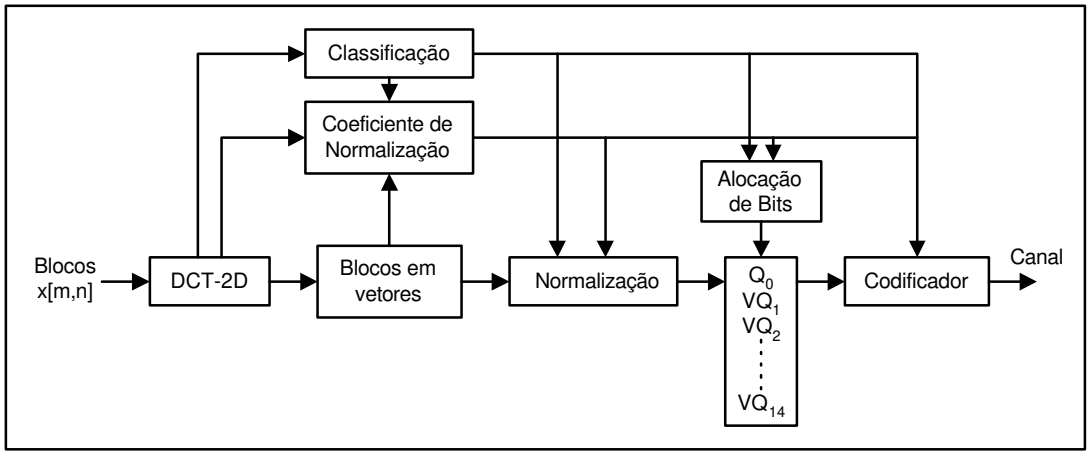


Fig. 4.9 – Codificador adaptativo DCT-VQ

O uso de quantizadores vetoriais universais para os coeficientes da DCT, segundos os autores, resultou em simplificação na complexidade do VQ e melhoria no desempenho da codificação.

Detalhes das operações do algoritmo proposto são mostrados a seguir.

1 – A imagem é dividida em blocos 8x8 e sobre cada bloco é aplicado a DCT-2D;

2 – Os blocos transformados são classificados em 4 (quatro) classes de acordo com seus níveis de atividades (energia ***ac***), tal que cada classe contenha o mesmo número de blocos **[125,126]**. A adaptatividade é obtida nessa operação uma vez que os parâmetros de codificação, como alocação de bits, coeficiente de normalização, etc, são definidos para cada classe;

3 – Cada bloco transformado é particionado no valor ***dc*** (***V₀***) e em 14 vetores (***V₁***, ***V₂***, ..., ***V₁₄***), compostos pelos coeficientes ***ac*** transformados, como mostrado na **Fig. 4.10**;

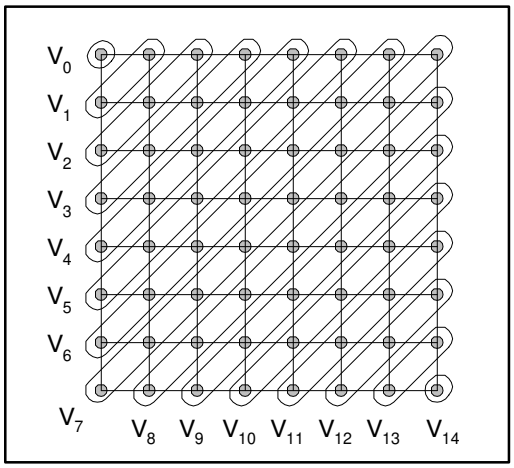


Fig. 4.10 – Particionamento de um bloco DCT em 14 vetores

4 – A variância de cada componente dos 14 vetores em cada classe é computada. Essas variâncias são usadas para determinar a média geométrica dos componentes de cada vetor, o coeficiente de normalização e a atribuição de bits para cada vetor em cada classe;

5 – Cada vetor (***V₁***, ***V₂***, ..., ***V₁₄***) é normalizado pelo coeficiente de normalização $\hat{\sigma}_i$ definido por:

$$\hat{\sigma}_i = \sqrt{\left(\sigma_{i1}^2 . \sigma_{i2}^2 . \sigma_{i3}^2 ..... \sigma_{ik_i}^2\right)^{1/k_i}} \tag{4.7}$$

onde σ_{ij}^2 representa a variância do ***j-ésimo*** componente do vetor $\boldsymbol{V_i}$ que tem dimensão $\boldsymbol{k_i}$.

6 – O termo ***dc*** é quantizado uniformemente em 8 bits;

7 – O número de bits atribuído aos vetores ($\boldsymbol{V_1}$, $\boldsymbol{V_2}$, ..., $\boldsymbol{V_{14}}$), correspondente ao tamanho do *codebook* de cada vetor quantizado, é dado por:

$$b_i = \frac{k_i}{2} log_2 \left(\frac{\frac{C_{k_i}}{k_i} \hat{\sigma}_i^2}{\overline{D}} \right) \tag{4.8}$$

onde

$$C_{k_i} = \frac{2}{\pi} \left(\frac{k_i + 2}{k_i} \right)^{k_i + 1} \Gamma\left(\frac{k_i}{2} + 1 \right)^{\frac{2}{k_i}} \tag{4.9}$$

sendo $\Gamma(.)$ a função Gama e $\overline{D}$ é um parâmetro de medida de distorção de codificação e que determina a taxa de bits. A distorção de codificação em cada vetor pode ser medida pela Equação (4.10) quando os coeficientes ***ac*** transformados são modelados com distribuição Laplaciana.

$$D_i = C_{k_i} \hat{\sigma}_i^2 \, 2^{\frac{-2b_i}{k_i}} \tag{4.10}$$

onde $\boldsymbol{D_i}$ denota a distorção de codificação do vetor $\boldsymbol{V_i}$, $\boldsymbol{b_i}$ representa o número de bits atribuído ao vetor $\boldsymbol{V_i}$. Para minimizar a distorção total no bloco, $\boldsymbol{D_i}$ deve ser constante e igual a $\overline{D}$ para todo $\boldsymbol{i}$;

8 – Os vetores $\boldsymbol{V_i}$ são quantizados vetorialmente pelos vetores $\boldsymbol{VQ_i}$ usando o *codebook* com tamanho correspondente à alocação de bits;

9 – Os índices de saída dos vetores quantizados $\boldsymbol{VQ_i}$ e as informações de *overhead* (alocação de bits, coeficientes de normalização e informação de classificação) são transmitidos;

10 – Se o número de bits atribuído a um vetor exceder a 8 bits, então o vetor correspondente sofre um segundo estágio de quantização vetorial (primeiro estágio: 8 bits; segundo estágio: bits restantes) conforme mostrado na **Fig. 4.11**. No segundo estágio VQ, o vetor erro (diferença entre o vetor de entrada e o vetor selecionado (*codeword*)) sofre uma outra VQ, etc, onde VQ_f é o quantizador vetorial do primeiro estágio e VQ_e o quantizador vetorial do segundo estágio.

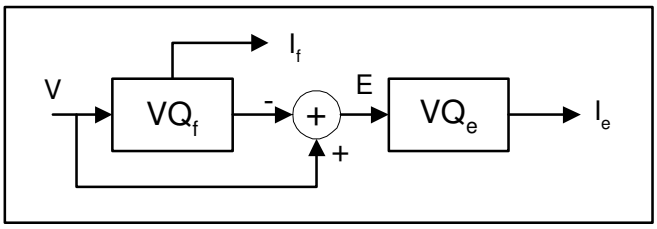


Fig. 4.11 – Quantização vetorial de dois estágios

Esse sistema de codificação foi estendido para imagens coloridas. Porém, as componentes primárias R, G e B são primeiramente convertidas para um outro sistema de sinais componentes denominadas C_0, C_1 e C_2, e sobre cada nova componente é aplicado o esquema de codificação.

4.2.3. DCT/VQ usando categorização com particionamento adaptativo de faixa

Um outro esquema adaptativo, similar ao proposto por Aizawa *et al.* **[124]**, foi desenvolvido por Omachi *et al.* **[127]**. O esquema proposto é mostrado na **Fig. 4.12**. Esse esquema está baseado no conceito da localização dos coeficientes transformados que têm magnitudes que mudam no domínio da DCT de acordo com a direção das bordas no domínio espacial.

Destaques deste algoritmo adaptativo são mostrados a seguir.

1 – A imagem é dividida em blocos 8x8 e sobre cada bloco é aplicada a DCT-2D;

2 – Categorizar os blocos baseado na localização dos coeficientes de grande magnitude. Existem quatro categorias conforme padrões mostrados na **Fig. 4.13**. O bloco que apresenta a maior soma absoluta dos coeficientes dentro da área sombreada é assumido pertencer aquela categoria. A categoria 1 refere-se a blocos que não apresentam bordas, enquanto que as categorias 2, 3 e 4 referem-se às bordas horizontal, vertical e diagonal, respectivamente;

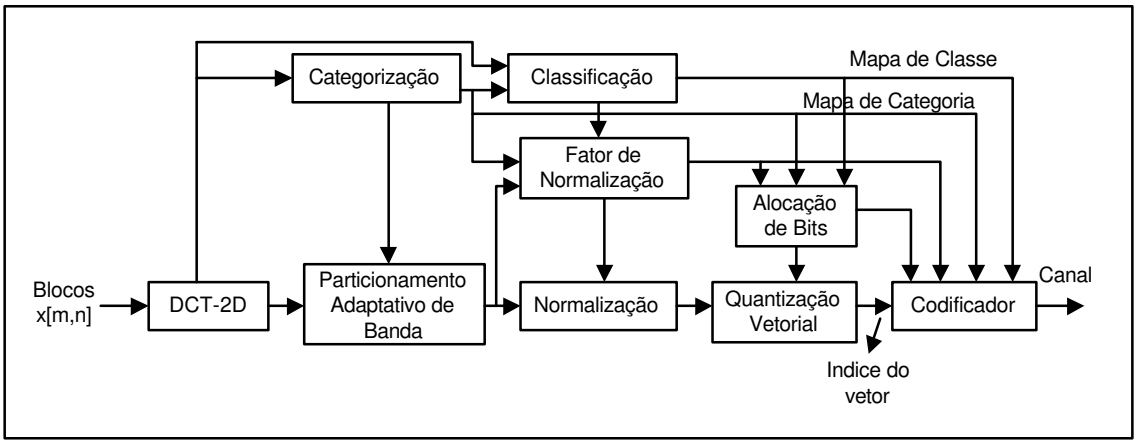


Fig. 4.12 – Esquema de VQ adaptativo por particionamento de faixa

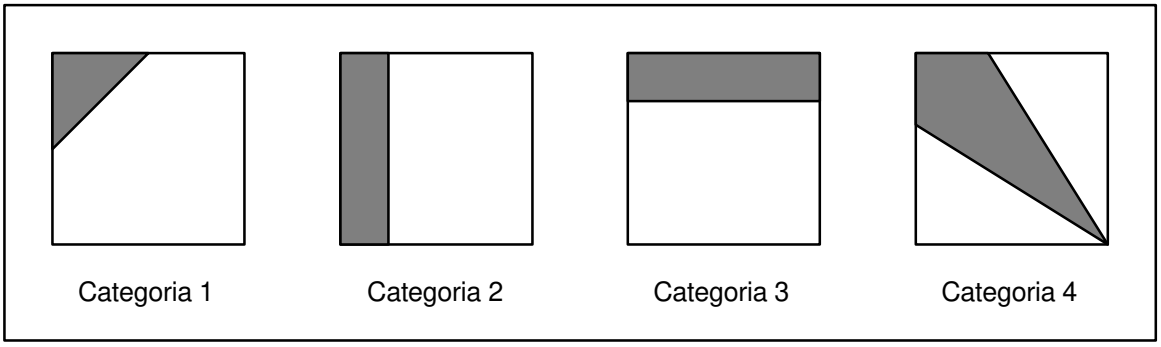


Fig. 4.13 – Padrões de máscara para categorização dos blocos transformados

3 – O particionamento de faixa em cada categoria é mudado adaptivamente, como mostrado na **Fig. 4.14**;

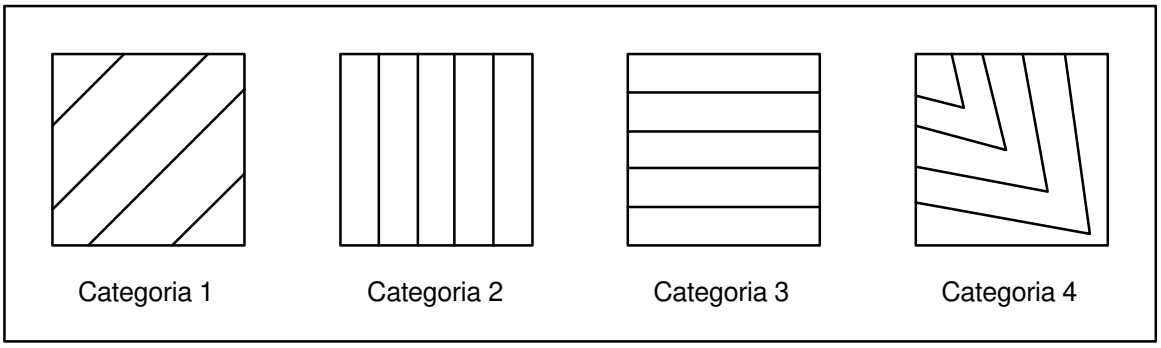


Fig. 4.14 – Particionamento de banda em cada categoria

4 – Os blocos em cada categoria são classificados em quatro grupos baseados na energia ***ac***. A normalização e a alocação de bits é desenvolvida para cada banda em cada classe;

5 – Os índices de saída dos vetores quantizados $\boldsymbol{V_i}$ e as informações de *overhead* (alocação de bits, fator de normalização, categoria e informação de classificação) são transmitidos.

4.3. Codificação de Imagens por Transformada baseada na Estrutura/Distorção

A codificação de imagens por transformada, em geral, é implementada sobre blocos pequenos, tais como 8x8 ou 16x16, não apenas devido a sua complexidade (de processamento e exigência de armazenamento que aumentam com o tamanho do bloco), mas também do ponto de vista da adaptatividade (codificação de blocos individuais pode ser seguido de atividades local ou detalhes). Como a codificação por bloco não assume nenhuma correlação entre blocos vizinhos, artefatos visuais tendem a aparecer nos limites do bloco, quando da codificação em baixas taxas. Várias técnicas têm sido desenvolvidas para minimizar ou eliminar essa estrutura de bloco. Algumas dessas técnicas serão descritas a seguir.

4.3.1. Sobreposição de blocos

Erros na codificação por transformada em blocos independentes produzem descontinuidades entre os blocos, pois as amostras nas fronteiras de um bloco não casam perfeitamente com as amostras das fronteiras dos blocos adjacentes. Esses erros poderão aparecer como efeitos de bloqueamento visível na imagem reconstituída. Com o objetivo de reduzir esse efeito de bloqueamento foi desenvolvida e proposta por Malvar e Staelin **[18, 19]** a transformada LOT, que utiliza o princípio da superposição de blocos para reduzir o efeito de bloqueamento na imagem reconstituída. A LOT combina a DCT de blocos adjacentes de maneira tal que a transformada resultante tem funções bases que sobrepõem blocos adjacentes. Para isso, as funções de base da LOT são maiores que o comprimento da transformada. Nesse caso, as funções bases decaem com transições suavizadas se aproximando de zero nas extremidades. Então, a LOT é virtualmente livre de erros de casamento de blocos. Dessa forma, se o número de coeficientes transformados forem ***M***, as funções base deverão ter comprimento ***L*** maior que ***M***. Assim, as funções base serão projetadas também sobre os blocos vizinhos. Dessa forma, as funções base de um bloco e de seus vizinhos (de ambos os lados) devem ser sobrepostas, como mostrado na **Fig. 4.15** para o caso unidimensional.

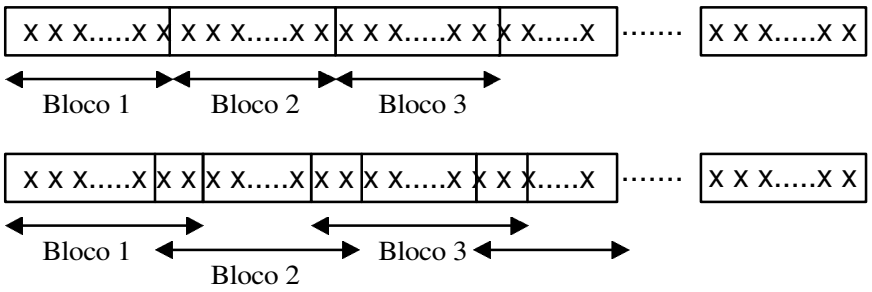


Fig. 4.15 – Sobreposição de blocos unidimensional: *pixels* vizinhos são comuns a blocos adjacentes

Por utilizar o princípio da superposição de blocos, a LOT não pode ser aplicada em uma seqüência discreta ***x[n]*** constituída de um único bloco, sendo necessário que a seqüência seja dividida em ***K*** blocos ($K \geq 2$) para que a LOT seja aplicada.

Para que sejam usadas as mesmas funções bases para a transformada direta e a transformada inversa, a porção da região de sobreposição entre blocos vizinhos deve ser ortogonal.

Para o caso de imagens 2D definida pela seqüência ***x[m,n]*** que tenha tamanho ***CxL***, a mesma deverá ser particionada em blocos ***NxN*** e as funções bases da LOT comporão as colunas de uma matriz **P** de dimensões ***LxN***, onde ***L = 2N***. Para se obter um bloco de coeficientes transformados com tamanho ***N***, a LOT deverá ser aplicada sobre um bloco de tamanho ***L*** das amostras de ***x[m,n]*** (*x[L, N]*), obtendo-se, assim, ***L-N*** amostras sobrepostas para cada lado. Portanto, para a LOT mapear um bloco com comprimento de ***L*** amostras de ***x[m,n]*** em um bloco com comprimento de ***N*** coeficientes transformados (mantendo-se a mesma taxa de amostragem), a seqüência ***x[m,n]*** deverá ser deslocada (para processamento) de ***N*** amostras a cada bloco a ser transformado. Dessa forma, as amostras de um bloco serão usadas em dois blocos consecutivos, configurando-se assim a sobreposição dos blocos. A **Fig. 4.16** mostra uma idéia como base para esse processo.

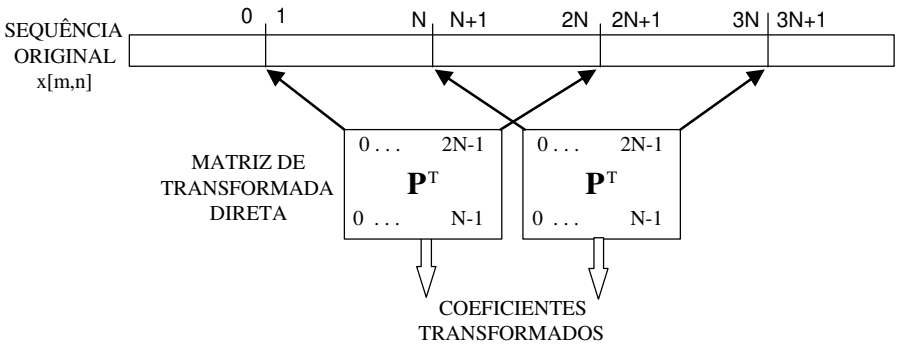


Fig. 4.16 – Idéia base do processo de transformação de um bloco de tamanho *L*

A transformada LOT inversa reconstitui um bloco de tamanho ***L*** a partir de um bloco de tamanho ***M*** de coeficientes transformados, com ***L - M*** amostras sobrepostas para cada lado.

Como pode ser visto na **Fig. 4.17**, a seqüência reconstituída é formada pela sobreposição de blocos adjacentes. A perfeição na reconstituição com a transformada LOT só é conseguida se a seqüência original de entrada for de comprimento infinito.

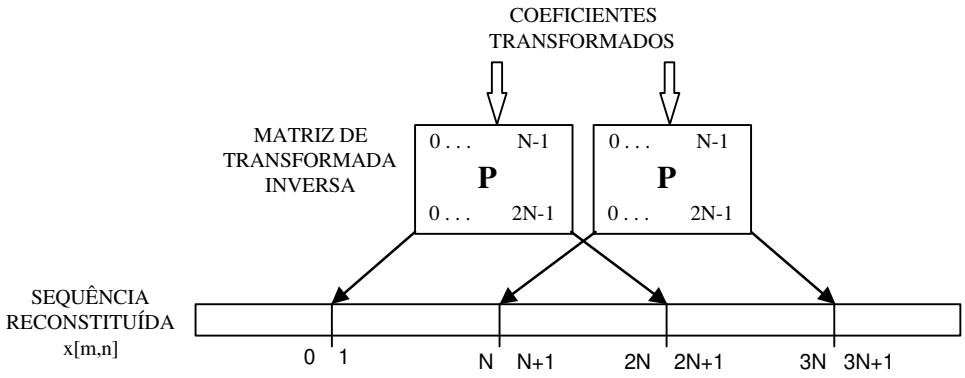


Fig. 4.17 – Reconstituição da seqüência x[m,n] original por transformada inversa

Para o caso de processamento de imagens (seqüências bidimensionais e de comprimento finito), a LOT é aplicada sobre as linhas e as colunas da mesma. Porém, devido ao procedimento de sobreposição dos blocos, deve-se observar a necessidade de uma modificação para a obtenção da LOT dos primeiros e dos últimos blocos de cada linha ou coluna. Assim as funções bases para esses blocos deverão ser menores.

4.3.2. Classificação de Atividades em Codificação Adaptativa por Transformada

Em codificação adaptativa de imagens, a atividade de um bloco pode ser descrita de várias maneiras. O conceito geral é alocar mais bits para a classe de mais alta atividade e vice-versa. Assumindo que uma imagem de tamanho ***C x L*** é dividida em blocos de tamanho ***NxN***, os coeficientes DCT-2D de cada bloco são dados por ***X[u,v]***, onde **u,v = 0, 1,..., N-1**. Os blocos da imagem podem ser divididos em várias classes para a codificação adaptativa. Algumas dessas classes são mostradas a seguir.

4.3.2.1. Energia AC

Considerando que o coeficiente DC representa o brilho médio de um bloco, a energia AC é uma indicação da atividade ou detalhe do bloco. Sua descrição é dada por:

$$E_{AC} = Energia\ AC\ de\ um\ Bloco = \sum_{u=0}^{N-1}\sum_{v=0}^{N-1}\left[X(u,v)\right]^2 - \left[X(0,0)\right]^2 \qquad (4.11)$$

Chen-Smith **[126]** apresentaram um sistema eficiente de codificação adaptativa de imagens por Transformada DCT-2D que explora essas características da imagem. Esse sistema apresenta uma otimização na alocação de bits para os coeficientes transformados quando comparado com o sistemas anteriores. O algoritmo básico apresentado por Chen-Smith é o seguinte:

1 – Uma imagem de tamanho ***MxM*** é dividida em blocos ***16x16*** e sobre cada bloco é aplicada a DCT-2D;

2 – A energia AC de todos os blocos transformados é calculada segundo a equação:

$$E_{AC}(m,l) = \sum_{u=0}^{15}\sum_{v=0}^{15}\left[X_{m,l}(u,v)\right]^2 - \left[X_{m,l}(0,0)\right]^2 \qquad (4.12)$$

sendo ***(m,l)*** a posição do bloco atual dentro da imagem onde, ***m*** = *1,2,...,M/16* e ***l*** = *1,2,...,M/16*;

3 – Os blocos transformados são classificados em ordem crescente de energia AC e o número total de blocos é dividido em ***K*** classes, de modo que cada classe contenha o mesmo número de blocos. A informação da classe a que pertence cada bloco é guardada para uso posterior (*K= 4)*;

4 – É calculada a média do conjunto das variâncias $\sigma_k^2(u,v)$ de cada coeficiente *AC* transformado para cada classe, onde *u,v = 1, 2,...,15* e *k = 0, 1,...,K*, segundo a equação:

$$\sigma_k^2(u,v) = \frac{256\,K}{M^2} \sum_{m=1}^{\frac{M}{16\sqrt{K}}} \sum_{l=1}^{\frac{M}{16\sqrt{K}}} \left[X_{m,l}(u,v)\right]^2 \tag{4.13}$$

5 – As variâncias $\sigma_k^2(u,v)$ são utilizadas para gerar matrizes $NB_k(u,v)$ de alocação de bits para cada classe;

A adaptatividade desse método consiste em alocar maior número de bits não só para as classes de maior atividade, como também para os coeficientes de mais baixa freqüência em cada classe.

Esta solução é teoricamente ótima, uma vez que os coeficientes de uma mesma posição no bloco transformado, mas pertencentes a regiões com características diferentes da imagem, não precisarão ser quantizados com o mesmo número de níveis. Coeficientes pertencentes a regiões de maior energia *AC***,** certamente, precisarão de maior número de níveis de reconstituição, enquanto que aqueles pertencentes a regiões mais suaves precisarão de menor número de níveis. O sistema apresentado gera uma informação de *overhead*, visto que é necessário a transmissão das informações das classes, matrizes de alocação de bits, fatores de normalização, etc, para a decodificação da imagem. Entretanto, apesar dessa sobrecarga de informação, o sistema apresenta desempenho superior ao caso da codificação sem a classificação dos coeficientes.

4.3.2.2. Soma das Magnitudes dos Coeficientes AC

Gimlett **[128]** originalmente propôs um parâmetro que definisse a atividade do bloco de coeficientes transformados. Esse parâmetro pode ser expresso por:

$$A = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} \left|X(u,v)\right| - \left|X(0,0)\right| \tag{4.14}$$

Grandes valores de ***A*** indicam blocos com forte presença de bordas, enquanto que a magnitude do coeficiente DC pode indicar blocos escuros com pouca energia AC. O parâmetro ***A*** juntamente com o valor DC ($|X(0,0)|$) foram utilizados por Gilge **[129,130]** para classificação de blocos em codificação de imagens para impressão em quatro cores (amarelo, magenta, ciano e preto). Essa classificação serviu para determinar o número de bits alocados, o limiar (acima do qual os coeficientes transformados são codificados) e também o número de coeficientes colocados iguais a zero em cada bloco. Gimlett sugeriu também medidas de atividades ponderadas tais como:

- Ponderação da Energia AC de um Bloco

$$PAC = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} w_1(u,v)\left[X(u,v)\right]^2 \; - \; \left[X(0,0)\right]^2 \tag{4.15}$$

- Soma das Magnitudes dos coeficientes AC Ponderados

$$SMP = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} w_2(u,v)\left|X(u,v)\right| \; - \; \left|X(0,0)\right| \tag{4.16}$$

onde $w_1(u,v)$ e $w_2(u,v)$ são os pesos para aplicação em codificação de imagens.

4.3.2.3. MACE e Direção

Para reduzir as diferenças entre blocos que pertencem a mesma classe, Wu e Burge **[131]** propuseram uma classificação de blocos baseada em suas características definidas como:

- Energia AC das Freqüência Medianas – (MACE)

$$MACE(p,q) = \sum_{u=0}^{p-1} \sum_{v=0}^{p-1} \left[X(u,v)\right]^2 \; - \; \sum_{u=0}^{q-1} \sum_{v=0}^{q-1} \left[X(u,v)\right]^2 \tag{4.17}$$

MACE representa a energia AC dentro de uma janela espacial no domínio da freqüência com *MACE(N-1,0)* como a energia AC do bloco.

- Características Direcionais (DIR)

$$DIR = \frac{\sum_{u=0}^{15}\sum_{v=0}^{15} tan^{-1}\left(\frac{u}{v}\right)\left|X(u,v)\right|}{\sum_{u=0}^{15}\sum_{v=0}^{15}\left|X(u,v)\right|} \tag{4.18}$$

Esse parâmetro tem exibido uma forte característica direcional.

4.3.2.4. Distribuição de Potência dos Coeficientes

Mukawa e Okubo **[132]** propuseram que os blocos transformados poderiam ser classificados baseados nas posições dos coeficientes com potência significante (coeficientes com potência grande o suficiente para ser transmitido). A energia *AC* definida por **[126]** em (4.12), não considera as características posicionais dos coeficientes significativos. Em **[132]**, os blocos são classificados de acordo com a distribuição de potência dos coeficientes. Cada coeficiente significante é uniformemente quantizado e codificado com um código de comprimento variável (VWLC). A adaptatividade é conseguida com a troca do código (VWLC) apropriado para a Função Densidade de Probabilidade de cada coeficiente quantizado.

4.3.2.5. Segmentação Adaptativa dos Blocos DCT-2D em Regiões

Um outro esquema de classificação de atividades do bloco transformado é o proposto por Franceschi *et al.* **[133]**. Esse método está baseado na segmentação do bloco transformado

por DCT-2D em regiões tal que a estatística de cada região seja quase estacionária. As regiões são segmentadas de maneira tal que as magnitudes dos coeficientes transformados em cada região estejam dentro de um limiar escolhido para aquela região. Tanto o número de regiões quanto o limiar podem ser variáveis. Um exemplo dessa segmentação é mostrado na **Fig. 4.18**.

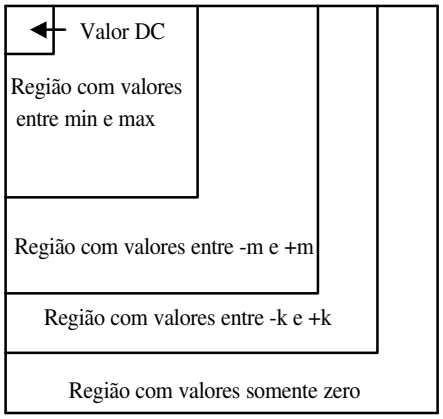


Fig. 4.18 – Segmentação adaptativa em blocos DCT-2D em regiões

Vários outros métodos de classificação de atividades dos blocos transformados foram propostos na literatura, como podem ser vistos em **[37,104,134-136]**. Os métodos abordados aqui serviram como base para o presente trabalho que será apresentado no capítulo seguinte.

CAPÍTULO 5

Codificação Adaptativa de Imagens Usando DCT

5.1. Introdução

A crescente exigência para transmissão e armazenamento de imagens digitalizadas com alta resolução provocou o surgimento de propostas de várias técnicas de compressão de faixa usando Quantização Vetorial (VQ) e/ou codificação por transformada.

Como visto em capítulos anteriores, a Quantização Vetorial (VQ) despertou considerável interesse, uma vez que a mesma apresentou desempenho quase ótimo de *taxa x distorção* **[117,124,137-139]**. Porém, estudos revelaram algumas dificuldades na sua aplicação tais como: 1- um quantizador vetorial é normalmente projetado no domínio espacial original pelo algoritmo de agrupamento denominado LBG **[140]**. Esse algoritmo usa treinamento de conjunto de dados derivados de imagens específicas para o projeto do *codebook*. Portanto, para imagens que não fizeram parte do projeto do *codebook*, o quantizador apresenta degradações consideráveis nas bordas dos blocos. Considerando que as bordas constituem uma porção considerável de informação de uma imagem, a degradação das mesmas apresentam um efeito muito incomodo na imagem recuperada **[141]**; 2 – a complexidade da VQ cresce exponencialmente com a taxa e com o tamanho dos blocos a serem processados, resultando em uma pesada tarefa computacional, o que restringe o uso dessa técnica a blocos relativamente pequenos **[117]**. Dessa forma, é difícil usar um quantizador vetorial universal no domínio espacial original.

Em codificação de imagens por transformada, uma imagem de tamanho *MxM* é processada em blocos individuais de tamanho *NxN*. Para codificação a baixas taxas, é comum (natural) o aparecimento do efeito de bloco **[142]** resultando na degradação da qualidade subjetiva e objetiva da imagem recuperada. Para a redução do efeito de bloco, vários estudos têm sido realizados e vários métodos têm sido propostos com esse objetivo **[20,143]**.

A implementação de um sistema prático tem como base a escolha de um esquema de codificação eficiente que combine o compromisso efetivo de complexidade e desempenho, independente da imagem a ser processada.

Considerando as dificuldades acima mencionadas, é proposto neste trabalho uma nova abordagem de codificação adaptativa eficiente usando a Transformada Rápida Cosseno Discreta (**FDCT**) **[144]**, com blocos de tamanho 8x8 para a redução de taxa de bits na compressão de imagens, e mantendo a qualidade subjetiva da imagem recuperada. O método de processamento está baseado em um modelo de fonte composto que visa reduzir não apenas o efeito de bloco, mas também as possíveis degradações de bordas inerentes do processo. O esquema proposto busca sempre um compromisso ótimo entre *taxa de bits* x *qualidade da imagem recuperada*, através da análise de comportamento dos blocos transformados e da estatística dos coeficientes, independente da imagem a ser processada. No processamento, os blocos transformados são analisados e classificados de acordo com seu nível de atividade e suas características perceptuais. Para a análise da atividade é usado como parâmetro o nível de energia *ac* e para características perceptuais são usadas possíveis predominâncias de bordas dentro do bloco. Os blocos transformados são ordenados de acordo com seu nível de atividade (energia ***ac***) e divididos em classes de energia **[125,126]**. Cada classe é analisada quanto as distintas características perceptuais que, por sua vez, são divididas em subclasses. Cada classe é analisada por um classificador para determinar as subclasses de cada bloco, com base na localização dos coeficientes de maior magnitude e de acordo com a predominância de orientação de borda presente no mesmo **[117]**. Os blocos que apresentam a maior energia ***ac*** dentro de uma determinada classe são escolhidos pertencer àquela classe **[127]**. Analisa-se primeiramente três orientações de borda: horizontal, vertical e diagonal. Posteriormente, é feita a análise com uma quarta borda: triangular superior. A adaptatividade é conseguida com a distribuição de bits entre as classes, favorecendo aquelas com maior nível de atividade. Os bits de cada classe são então divididos entre suas subclasses correspondentes. Os elementos transformados de cada subclasse são quantizados não uniformemente e codificados como prescrito pela matriz de distribuição de bits de cada subclasse. Cada elemento individual de cada subclasse é contemplado com bits de acordo com a variância daquele elemento naquela subclasse. Para limitar o âmbito da investigação, restringe-se ao estudo de imagens paradas.

Porém, o método pode ser estendido para seqüências de imagens, auxiliando no processo de codificação híbrida. É demonstrado que, com o método proposto, a qualidade visual da imagem recuperada é comparável com aquela produzida pelos codificadores existentes para a taxa de 1 bit/pixel, porém com uma melhora na relação sinal/ruído para as mesmas taxas como será demonstrado.

5.2. Codificação Adaptativa de Imagens usando DCT

Para a codificação adaptativa de imagens usando DCT, primeiramente, a imagem de entrada de dimensões (***C x L***), a ser processada pelo sistema, é dividida em blocos de tamanho 8x8 *pixels*, nos quais é executada uma transformada FDCT bidimensional. Sobre os blocos transformados é realizada toda a operação de processamento adaptativo. A descrição do sistema de processamento adaptativo proposto é realizada a seguir.

5.2.1. Transformada Rápida Cosseno Discreta

A transformada cosseno discreta de Fourier unidimensional de uma função ***x(t)*** é dada por **[156]**:

$$X(k) = \frac{2}{N} e(k) \sum_{n=0}^{N-1} x(n) \cos\left[\frac{(2n+1)\pi k}{2N}\right] \tag{5.1}$$

onde

$$e(k) = \begin{cases} \frac{1}{\sqrt{2}} & ; \quad k = 0 \\ 1 & ; \quad k \neq 0 \end{cases} \qquad k = 0,1,2,\cdots,N-1 \tag{5.2}$$

e sua transformada inversa dada por:

$$x(k) = \sum_{k=0}^{N-1} e(k)X(k)\cos\left[\frac{(2n+1)\pi k}{2N}\right] \tag{5.3}$$

Um número razoável de algoritmos rápidos para se calcular a DCT foi descrito na literatura **[145–150]**. Entre os algoritmos mais eficientes está o desenvolvido por Chan and Ho **[144]** (e utilizado no sistema proposto), que segue a lógica similar ao algoritmo da FFT, exceto por uma reordenação preliminar dos dados de entrada. Assim, considerando a seqüência {***x(n)***} de ***N*** pontos, ***N*** par e múltiplo de ***2***, essa seqüência {***x(n)***} pode então ser reordenada para gerar uma nova seqüência ***y(m)*** na seguinte forma:

$$\begin{aligned} y(n) &= x(2n) \\ y(N-1-n) &= x(2n+1) \end{aligned} \qquad n = 0,1,2,\cdots,\frac{N}{2}-1 \tag{5.4}$$

O efeito dessa reordenação pode ser observado considerando-se as seqüências seguintes:

1 – Considerando uma seqüência ímpar:

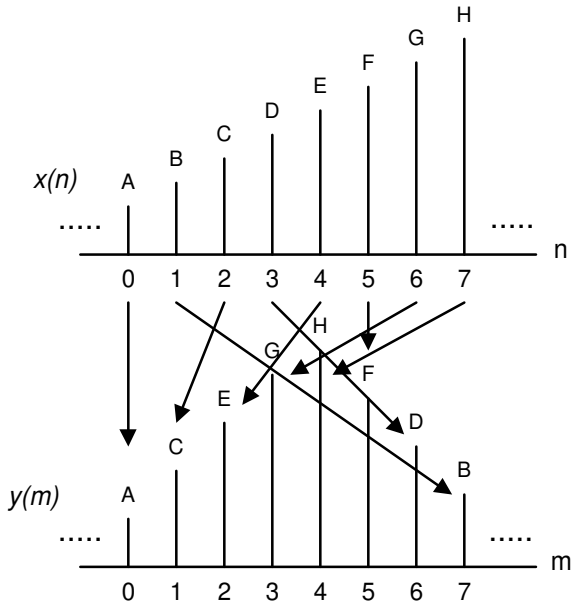


Fig. 5.1 – Visualização da Equação (5.4) para seqüência ímpar

2 – Considerando uma seqüência par:

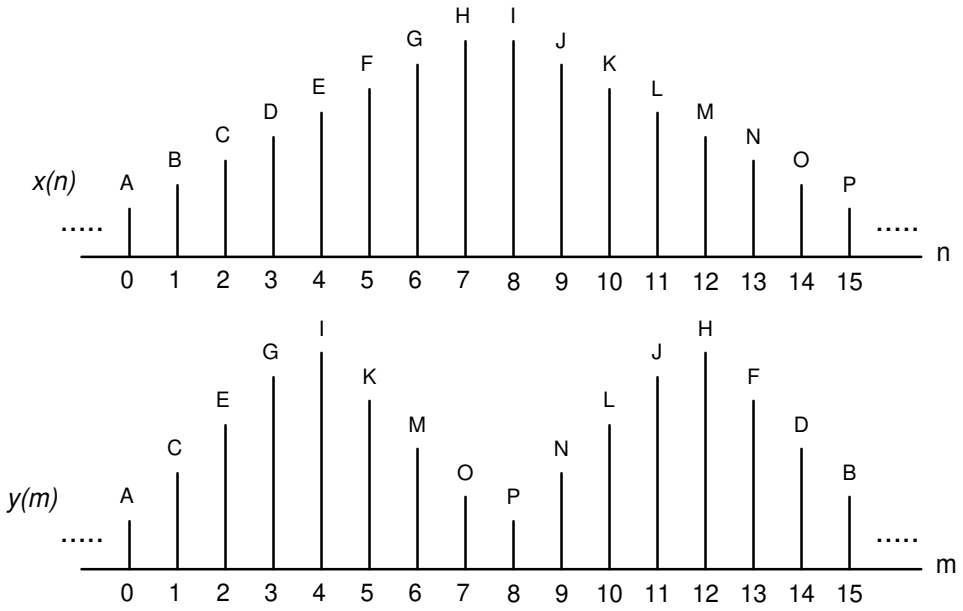


Fig. 5.2 – Visualização da Equação (5.4) para seqüência par

Então, aplicando-se a DCT na seqüência {***y(m)***}, seguido de algumas manipulações matemáticas, resulta em (exceto pelo fator de escala):

$$X(k) = \sum_{n=0}^{N-1} y(n) \cos\left[\frac{(4n+1)\pi \quad k}{2N}\right] \tag{5.5}$$

Fazendo a decimação em freqüência na Equação (5.5) e expressando-a em elementos pares e ímpares tem-se, respectivamente:

$$X(2k) = \sum_{n=0}^{N-1} y(n) \cos\left[\frac{(4n+1)\pi \quad 2k}{2N}\right] \tag{5.6}$$

$$X(2k+1) = \sum_{n=0}^{N-1} y(n) \cos\left[\frac{(4n+1)\pi \quad (2k+1)}{2N}\right] \tag{5.7}$$

Após algumas manipulações nas Equações (5.6) e (5.7), obtém-se o gráfico de fluxo do algoritmo da FDCT unidimensional de 8 pontos mostrado na **Fig. 5.3**.

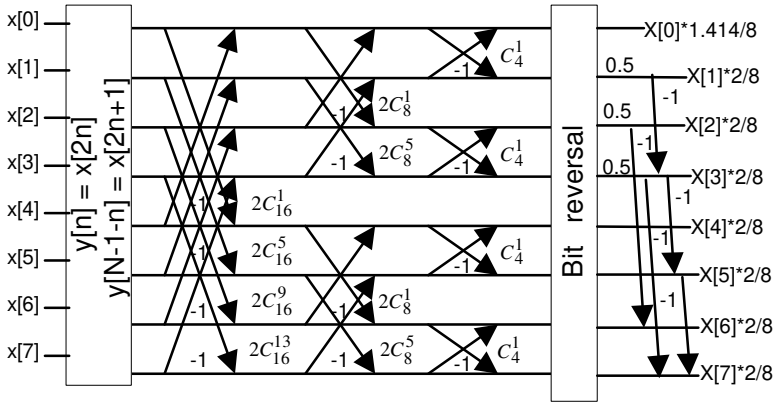


Fig. 5.3 – Gráfico de fluxo da FDCT.

O gráfico de fluxo do algoritmo da IFDCT unidimensional de 8 pontos é mostrado na **Fig. 5.4**.

Os algoritmos rápidos para a FDCT-2D e IFDCT-2D podem ser realizados usando a aproximação linha-coluna, similar aos algoritmos utilizados na FFT-2D e IFFT-2D, respectivamente. Detalhes desses algoritmos são mostrados no **Apêndice B**.

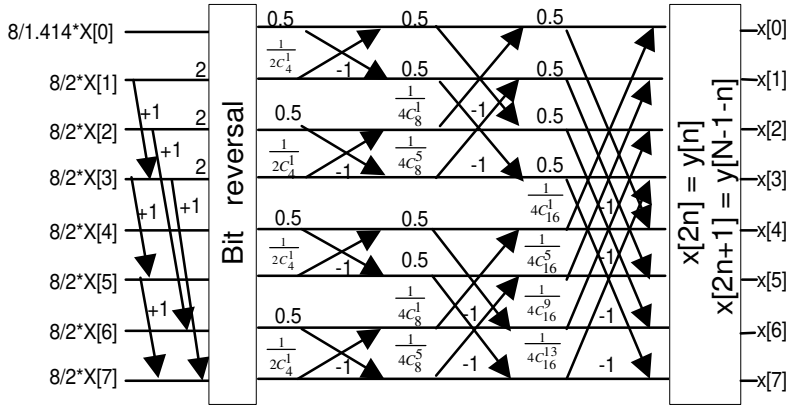


Fig. 5.4 – Gráfico de fluxo da IFDCT.

5.2.2. Classificação de atividade dos blocos Transformados

Os blocos transformados pelo método anteriormente exposto são analisados e classificados de acordo com o nível de atividade da energia ***ac*** conforme a equação:

$$e_{AC}(k,l) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} [X_{k,l}(i,j)]^2 - [X_{k,l}(0,0)]^2 \qquad k,l = 0,1....K,L \qquad (5.8)$$

onde ***K*** e ***L*** são múltiplos de ***i*** e ***j***, respectivamente, ***K = Lin/N*** (*Lin* = número de linhas da imagem), ***L = Col/N*** e ***N = 8***, (*Col* = número de colunas da imagem). A energia ***ac*** de cada bloco é classificada em ordem crescente de atividade, quando, então, é extraído um histograma ***histo[e_{AC}(k,l)]*** para a determinação da distribuição de probabilidade cumulativa da energia ***ac*** dos blocos. Essa distribuição é dividida em quatro partições não uniformes, conforme os níveis de energia ***ac*****,** gerando quatro classes de energia separadas pelos limites *0,25*, *0,50* e *0,75* **[126],** tal que cada bloco pertence a uma classe determinada como mostra a **Fig. 5.5**.

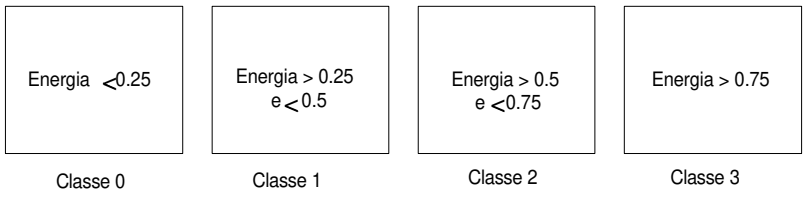


Fig. 5.5 - Classificação dos blocos em níveis de energia

A classe de cada bloco é definida por:

$$classe(i,j) = m \qquad m = 0,1,2 \text{ ou } 3$$
$$(i,j) = (0,1,...,\frac{K}{N}\ ,\ 0,1,...,\frac{L}{N}) \qquad (5.9)$$

onde ***classe(i,j)*** é uma matriz de classificação de energia e ***m*** é o número da classe a que o bloco pertence. A enumeração das classes é realizada no modo crescente de energia, onde blocos com maior energia pertencem às classes mais altas. Na determinação de cada classe não é levada em consideração a localização dos coeficientes de maior energia dentro dos blocos.

Para cada classe ***m***, obtém-se a energia ***ac*** total daquela classe, calculada por (5.10), onde, e_{ACmj} é a energia ***ac*** do bloco ***j*** na classe ***m***, M_m é o número de blocos na classe ***m*** e E_m é a energia ***ac*** total da classe ***m***. É calculado também o número de blocos que a compõem.

$$E_m = \sum_{j=1}^{M_m} e_{AC\,m\,j} \tag{5.10}$$

Em seguida determina-se o número de bits necessários para cada classe ***m***, baseado na energia ***ac*** total daquela classe, dado pelo modelo (5.11) **[151]**.

$$bit_{cl}(m) = [E_m]^{1/3} \tag{5.11}$$

Cada classe ***m*** de energia é primeiramente dividida em três subclasses de acordo com a predominância de orientação de borda (subclasses 1, 2 e 3). Posteriormente, é acrescida mais uma subclasse (subclasse 4). As orientações das bordas estudadas são na direção horizontal, vertical, diagonal e borda triangular superior como mostradas na **Fig. 5.6**.

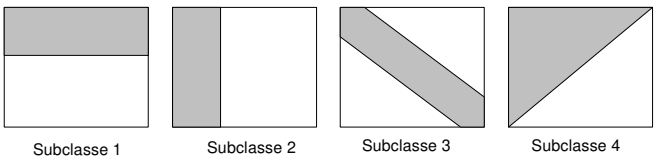


Fig. 5.6 – Direcionamento de bordas

É gerada uma matriz de classificação de subclasse $de_{ac}(i, j)$ dentro de cada classe de acordo com a predominância de orientação de borda por energia ***ac***, dada por:

$$E_h = \sum_{i=0}^{3} \sum_{j=0}^{N-1} [X(i,j)]^2 \tag{5.12}$$

$$E_v = \sum_{i=0}^{N-1} \sum_{j=0}^{3} [X(i,j)]^2 \tag{5.13}$$

$$E_d = \sum_{i,j=3 col.diag} [X(i,j)]^2 \tag{5.14}$$

$$E_{Ts} = \sum_{i,j=triang.sup.} [X(i,j)]^2 \tag{5.15}$$

onde $\boldsymbol{de_{ac}(i,j) = 1, 2, 3}$ ou $\boldsymbol{4}$, conforme o bloco da subclasse seja horizontal ($\boldsymbol{E_h}$), vertical ($\boldsymbol{E_v}$), diagonal ($\boldsymbol{E_d}$) ou triangular superior ($\boldsymbol{E_{Ts}}$), respectivamente.

5.2.3. Alocação de bits para os blocos Transformados

Após obtida a classificação de subclasse, é calculado o número total de blocos por subclasse e a variância média de cada elemento em cada subclasse conforme a Equação (5.16):

$$Var_m(p,q) = \frac{1}{nbloc(p)} \sum_{i=0}^{L-1} \sum_{j=0}^{C-1} [X_{i,j}(p,q)]^2 \tag{5.16}$$

onde $p = 0,1,...,11$ ou $p = 0,1,...,15$ número da subclasse, $q = 0,1,...,Q-1$ número do elemento no bloco, $Q = NxN$, $L = Lin/N$, $C = Col/N$ e $nbloc(p)$ = número de blocos em cada subclasse.

A matriz de alocação de bits para cada subclasse é calculada baseada na técnica de controle da taxa de bits conforme as propriedades estatísticas da imagem e dada por (5.17):

$$bit(p,q) = \left\lfloor \frac{1}{Q-1} \left[R + \frac{1}{2} \log_2 \left(\frac{Var_m(p,q)}{K} \right) \right] \right\rfloor \tag{5.17}$$

onde

$$K = \prod_{p,q} \left[Var_m(p,q) \right]^{1/(Q-1)} ,$$

$\lfloor . \rfloor$ é o maior inteiro do argumento, R é um parâmetro determinado por $bit_{cl}(m)$ (Equação (5.11)), sendo que $0 \leq \beta\iota\tau(\pi,\theta) \leq 8$. O denominador K do segundo membro da Equação (5.17) é a média geométrica das variâncias médias de cada elemento ***ac*** de cada subclasse **[152,153]**.

5.2.4. Normalização e Quantização dos coeficientes Transformados

Após definidos a matriz de alocação de bits ***bit(p,q)*** e a variância média de cada elemento em cada subclasse $Var_m(p,q)$, define-se para cada subclasse um fator de normalização fn_p e uma matriz de posicionamento de elementos diferentes de zero. O fator de normalização para cada subclasse é baseado na maior $Var_m(p,q)$ que tem ***bit(p,q) = 1*** como definido pela Equação (5.18).

$$\begin{aligned} &fn_p = \left(Var_m(p,q)\right)^{1/2} \quad se \quad bit(p,q) = 1 \\ &p = 0\ a\ 11\ ou\ 0\ a\ 15 \quad e \quad q = 1\ a\ 63 \end{aligned} \tag{5.18}$$

As amostras da imagem a ser processada pelo sistema e transformadas por FDCT são normalizadas pelos fatores de normalização fn_p, dependente da subclasse a que o bloco pertence, e orientadas pela matriz de classificação de subclasse e a matriz de posicionamento de elementos diferentes de zero. A normalização do coeficiente ***dc*** de cada bloco é dada pela Equação (5.19) e a dos coeficientes ***ac*** é dada pela Equação (5.20).

$$X_p(0,0) = \frac{X(0,0)}{2 * fn_p} \tag{5.19}$$

$$X_p(i,j) = \frac{X(i,j)}{fn_p . 2^{bit(i,j)-1}} \tag{5.20}$$

Em seguida, as amostras normalizadas são otimamente quantizadas, com o número de níveis de quantização usando o método de quantização de Lloyd-Max **[154,155]** e o número de bits de acordo com a matriz de alocação de bits ***bit(p,q)*** (para a simulação do sistema, usa-se os modelos da função de densidade de probabilidade Gaussiana e Laplaciana para a determinação dos níveis de decisão e de reconstrução). As amostras otimamente quantizadas são então codificadas e incorporadas com os dados de informação de *overhead* (tamanho da imagem, matriz de classificação das subclasses, matrizes de alocação de bits e coeficientes de normalização) sendo então, transmitidas por um canal ou armazenadas em bancos de dados.

A informação total de *overhead* pode ser escrita como:

$$b_T = b_{TM} + b_{MC} + b_{fn} + b_{MdB} \ , \tag{5.21}$$

onde b_{TM}, b_{MC}, b_{fn} e b_{MdB} representam a média de bits de codificação de *overhead* requeridas para codificar: o tamanho da imagem, a matriz de classificação de subclasses, os fatores de normalização e as matrizes de alocação de bits respectivamente. Esses fatores podem ser representados pelas equações:

$$b_{TM} = \frac{32}{LC} \ bits \ ; \tag{5.22}$$

$$b_{MC} = \frac{n\left(\frac{LC}{N^2}\right)}{LC} \ bits \ ; \tag{5.23}$$

$$b_{fn} = \frac{Pbn}{LC} \ bits \ ; \tag{5.24}$$

$$b_{MdB} = \frac{PN^2 \ fc}{LC} \ bits \ . \tag{5.25}$$

Sendo,

$$fc = \frac{\sum_{p=0}^{P-1}\sum_{q=0}^{Q-1} bit(p,q)}{8N^2} \quad para \quad bit(p,q) \neq 0 \ , \tag{5.26}$$

onde ***L*** e ***C*** representam o tamanho da imagem, ***n*** é o número de bits necessários para representar cada elemento na matriz de classificação de blocos, ***P*** é o número de matrizes de alocação de bits (***P = 12*** ou ***16***), ***bn*** é o número de bits usados para representar os fatores de normalização, ***fc*** é um fator de compressão, ***N*** é o tamanho do bloco usado na transformação e $\boldsymbol{Q = N^2}$ é o número de elementos em um bloco transformado. O número 32 corresponde a 4 bytes para armazenar o tamanho da imagem. A quantidade de bits de *overhead* necessárias para codificar uma imagem é muito pequena e, portanto, terá pouca contribuição no acréscimo da taxa de bits final de codificação da imagem.

Na **Fig. 5.7** é mostrado o diagrama em blocos do sistema de codificação adaptativo proposto para imagens monocromáticas e na **Fig. 5.8** é mostrado o sistema de decodificação. O método pode ser aplicado também à imagens coloridas, bastando fazer a conversão das componentes primárias R, G e B para um novo plano de sinais, que pode ser YIQ (*NTSC - National Television System Comittee*), YUV (*PAL - Phase Alternating by Line*) ou YCrCb (*HDTV - High Definition Television*), e aplicar o método a cada componente individualmente como feito para imagem monocromática.

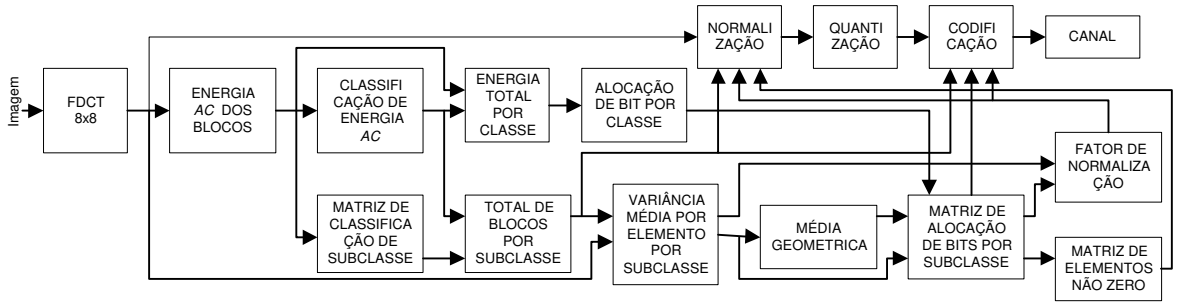


Fig. 5.7 – Sistema de codificação adaptativa de imagens por transformada cosseno.

Para aplicações em tempo real, o processo será realizado em dois passos: no primeiro passo, são realizados os cálculos para a obtenção das informações de *overhead*; no segundo passo, são realizadas a normalização e quantização dos coeficientes transformados, a codificados dos dados, a adição do *overhead* e a transmissão dos dados sobre o canal. Se a

capacidade de armazenamento dos dados for compativel com o tamanho do quadro mais as informações de *overhead*, espera-se que o tempo de processamento tenha um atrazo de no máximo dois quadros.

Na recepção, serão necessárias as informações de *overhead* para a decodificação da imagem. No receptor, os dados de *overhead* são primeiramente armazenados e os quadros em seguida decodificados. Com a aplicação da transformada cosseno inversa, a imagem codificada é recuperada.

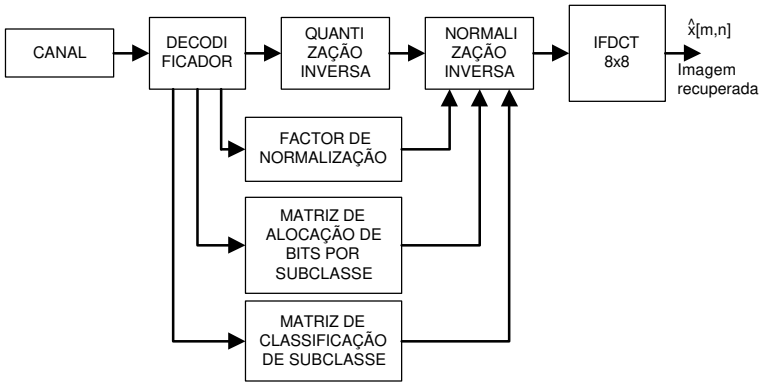


Fig. 5.8 – Sistema de decodificação

5.3. Simulações e Resultados

Apresenta-se a seguir os resultados obtidos por simulação do sistema de codificação proposto usando a FDCT **[144]**. Várias simulações foram realizadas com o objetivo de avaliar o desempenho do sistema proposto de codificação adaptativa. Foram usadas na simulação várias imagens SMPTE (*Society of Motion Picture and Television Engineering*) (512x512), Lenna (512x512) e outras, usando-se blocos de tamanho 8x8. As imagens transformadas foram codificadas a uma taxa média de 1 bit/pixel e quantizadas usando-se o método de quantização ótima de Lloyd-Max com modelo de distribuição laplaciana, gaussiana e uniforme. A avaliação do sistema é feita pelos critérios subjetivo comparando-se

subjetivamente a imagem original e a imagem reconstruída e objetivo, pela medida da relação sinal ruído de pico (PSNR) entre as amostras da imagem original ***x[m,n]*** e as amostras da imagem recuperada $\boldsymbol{x_r[m,n]}$ (Equação (5.27)).

$$PSNR_{dB} = 10 \log_{10} \left(\frac{255^2}{\sum_{i=0}^{N_1-1} \sum_{j=0}^{N_2-1} (x[i,j] - x_r[i,j])^2} \right) \tag{5.27}$$

Os resultados da simulação são comparados com outros métodos de codificação, como: LOT **[142]**, DSTr **[20]**, DCT convencional **[156]** e outros trabalhos relacionados.

Para fazer uma análise comparativa entre alguns métodos de codificação, foi escolhida como padrão a imagem Lenna, com tamanho 512x512. Primeiramente, é aplicada nessa imagem uma codificação usando DCT, LOT e DSTr, com blocos de tamanho 8x8, com uma taxa média de 1 bit/pixel e quantização com distribuição uniforme. É usada, nesse caso, uma única matriz de alocação de bits para os coeficientes $\boldsymbol{b_{ij}}$ dos blocos transformados, onde ***i,j = 0, 1, ..., 7***. O número de níveis de reconstrução para cada coeficiente é dado por:

$$L_{i,j} = 2^{b_{i,j}} \tag{5.28}$$

Esse processamento servirá como parâmetro de comparação. Posteriormente são aplicados outros métodos de codificação.

Mostra-se a seguir as **Figs. 5.9** a **5.12** com as imagens Lenna original e as imagens codificadas e recuperadas com as transformadas DCT, LOT e DSTr, bem como as matrizes de alocação de bits para cada imagem correspondente. Analisando as matrizes de alocação de bits, observa-se a propriedade de concentração de energia nos coeficientes de baixa freqüência de cada transformada correspondente.

Fig. 5.9 – Imagem Lenna (512x512) original

8	5	3	3	2	1	1	0
4	3	3	2	2	1	1	0
2	3	2	2	1	1	0	0
2	2	2	1	1	1	0	0
1	1	1	1	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Fig. 5.10 - Codificada com DCT, taxa 1bpp, PSNR=31,52 dB e uma matriz de alocação de bits.

8	3	2	5	1	3	1	2
2	2	1	2	0	2	0	1
1	1	0	1	0	1	0	0
4	3	1	3	1	2	0	1
1	0	0	1	0	0	0	0
2	1	1	2	0	1	0	0
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0

Fig. 5.11 - Codificada com LOT, taxa 1 bpp, PSNR=31,85 dB e uma matriz de alocação de bits.

8	5	3	2	2	1	1	0
5	5	3	2	2	1	1	1
2	3	2	1	1	1	0	0
2	2	1	1	1	0	0	0
1	1	1	0	0	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Fig. 5.12 - Codificada com DSTr, taxa 1bpp, PSNR=31,17 dB e uma matriz de alocação de bits.

Aplicando-se nessa mesma imagem a codificação usando DCT, LOT e DSTr, com blocos de tamanho 8x8, taxa média de 1 bit/pixel, quantização de Lloyd-Max e considerando que as distribuições dos coeficientes transformados apresentam modelos laplaciana e gaussiana respectivamente, e usando uma única matriz de alocação de bits obteve-se os resultados mostrados na **Tabela 1**.

Transformada	Quantização		
	Laplaciana	Uniforme	Gaussiana
DCT	32.54 dB	31.52 dB	31.20 dB
LOT	32.93 dB	31.85 dB	31.62 dB
DSTr	32.23 dB	31.17 dB	31.09 dB

Tabela 1 – Desempenho de PSNR para a imagem Lenna codificada com DCT, LOT e DSTr, com uma matriz de alocação de bits.

Os resultados das simulações apresentados na **Tabela 1** mostram que o desempenho da codificação da imagem a taxa de 1 bit/pixel com uma única tabela de alocação de bits, usando-se DCT, LOT e DSTr, são aproximadamente iguais para cada modelo de quantização, sendo que a quantização laplaciana apresentou um melhor desempenho em relação as outras.

Analisando-se os métodos apresentados, pode-se concluir que esses métodos de codificação não apresentam uma solução ótima, visto que todos os coeficientes na mesma posição em um bloco são quantizados com o mesmo número de bits, independente da posição do bloco na imagem. Como uma imagem apresenta regiões com os mais diversos comportamentos, certamente, em regiões com muitos detalhes alguns dos coeficientes necessitarão de mais bits para representá-los e em regiões uniformes ou quase uniformes, esses mesmos coeficientes necessitarão de menos bits. O presente trabalho visa exatamente propor uma solução para esse problema e comparar os resultados com outras soluções propostas.

Uma solução para melhorar a alocação dos bits, foi apresentada por Chen-Smith **[126]**. Nesse método, os blocos transformados são analisados e classificados quanto ao nível de energia ***ac***, como visto no capítulo anterior. Nesse método, a análise dos blocos não leva em consideração as características posicionais dos coeficientes significantes, nem as características de bordas presentes nos blocos. Esse algoritmo, apesar de gerar uma informação adicional maior, resultou em um desempenho melhor em relação ao caso anterior. Com esse algoritmo, aplica-se sobre a imagem Lenna uma codificação com taxa média de 1 bit/pixel usando DCT, LOT e DSTr com blocos 8x8, quantização de Lloyd-Max com distribuição laplaciana e classificação dos blocos em 4 classes de energia. Os resultados das simulações são mostrados a seguir nas **Figs. 5.13 – 5.15**.

Fig. 5.13 - Codificada com DCT, taxa 1bpp, PSNR=36.75 e 4 classes de energia.

Fig. 5.14 - Codificada com LOT, taxa 1bpp, PSNR=36.68 e 4 classes de energia.

Fig. 5.15 - Codificada com DSTr, taxa 1 bpp,
PSNR=36.59 e 4 classes de energia.

Pode ser observado que com esse algoritmo houve um ganho de aproximadamente 4 dB em média na PSNR, para cada método de codificação, quando comparado com o método anterior, além de ser observado uma melhoria na qualidade subjetiva das imagens recuperadas. Codificando-se ainda a imagem Lenna com taxas de 1, 2 e 3 bpp e usando-se os mesmos métodos de codificação, tamanho de bloco, quantização laplaciana e 4 classes de energia com alocação adaptativa de bits, obteve-se os resultados de PSNR mostrados na **Tabela 2**.

Transformada	Quantização laplaciana		
	1 bit/pixel	2 bit/pixel	3 bit/pixel
DSTr	36.59 dB	40.05 dB	43.12 dB
LOT	36.68 dB	39.96 dB	42.59 dB
DCT	36.75 dB	40.43 dB	42.92 dB

Tabela 2 - Desempenho de PSNR para a imagem Lenna codificada com
DSTr, LOT e DCT, 4 classes de energia e quantização laplaciana.

Observa-se na **Tabela 2** que as PSNR para esse método de codificação estão bem próximas entre si para taxas de 1, 2 e 3 bpp. Porém, Pelaes e Yuzo **[20]** mostram que a qualidade subjetiva da imagem recuperada é superior quando é usada a DSTr.

No método proposto, os blocos são classificados quanto ao nível de energia ***ac*** e, posteriormente, são analisados e classificados quanto as características posicionais dos coeficientes significantes, baseado na distribuição de potência dos coeficientes, bem como nas características de borda. Como um exemplo da aplicação, a imagem Lenna 256x256 é codificada adaptativamente com o esquema de codificação proposto, com classificação em níveis de energia, formando 4 classes de energia e 3 subclasses conforme a orientação de borda, com taxa média de 1 bit/pixel e considerando que as distribuições dos coeficientes transformados apresentam um modelo laplaciano. As classes de energia são conforme a **Fig. 5.5** e a classificação de subclasses segundo a **Fig. 5.6**. Os resultados obtidos são mostrados nas **Figs. 5.16** e **5.17** e **Tabelas 3** e **4**.

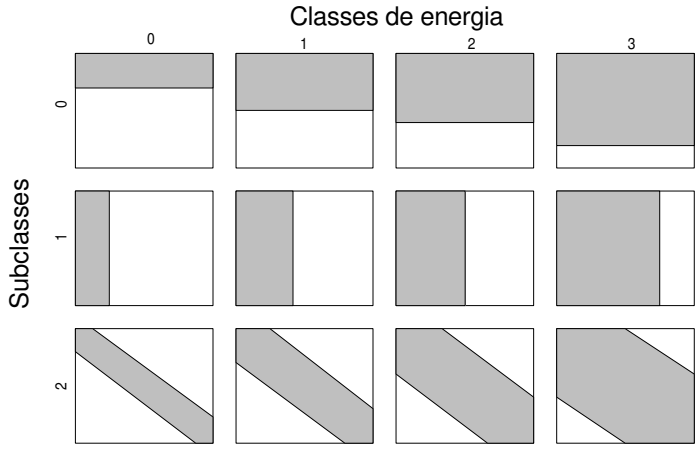


Fig. 5.16 – Classificação dos blocos em classes e subclasses de energia

A quantidade de blocos e número de bits para cada categoria é mostrado na **Tabela 3**.

	Classes de energia			
Subclasses	127 blocos 27 bits	141 blocos 47 bits	172 blocos 75 bits	172 blocos 127 bits
	70 blocos 26 bits	64 blocos 45 bits	52 blocos 78 bits	42 blocos 128 bits
	52 blocos 25 bits	57 blocos 45 bits	32 blocos 75 bits	43 blocos 125 bits
Total	78 bits	137 bits	228 bits	380 bits
Taxa	823 bits/768 pixel = 1.0716 bits/pixel			

Tabela 3 – Distribuição de blocos e número de bits por categoria

A **Fig. 5.17** mostra as matrizes de alocação de bits para as subclasses conforme as orientações estabelecidas na **Fig. 5.16**.

$$\begin{bmatrix} 8 & 3 & 1 & 1 & 1 & 0 & 1 & 1 \\ 2 & 2 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 8 & 4 & 3 & 2 & 2 & 1 & 1 & 0 \\ 3 & 2 & 2 & 1 & 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 8 & 5 & 4 & 3 & 2 & 2 & 2 & 1 \\ 3 & 3 & 3 & 2 & 2 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 8 & 6 & 5 & 4 & 3 & 3 & 3 & 2 \\ 4 & 4 & 4 & 3 & 3 & 3 & 2 & 2 \\ 3 & 3 & 3 & 3 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 8 & 3 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 8 & 3 & 2 & 1 & 0 & 0 & 0 & 0 \\ 3 & 2 & 1 & 1 & 0 & 0 & 0 & 0 \\ 2 & 2 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 8 & 4 & 2 & 1 & 1 & 1 & 1 & 0 \\ 4 & 3 & 2 & 2 & 1 & 0 & 0 & 0 \\ 3 & 3 & 2 & 1 & 1 & 1 & 1 & 0 \\ 3 & 2 & 2 & 1 & 1 & 1 & 1 & 0 \\ 2 & 2 & 2 & 1 & 1 & 1 & 1 & 0 \\ 2 & 2 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 8 & 5 & 4 & 2 & 2 & 2 & 1 & 1 \\ 5 & 4 & 3 & 2 & 2 & 2 & 1 & 1 \\ 4 & 4 & 3 & 2 & 1 & 1 & 1 & 1 \\ 3 & 3 & 3 & 2 & 2 & 2 & 1 & 1 \\ 3 & 3 & 3 & 2 & 2 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 1 & 2 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 8 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 8 & 3 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 2 & 2 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 8 & 4 & 2 & 2 & 1 & 1 & 1 & 1 \\ 3 & 3 & 2 & 1 & 1 & 1 & 1 & 0 \\ 2 & 2 & 2 & 2 & 1 & 1 & 0 & 0 \\ 2 & 2 & 2 & 2 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 8 & 5 & 3 & 2 & 2 & 1 & 1 & 0 \\ 5 & 4 & 4 & 2 & 2 & 2 & 1 & 1 \\ 3 & 4 & 3 & 3 & 3 & 2 & 1 & 1 \\ 2 & 2 & 3 & 3 & 3 & 2 & 1 & 1 \\ 2 & 2 & 2 & 3 & 2 & 2 & 2 & 2 \\ 1 & 1 & 1 & 2 & 2 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 \end{bmatrix}$$

Fig. 5.17 - Matrizes de alocação de bits das subclasses

Quando usado um coeficiente de normalização para cada subclasse dada pela Equação (5.19), e distribuição dos coeficientes transformados com modelos laplaciano e gaussiano, obteve-se os resultados indicados na **Tabela 4**, para a imagem Lenna com tamanhos 512x512 e 256x256 respectivamente para 4 classes de energia e 3 subclasses.

Imagem	Quantização	
	laplaciana	gaussiana
Lenna512x512	38.69 dB	38.45 dB
Lenna256x256	33.88 dB	33.80 dB

Tabela 4 - Desempenho de PSNR para a imagem Lenna codificada pelo método adaptativo proposto.

Analisando-se os resultados obtidos, é visto na **Tabela 2** que com a utilização de alocação adaptativa de bits, com 4 classes de energia, a PSNR apresentou uma melhora significativa com relação ao primeiro método mostrado na **Tabela 1**, para uma taxa média de 1 bit/pixel. Porém, com a codificação utilizando-se o método proposto pelo esquema da **Fig. 5.7**, o sistema apresentou uma melhoria de desempenho de aproximadamente 2 dB de PSNR como mostra a **Tabela 4**, quando comparado com os dados da **Tabela 2** para a mesma taxa. As **Figs. 5.18** a **5.21** mostram a imagem Lenna 512x512 original e as imagens codificadas e recuperadas com taxa média de 1 bit/pixel usando método ótimo de quantização de Lloyd-Max, com modelos de função de densidade de probabilidade laplaciana e gaussiana, utilizando-se 4 classes de energia e usando o modelo adaptativo proposto com 4 classes e 3 subclasses.

Fig. 5.18- Imagem 512x512 original

Fig.5.19 – Quantização laplaciana e 4 classes

Fig. 5.20 - Quantização gaussiana 4 clas./3sub

Fig. 5.21 – Quantização laplaciana 4clas./3sub

Analisando-se subjetivamente as **Figs. 5.20** e **5.21**, observa-se que não existe diferença perceptível quando comparadas com a imagem original (**Fig. 5.18**). Quando comparadas com a imagem na **Fig. 5.19**, observa-se uma ligeira diferença nos detalhes do chapéu, que aparecem um pouco menos realçados.

Codificando-se a imagem Lenna 512x512 adaptativamente com as mesmas condições anteriores, porém acrescendo-se uma quarta subclasse (triangular superior) conforme **Fig. 5.22**, para taxa média de 1 bit/pixel e considerando-se que as distribuições dos coeficientes transformados têm modelos laplaciano e gaussiano, respectivamente, obteve-se os resultados indicados na **Tabela 5** e **Fig. 5.23**.

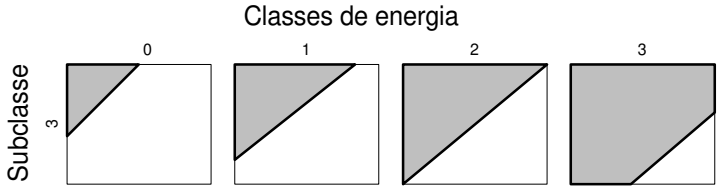


Fig. 5.22 – Classificação dos blocos em classes e subclasses de energia

Classificação	Quantização	
	Laplaciana	Gaussiana
4class/3sub	38.69	38.45
4class/4sub	38.73	38.53

Tabela 5 – Desempenho de PSNR para imagem Lenna codificada com média de 1 bit/pixel.

Fig. 5.23 - Quantização laplaciana 4clas./4sub

Com o acréscimo de uma quarta subclasse, foi observado que houve uma pequena melhora na PSNR. Conclui-se, portanto, não ser justificável o acréscimo de uma quarta subclasse, visto que isso resulta em um aumento de *overhead*.

Com o objetivo de melhor avaliar o desempenho do sistema de codificação adaptativo proposto, outras imagens foram codificadas com as mesmas condições anteriores, com classificação em níveis de energia, formando 4 classes de energia e 3 subclasses e, posteriormente, com 4 subclasses conforme a orientação de borda e considerando que as distribuição dos coeficientes transformados apresentam os modelos laplaciana e gaussiana. Os resultados obtidos são mostrados na **Tabela 6**.

Para uma avaliação subjetiva, é mostrado nas **Figs. 5.24 – 5.32** as imagens originais e recuperadas após a codificação com o sistema proposto. Como pode ser observado, através de

uma análise subjetiva, as imagens codificadas e recuperadas não apresentam diferenças perceptíveis quando comparadas com as imagens originais.

Imagens	Classificação	Quantização	
		laplaciana	gaussiana
Cozinha 512x512	4class/3sub-class	44,41	44,31
Zelda 512x512		44,36	43,38
Bird 256x256		43,02	43,27
Cozinha 512x512	4class/4sub-class	42,83	42,86
Zelda 512x512		41,64	41,14
Bird 256x256		40,24	40,17

Tabela 6 - Desempenho de PSNR para as imagens codificadas adaptativamente por DCT com o sistema proposto.

Fig. 5.24 – Imagem original "Cozinha"

Fig. 5.25 – Imagem codificada com 4 classes e 3 subclasses, laplaciana.

Fig. 5.26 - Imagem codificada com 4 classes e 4 subclasses, laplaciana.

Fig. 5.27 – Imagem original "Zelda"

Fig. 5.28 – Imagem codificada com 4 classes e 3 subclasses, laplaciana.

Fig. 5.29 – Imagem codificada com 4 classes e 4 subclasses, laplaciana.

Fig. 5.30 – Imagem original “Bird”

Fig. 5.31 – Imagem codificada com 4 classes e 3 subclasses, laplaciana

Fig. 5.32 – Imagem codificada com 4 classes e 4 subclasses, laplaciana.

O número de resultados obtidos nas simulações é relativamente grande, tornando impraticável a apresentação de todas as imagens processadas e recuperadas para uma análise subjetiva das mesmas. Porém, é apresentado a seguir algumas Tabelas com alguns dados objetivos resultantes do processo de simulação do sistema. Com esses dados, pode-se fazer uma melhor análise dos resultados.

Comparação da Relação Sinal/Ruído, taxa de compressão, e método de compressão.

Tabela 7 - Codificação com 4 Classes de Energia: Quantização por Laplace e Gauss, Coeficiente de Normalização $x/(f_n\, 2^{bij-1})$ (f_n = maior σ para bij = 1), Imagem Lenna, 256x256, Blocos DCT 8x8, *Overhead* = 0.811767 %.

Método	Quantizador	bit/pixel	Bits/Classe	PSNR
4 Classes	Laplace	1.082031	30,50,76,121	31.158860
4 Classes	Gauss	1.082031	30,50,76,121	30.924276
4 Classes	Laplace dc-8bpp	1.000000	25,46,69,116	30.666890
4 Classes	Gauss dc-8bpp	1.000000	25,46,69,116	30.392169

Tabela 8 - Imagem Lenna, 512x512, Blocos DCT 8x8, *Overhead* = 0.202942 %.

Método	Quantizador	bit/pixel	Bits/Classe	PSNR
4 Classes	Laplace	1.183593	30,47,78,148	36.752084
4 Classes	Gauss	1.183593	30,47,78,148	36.371900
4 Classes	Laplace dc-8bpp	1.078125	22,38,67,149	36.180787
4 Classes	Gauss dc-8bpp	1.078125	22,38,61,159	35.779283

Tabela 9 - Codificação com 4 Classes de Energia: Quantização por Laplace e Gauss, Coeficiente de Normalização $(x - \mu) / \sigma$, Imagem Lenna, 256x256, Blocos DCT 8x8, *Overhead* = 2.37426758 %.

Método	Quantizador	bit/pixel	Bits/Classe	PSNR
4 Classes	Laplace	1.082031	30,50,76,121	31.620557
4 Classes	Gauss	1.082031	30,50,76,121	31.439385
4 Classes	Laplace dc-8bpp	1.000000	25,46,69,116	31.189927
4 Classes	Gauss dc-8bpp	1.000000	25.46.69.116	31.042756

Tabela 10 - Imagem Lenna, 512x512, Blocos DCT 8x8, *Overhead* = 0.59356689 %.

Método	Quantizador	bit/pixel	Bits/Classe	PSNR
4 Classes	Laplace	1.183593	30,47,78,148	37.103440
4 Classes	Gauss	1.183593	30,47,78,148	36.737672
4 Classes	Laplace dc-8bpp	1.078125	22,38,67,149	36.333535
4 Classes	Gauss dc-8bpp	1.078125	22,38,67,149	35.980264

Tabela 11 - Codificação com 4 Classes de Energia e 3 Subclasses: Quantização por Laplace e Gauss, Coeficiente de Normalização $x/(f_n\, 2^{bij-1})$ (f_n = maior σ para bij = 1), Imagem Lenna, 256x256, Blocos DCT 8x8, *Overhead* = 2.8137207 %.

Método	Quantizador	bit/pixel	Bits/Classe	PSNR
4 Cl/3Sub	Laplace	1.071614	27,47,75,127 26,45,78,128 25,45,75,125	33.883976
4 Cl/3Sub	Gauss	1.071614	Idem	33.799427
4 Cl/3Sub	Laplace dc-8bpp	1.040365	29,47,74,119 28,51,71,120 26,46,71,117	33.332268
4 Cl/3Sub	Gauss dc-8bpp	1.040365	Idem	33.248058

Tabela 12 - Imagem Lenna, 512x512. Blocos DCT 8x8, *Overhead* = 1.875305 %.

Método	Quantizador	bit/pixel	Bits/Classe	PSNR
4 Cl/3Sub	Laplace	1.131510	32,57,81,125 30,55,80,125 26,46,71,117	38.693462
4 Cl/3Sub	Gauss	1.131510	Idem	38.450367
4 Cl/3Sub	Laplace dc-8bpp	1.160156	26,40,73,159 26,41,72,155 23,39,74,163	38.314266
4 Cl/3Sub	Gauss dc-8bpp	1.160156	Idem	38.113895

RESULTADOS DAS SIMULAÇÕES REALIZADAS COM OUTRAS IMAGENS

Tabela 13 - Codificação com 4 Classes de Energia e 3 Subclasses: Quantização por Laplace, Coeficiente de Normalização $x/(f_n\, 2^{bij-1})$ (f_n = maior σ para bij = 1), Imagem BIRD, 256x256, Blocos DCT 8x8, *Overhead* = 2.8137207 %.

Método	Quantizador	bit/pixel	PSNR
4 Cl/3Sub	Laplace dc-8bpp	0.734375	43.022568

Tabela 14 – Imagens: Cozinha, Zelda e Sala respectivamente, 512x512, Blocos DCT 8x8, *Overhead* = 1.875305 %.

Método	Quantizador	bit/pixel	PSNR
4 Cl/3Sub	Laplace dc-8bpp	1.272035	44.406796
4 Cl/3Sub	Laplace dc-8bpp	1.226562	44.358238
4 Cl/3Sub	Laplace dc-8bpp	1.266927	44.126857

RESULTADOS DAS SIMULAÇÕES REALIZADAS COM 4 SUBCLASSES

Tabela 15 - Codificação com 4 Classes de Energia e 4 Subclasses: Quantização por Laplace e Gauss, Coeficiente de Normalização $x/(f_n\, 2^{bij-1})$ (f_n = maior σ para bij = 1), Imagem Lenna, 512x512, Blocos DCT 8x8, *Overhead* = 1.979064941 %.

Método	Quantizador	bit/pixel	Bits/Classe	PSNR
4 Cl/4Sub	Laplace	1.278320	34,51,82,166 34,48,83,164 26,46,82,166 34,48,41,164	38.733635
4 Cl/4Sub	Gauss	Idem	Idem	38.526489
4 Cl/4Sub	Laplace dc-8bpp	1.173828	27,43,72,155 29,43,76,161 22,38,74,160 28,43,70,161	38.281849
4 Cl/4Sub	Gauss dc-8bpp	Idem	Idem	38.145763

Tabela 16 - Imagem Cozinha, 512x512, Blocos DCT 8x8, *Overhead* = 1.979064941 %.

Método	Quantizador	bit/pixel	Bits/Classe	PSNR
4 Cl/4Sub	Laplace	1.416992	30,48,85,208 27,49,89,197 21,44,90,197 27,46,87,200	42.833107
4 Cl/4Sub	Gauss	Idem	Idem	42.858665
4 Cl/4Sub	Laplace dc-8bpp	1.308594	26,41,78,194 22,42,86,192 14,38,83,191 24,41,78,190	42.443520
4 Cl/4Sub	Gauss dc-8bpp	Idem	Idem	42.340488

Tabela 17 - Imagem Zelda, 512x512, Blocos DCT 8x8, *Overhead* = 1.979064941 %.

Método	Quantizador	bit/pixel	Bits/Classe	PSNR
4 Cl/4Sub	Laplace	1.369141	27,46,76,194 30,49,81,190 27,51,82,193 31,46,84,195	41.635773
4 Cl/4Sub	Gauss	Idem	Idem	41.136406
4 Cl/4Sub	Laplace dc-8bpp	1.227539	25,40,65,188 25,38,76,182 15,40,66,185 25,40,64,183	40.976070
4 Cl/4Sub	Gauss dc-8bpp	Idem	Idem	40.723980

Tabela 18 - Imagem BIRD, 256x256, Blocos DCT 8x8.

Método	Quantizador	bit/pixel	Bits/Classe	PSNR
4 Cl/4Sub	Laplace	0.840820	26,29,52,116 31,29,51,109 22,25,51,106 26,29,52,107	40.238998
4 Cl/4Sub	Gauss	Idem	Idem	40.169426
4 Cl/4Sub	Laplace dc-8bpp	0.757812	20,24,47,109 24,27,47,101 17,22,44,101 23,25,46,99	39.797310
4 Cl/4Sub	Gauss dc-8bpp	Idem	Idem	39.729656

As **Tabelas 7** a **18** mostram outros resultados obtidos por simulação do sistema proposto. As **Tabelas 7** a **10** apresentam os resultados das simulações de codificação com 4 classes de energia para quantização laplaciana e gaussiana com imagens Lenna, com tamanho 256x256 e 512x512. Nas **Tabelas 7** e **8** é usado como coeficiente de normalização o fator $x/(f_n\ 2^{bij\ -\ 1})$, onde f_n é igual ao maior σ do bloco para o qual bij = 1. Nas **Tabelas 9** e **10** é usado como coeficiente de normalização o fator $(x - \mu) / \sigma$. É visto nas **Tabelas 7** a **10** que quando o coeficiente DC foi codificado com 8 bits, houve uma pequena diminuição da PSNR. Isso deve-se ao fato do deslocamento de bits de outros coeficientes para o coeficiente DC. É visto também que quando foi usado o fator de normalização $(x - \mu) / \sigma$, houve uma pequena melhora na PSNR.

As **Tabelas 11** e **12** mostram os resultados das simulações de codificação com 4 classes e 3 subclasses de energia para quantização laplaciana e gaussiana com as imagens Lenna com tamanho 256x256 e 512x512, usando-se como coeficiente de normalização o fator $x/(f_n\ 2^{bij\ -\ 1})$. É observado que quando o coeficiente DC foi codificado com 8 bits, houve também uma pequena diminuição da PSNR para ambos os tamanhos das imagens.

As **Tabelas 13** e **14** mostram os resultados das simulações para as imagens BIRD 256x256, Cozinha, Zelda e Sala 512x512, para as mesmas condições adotadas nas **Tabelas 11** e **12**. Para essas imagens, mostra-se que houve uma expressiva melhora na PSNR quando comparado com os resultados nas **Tabelas 11** e **12**, o que mostra que o desempenho do codificador proposto é ajustável ao tipo de imagem que está sendo processada, buscando sempre a menor taxa e mantendo a qualidade subjetiva da imagem.

As **Tabelas 15** a **18** mostram os resultados das simulações de codificação com 4 classes e 4 subclasses de energia para quantização laplaciana e gaussiana para as imagens Lenna, Cozinha, Zelda 512x512 e Bird 256x256, respectivamente. Observa-se que os resultados da **Tabela 15** apresentaram uma pequena melhora da PSNR quando comparado com os resultados da **Tabela 12**. Porém, os resultados das **Tabelas 16** a **18** apresentaram uma diminuição da PSNR quando comparados com as **Tabelas 13** e **14**. Com esses resultados, pode-se concluir que o acréscimo de uma quarta subclasse de energia poderá ou não acarretar em uma melhora da PSNR, dependendo, portanto, do tipo de imagem que está sendo processada. Porém, tanto o processamento com 4 classes e 3 subclasses como com 4 classes e 4 subclasses apresentaram uma melhoria da PSNR quando comparado com os outros métodos de codificação.

CAPÍTULO 6

Conclusões

6.1 – Considerações sobre as transformadas

Com o advento de digitalização, a demanda por serviços que usam sinais digitalizados tem crescido consideravelmente, levando à necessidade de desenvolvimento de sistemas capazes de integrar vários serviços com características diferentes em uma mesma rede, surgindo portanto, a Rede Digital de Serviços Integrados. Para essa rede se tornar uma rede completa, é necessário que a mesma suporte também os serviços e demandas de informações que envolvem os sinais de imagens digitalizadas. Devido ao seu campo de aplicação ser muito grande, passando pelas áreas de automação industrial, segurança, saúde, entretenimento, espacial, histórico, memorial, etc, o processamento digital de imagem se tornou o objeto de estudo em todo o mundo, com o objetivo de absorção do domínio das técnicas de manipulação das mesmas.

Os sinais de imagem são conhecidos por ter um grau de complexidade muito alto quanto ao seu processamento, uma vez que esses sinais apresentam normalmente uma largura de faixa elevada, bem como detalhes que dependendo da aplicação se tornam um complicador no processamento. Especificamente, nos sinais de imagens destinados a transmissão e/ou armazenamento, a largura de faixa é uma característica preocupante para o sistema de processamento, devido às limitações de faixa do canal utilizado, bem como do tempo necessário para sua transmissão ou espaço para o armazenamento. Quanto aos detalhes presentes na imagem, dependendo do usuário, esses terão maior ou menor importância.

O processamento digital de sinais teve um crescimento considerável a partir do desenvolvimento de um algoritmo rápido e eficiente da Transformada Discreta de Fourier (DFT) proposto por Cooley e Tukey em 1965. Em 1974, o processamento digital de sinais teve um novo impacto quando Ahmed, Natarajan e Rao apresentaram a Transformada Cosseno

Discreta (DCT) baseada na Transformada Rápida de Fourier (FFT). Porém, o principal ponto de inovação aconteceu em 1977 quando Chen, Smith e Fralick apresentaram um algoritmo rápido e eficiente para a implementação da DCT baseada em aritmética real. A partir daí, vários outros algoritmos rápidos foram propostos na literatura. A DCT, desde então, tem sido a transformada mais utilizada em processamento digital de imagens pelo fato de a mesma ter apresentado um desempenho superior às demais transformadas, exceto a KLT, sendo por isso considerada uma transformada sub-ótima.

A DCT nos dias de hoje apresenta uma grande variedade de possibilidades de aplicação no campo de processamento digital de sinais, tanto de voz como de imagens. No campo de processamento de imagens, o presente trabalho procurou abordar apenas os aspectos de codificação voltados para a compressão. A codificação de imagens por transformada para a compressão, é o procedimento que busca a representação de uma imagem com um número de bits o menor possível, capaz de, na recuperação, representá-la com uma certa qualidade e inteligibilidade, baseado em um certo critério de fidelidade necessário para uma dada aplicação. O objetivo do emprego da transformada em processamento de imagens digitalizadas é mapear as amostras da imagem que estão no domínio espacial para o domínio da transformada. Essa mudança de domínio possibilita a extração ou até a eliminação de redundâncias existentes no domínio espacial, bem como a eliminação de informações irrelevantes. A transformada que realiza melhor esse tipo de tarefa é a DCT. Sendo por isso, a transformada mais usada na codificação de imagens e adotada nos padrões JPEG e MPEG.

Como a eficiência da transformada está na sua habilidade de descorrelacionar as amostras vizinhas, a mesma não é aplicada na imagem como um todo e sim em blocos de pequenas dimensões (4x4, 8x8, 16x16 etc.). Quanto maior o tamanho do bloco, menor é a eficiência da transformada. Por outro lado, quanto menor o tamanho do bloco, maior é o esforço computacional necessário para processar a imagem. O tamanho de bloco 8x8 é o mais usado no processo de compressão de imagens, sendo que em alguns casos (MPEG-2) é usado também blocos com tamanho 16x16.

Em sistemas de compressão de imagens, o objetivo é representar a imagem com o menor número possível de bits. Como a transformação é feita por pequenos blocos, a imagem recuperada apresenta um problema de descontinuidade entre blocos adjacentes (efeito de

bloqueamento), que é inerente ao processo, quando é usada compressão em baixas taxas. Com o objetivo da eliminação desse efeito, vários outros métodos de codificação de imagens por blocos têm sido propostos. Entre os métodos conhecidos se encontram a quantização vetorial e suas várias possibilidades, e outras transformadas tais como a LOT, transformada *wavelet*, a DSTr etc.

A transformada tem a propriedade de concentrar a energia do bloco em alguns poucos coeficientes transformados (coeficientes de baixa frequência). Portanto, para a compressão de imagens com baixas taxas, a transformada é complementada com técnicas de quantização e alocação de bits que fornecerão um certo grau de qualidade à imagem recuperada. Para aumentar a extração de redundância, pode-se ainda usar a codificação de entropia para representar os coeficientes quantizados.

Outros métodos muito usados na codificação de imagens são os que usam a técnica de quantização vetorial. Esses métodos têm mostrado resultados excelentes na compressão de imagens para baixas taxas. Porém, para processamento em blocos maiores que 5x5 esses métodos são praticamente inviáveis. A dificuldade na utilização desse método reside no fato de o mesmo estar baseado na construção de um *codebook* onde são armazenados as palavras código. A construção do *codebook* é realizada através do treinamento de um banco de imagens, denominadas de imagens base que podem estar no domínio espacial ou transformada. Como o processamento das imagens é feito por blocos, cada bloco pode apresentar uma infinidade de níveis diferentes, além do posicionamento e orientação de bordas dentro do bloco. Essas variáveis requerem a construção de um *codebook* bastante grande para que o método seja eficiente no processamento de imagens genéricas. Como o *codebook* normalmente tem tamanho limitado com 32, 64, 128, 256 palavras código, imagens que não fizeram parte da construção do *codebook* não serão eficientemente codificadas. Uma das soluções propostas para compensar as dificuldades do uso da quantização vetorial e as deficiências do uso das transformadas foi o uso combinado de ambas, DCT-VQ, sendo que, com esse objetivo, vários trabalhos foram propostos para a compressão de imagens.

6.2 – Resultados deste trabalho

Este trabalho apresentou como contribuição uma proposta que combina as técnicas de classificação dos blocos, usadas em VQ e em DCT-VQ para a construção do *codebook*, e que está baseada nas bordas presentes nos blocos bem como suas orientações e as técnicas de classificação dos blocos por classes de energia, porém, levando em consideração os posicionamentos dos coeficientes de maior energia dentro de cada bloco. Essa combinação possibilitou o desenvolvimento de um sistema adaptativo de codificação de imagem para compressão em baixas taxas independente da imagem a ser processada. O sistema proposto usa a DCT como transformada base e a adaptatividade é conseguida com a distribuição dos bits entre os blocos pertencentes às várias classes e subclasses.

Neste trabalho foi implementado o sistema de codificação adaptativo proposto. O sistema foi aplicado a várias imagens com o objetivo de se fazer uma comparação subjetiva e objetiva com outros sistemas de codificação. Para se fazer uma comparação relativa, foram desenvolvidos também os sistemas que usam as transformadas DCT e LOT. Um outro método de codificação que usa a DSTr-2D, proposto por Pelaes e Yuzo, foi também usado como parâmetro de comparação. A DCT foi escolhida por ser a transformada mais usada no processamento de imagens, por apresentar um melhor desempenho comparado com as outras transformadas e por apresentar um forte efeito de bloqueamento quando usada na codificação de imagens a baixas taxas. A LOT foi escolhida por apresentar um reduzido efeito de bloqueamento e um bom desempenho quando usada na codificação a baixas taxas, e a DSTr por ter apresentado desempenho comparável a DCT e a LOT quando foi usada em baixas taxas, além de apresentar menor efeito de bloqueamento que as demais.

Para se obter um melhor parâmetro de comparação, foi escolhida a imagem Lenna com tamanho 512x512 como a imagem padrão, e sobre esta foram aplicados os métodos de codificação com uma taxa média de 1 bit/pixel. Primeiramente, foram aplicados os métodos de codificação com DCT, LOT e DSTr sobre a imagem, usando apenas uma única matriz de alocação de bits e considerando que os coeficientes transformados apresentavam uma distribuição uniforme, gaussiana e laplaciana, sendo aplicada a quantização ótima de Lloyd-Max e com alocação dos bits pela regra do log da variância. Os resultados obtidos com esse

processo foram mostrados na **Tabela 1** (Capítulo 5) onde é observado que a distribuição com modelo laplaciano apresentou melhor PSNR, o modelo uniforme apresentou um resultado médio e o modelo gaussiano o pior resultado, porém, sem vantagens significativas para nenhum método. Foi mostrado apenas a imagem recuperada com o modelo uniforme de quantização. Foi observado que as imagens recuperadas com esse modelo apresentaram um forte efeito de bloqueamento, sendo que o mais acentuado foi quando da aplicação da DCT. Após essa avaliação, foi aplicado o algoritmo proposto por Chen e Smith que usa o método de classificação de energia dos blocos. Com esse método, foram utilizados quatro matrizes de alocação de bits de tal forma que a taxa média de bits resultasse em 1, 2 e 3 bit/pixel, porém, com blocos 8x8 (Chen usa blocos 16x16). Os resultados foram mostrados na **Tabela 2** (Capítulo 5). Pode ser observado que com esse método obteve-se um ganho de aproximadamente 4 dB em média na PSNR, para cada esquema de codificação, além de ser observado uma melhoria na qualidade subjetiva das imagens recuperadas quando comparado com os métodos anteriores como mostra as **Figs. 5.13** a **515**.

Da mesma forma, o modelo proposto foi aplicado à imagem Lenna 512x512 com taxa média de 1 bit/pixel, usando os modelos de distribuição gaussiano e laplaciano. Os resultados foram mostrados nas **Tabelas 4** e **5** (Capítulo 5). Pode ser observado que com o método proposto obteve-se um ganho de aproximadamente 2 dB em média de PSNR quando comparado com o método proposto por Chen e Smith, usando-se a DCT, LOT e DSTr para a mesma taxa, como pode ser observado comparando as **Tabelas 2** e **4** (Capítulo 5), além de ser observado uma melhoria significativa da qualidade subjetiva das imagens recuperadas como pode ser visto nas **Figs. 5.19 - 5.21** e **5.23**. O método proposto pode ser aplicado também à imagens coloridas, bastando fazer a conversão das componentes primárias R, G e B para um novo plano de sinais que pode ser YIQ, YUV ou YCrCb, e aplicar o método a cada componente individualmente, como é feito na imagem monocromática. Para aplicações em tempo real, espera-se que o processo tenha um atraso de no máximo dois quadros, visto que será necessário fazer a obtenção das informações requeridas para a compressão e de *overhead*. Isso será possível se a capacidade de armazenamento dos dados for compatível com o tamanho do quadro mais o *overhead*. No receptor, os dados de *overhead* são primeiramente

armazenados e os quadros em seguida decodificados. Com a aplicação da transformada cosseno inversa, a imagem codificada é recuperada.

Enfim, o método proposto mostrou-se eficiente quanto à qualidade subjetiva da imagem recuperada quando comparado os resultados com outros métodos para as mesmas condições. Além disso, mostrou-se ser um método ótimo, independente da imagem, pois esse método procura se ajustar à complexidade da imagem a ser processada apresentando uma taxa de compressão ótima, diferente para cada tipo de imagem.

6.3 – Sugestões para trabalhos futuros e correlatos

Com o desenvolvimento de processadores rápidos, esse método poderá ser implementado e aplicado em sistemas de compressão de imagens, onde hoje é usada a DCT ou outra transformada. Portanto, o aprofundamento dos estudos e aperfeiçoamento desse método poderão ser realizados no sentido de buscar sua aplicação em áreas como:

- Vídeoconferência;
- Transmissão de imagens de vídeo em baixas taxas;
- Transmissão de sinais de TV digital em Radiodifusão;
- Compressão de imagens com perda para armazenamento;
- Codificação de imagens usando DSTr com quantização Vetorial.
- Utilização em sistemas de HDTV para compressão intraquadro;
- Utilização para transmissão de imagens via redes digitais integradas;
- Utilização do sistema combinado com outros métodos.

6.4 – Conteúdo do Trabalho

Este trabalho foi dividido em capítulos cujos conteúdos são descritos de forma resumida conforme mostrados a seguir:

No Capítulo 1, procurou-se mostrar uma visão geral deste trabalho, dando enfoque aos objetivos, vantagens e limitações do uso da codificação de imagens por transformada para a transmissão e ou armazenamento. Procurou-se mostrar também os vários aspectos relacionados com a codificação de imagens através de representação em esquema de blocos dos processos envolvidos.

No Capítulo 2, procurou-se mostrar a evolução do desenvolvimento da transformada DCT, tendo como base a Transformada de Fourier (FT) e a Transformada Cosseno de Fourier (FCT). Procurou-se mostrar os vários tipos de DCT, desde a DCT do Tipo I até a DCT do Tipo IV, sendo que o maior enfoque foi dado a DCT do Tipo II, visto que essa foi utilizada neste trabalho. Procurou-se mostrar também os efeitos da aplicação da DCT-1D e DCT-2D em seqüências periódicas obtidas através da repetição de seqüências simétricas e não simétricas em relação a seu ponto central.

No Capítulo 3, procurou-se mostrar os aspectos relacionados às propriedades da transformada DCT, sua aplicação em blocos da imagem, os aspectos relacionados à seleção dos coeficientes transformados e à ordenação dos coeficientes escolhidos. Procurou-se mostrar também os aspectos relacionados à quantização dos coeficientes transformados, o projeto de quantizadores e as técnicas de alocação de bits para os coeficientes mais significativos.

No Capítulo 4, procurou-se mostrar os aspectos relacionados à aplicação da DCT em compressão de imagens, analisando-se vários esquemas propostos na literatura para a codificação de imagens usando combinação com outras técnicas de codificação.

No Capítulo 5, é mostrada a proposta do esquema de codificação adaptativa de imagens usando a DCT. Nesse capítulo, são abordados os vários aspectos relacionados aos processos envolvidos na classificação dos blocos, escolha dos coeficientes, alocação de bits e codificação dos blocos transformados da imagem. São mostrados também os resultados obtidos através da simulação do sistema, bem como as avaliações objetivas dos mesmos.

No Apêndice A, são apresentadas as tabelas dos quantizadores ótimos de Lloy-Max para as funções densidade de probabilidade laplaciana e gaussiana, usadas para a quantização dos coeficientes transformados no processo de codificação proposto.

No Apêndice B, são apresentados os aspectos relacionados ao desenvolvimento do algoritmo rápido para a implementação da FDCT utilizada no esquema proposto.

No Apêndice C, são apresentadas as listagens dos programas desenvolvidos para a simulação do sistema proposto.

6.5 – Destaques das contribuições realizadas

Uma das dificuldades encontradas em codificação de imagens usando Quantização Vetorial é a implementação do *codebook* quando os blocos a serem processados aumentam de tamanho. Outra dificuldade é a degradação das qualidades subjetiva e objetiva das imagens que não fizeram parte do treinamento para a composição do *codebook*. Essas imagens apresentam características diferentes das que serviram de base para a composição do dicionário. Em codificação de imagens por transformada, as dificuldades encontradas estão relacionadas com as degradações das qualidades subjetiva e objetiva das imagens recuperadas, devido ao efeito de bloco apresentado, quando as imagens são codificadas em baixas taxas.

O esquema proposto visa superar essas dificuldades buscando sempre um compromisso ótimo entre *taxa de bits* x *qualidade da imagem recuperada*, através da análise de comportamento dos blocos transformados e da estatística dos coeficientes, independentes da imagem a ser processada. Para cada imagem, o esquema proposto busca uma taxa ótima de bit/pixel que mantenha a qualidade subjetiva da mesma. Além disso, todos os dados necessários para a decodificação da imagem codificada se encontram presentes na informação de *overhead*, exceto as tabelas do quantizador ótimo de Lloyd-Max que devem estar presentes no decodificador.

Os resultados obtidos neste trabalho foram apresentados em detalhes no Item 6.2. Cabe ressaltar que para a implementação do codificador e decodificador propostos, foram desenvolvidos todos os programas necessários para a simulação dos mesmos. Esses programas

foram escritos em linguagem C padrão ANSI e se encontram listados no Apêndice C. O pacote de programas está dividido em programas principais e auxiliares. Os auxiliares são compostos de funções de manipulação de arquivos, vetores, matrizes e manipulação de memória. Os programas principais são os que contém a função *main( )* e somente esses geram os programas executáveis. Cada programa principal usa somente algumas funções auxiliares. O desenvolvimento desses programas fazem parte também das contribuições deste trabalho.

Dessa forma, foram apresentadas as contribuições do presente trabalho, bem como os resultados obtidos por simulação do sistema proposto. É esperado que o presente trabalho sirva como referência não só para outros trabalhos, como também para o próprio aprofundamento dos estudos do mesmo em outras aplicações para processamento digital de imagens.

REFERÊNCIAS

[1] Musmanh, H. G.; "A comparison of extended differential coding schemes for video signals", Zurich Seminar, pp. C1(1) – C1(7), 1974.

[2] Jain, Anil K., "Fundamentals of Digital Image Processing", Prentice-Hall,1987.

[3] Lim, Jae S., "Two-dimensional Signal Processing and Image Processing", Prentice-Hall,1990.

[4] Jayant, N. S., and P. Noll, "Digital Coding of Waveforms", Prentice-Hall, 1984

[5] W. K. Pratt, "Image Transmission Techniques", Advances in Eletronics and Electron Physics, Suplement 12, Academic Press, 1979.

[6] H. Murakami; Y.Ratori; H.Yamamoto. "Comparison between DPCM and Hadamard Transform Coding in the Composite Coding of the NTSC Color TV Signal", IEEE Trans. on Comm. Vol. COM-30, N^{o}, March 1982.

[7] C. K. P. Clarke, "Hadamard Transformations: Assessment of Bit Rate Reduction Methods", BBC Res. and Department Rep. 1976/28.

[8] Poularikas, Alexander D., "The Transforms and Applications Handbook ", CRC Press and IEEE Press, 1996.

[9] K. R. Rao and P. Yip, "Discrete Cosine Transform: Algorithms, Advantages and Applications". Academic Press, INC, 1990.

[10] Malvar, Henrique S., " Signal Processing with Lapped Transform ", Artech House, 1992.

[11] Clarke, R. J., " Relation Between de Karhunen-Loève and Cosine Transforms ", Proc.IEE, Pt. F, vol. 128, pp. 359-360, Nov. 1981.

[12] K. A. Prabhu; A.W.Netravali, "Motion Compensated Component Color Coding", IEEE Trans. on Comm. Vol. COM-30, N^{o}12, December 1982.

[13] R. H. Stafford, "Digital Television - Bandwidth Reduction and Communication Aspects", J.Wiley & Sons, 1980.

[14] Y. Iano, "Digitalização de Sinais de TV Através de um Sistema MCPD com Predição e Quantização", Tese de Doutorado, DECOM/FEE/UNICAMP, Janeiro de 1986.

[15] J. B. T. Yabu-Uti; N. Alens; Y. Iano; A. O. Alonso; L.C.Martini, Redução da Taxa de Bits para Transmissão do Sinal de TV Composto PAL-M", Contrato TELEBRÁS 033/80, Relatório Técnico RT-99, Março l983.

[16] I. Dinstein, "DPCM Prediction for NTSC Cornposite Signals", COMSAT Technical Review, Vol. *7,* N°2, pp. 429-446, Feb 1977.

[17] K. Sawada; H.Kotera, "32 Mbit/s Transmission of NTSC Color TV Signals by Composite DPCM Coding", IEEE Trans, on Comm., Vol. COM-26, N°2, pp. 1432- 1439, October 1978,.

[18] H. S. Malvar, and D. H. Staelin, "Reduction of Blocking Effect in Image Coding with Lapped Orthogonal Transform", IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing, New York, pp. 781-748, April 1988.

[19] H. S. Malvar and D. H. Staelin, "The LOT: Transform Coding Without Blocking Effect", IEEE Trans. on Acoustics, Speech, and Signal Processing, vol. 37, april 1989.

[20] Pelaes, E. G. and Yuzo Iano. *"Image Coding Using Discrete Sine Transform with Axis Rotation,"* IEEE Trans. on Consumer Electronics, vol. 44 N° 4 pp. 1284-1290, Nov. 1998.

[21] Majid Rabbani and Paul W. Jones, "Digital Image Compression Techniques", SPIE Optical Engeneering Press, 1991.

[22] International Organization for Standard/International Electrotechnical Commission. *"MPEG-2 Video"*. Doc. ISSO/IEC IS 13818-2. International Standard, May1996.

[23] ITU-T Recommendation H.262, ISSO/IEC 13182-2, "Information Technology – Generic Coding of Motion Pictures and Associated Audio Information: Video", August. 1995.

[24] Oppeenheim, A. V., and R. W. Schafer, " Discrete-Time Signal Processing " Prenyice-Hall, 1989.

[25] D.F. Elliot and K.R.Rao, "Fast transforms: Algorithms, analyses and applications" New York, NY: Academic Press, 1982.

[26] J. W. Cooley and J.W.Jukey, "An algorithm for the machine calculation of complex Fourier series", Math of Comput., vol. 19, pp. 297-301, 1965.

[27] N.Ahmed, T.Natarajan, and K.R.Rao, "Discrete cosine transform", IEEE Trans. Comput., vol. C-23, pp.90-93, Jan. 1974.

[28] W.H.Chen, C.H.Smith, and S.C.Fralick, "A fast computational algorithm for the discrete cosine transform", IEEE Trans. Commun., vol. COM-25, pp.1004-1009, Sept. 1977.

[29] Max, Joel, "Quantizing for Minimum Distortion ", IRE Trans. Inform. Theory, vol. IT-6, pp. 267-272, Mar. 1960.

[30] H. Kitajima, "A symmetric cosine transform", IEEE Trans. Comput., vol. C-29, pp317-323, Apr. 1980.

[31] Z. Wang, "Fast algorithms for the discrete *W* transform and for the discrete Fourier transform", IEEE Trans. Acoust., Speech, and Signal Process., Vol. ASSP-32, pp803-816, Aug. 1984.

[32] R.M.Haralick, "A storage efficient way to implement the discrete cosine transform", IEEE Trans. Comput., vol. C-25, pp. 764-765, Jun. 1976.

[33] B. D. Tseng and W.C.Miller, "On computing the discrete cosine transform", IEEE Trans. Comput., vol. C-27, pp. 966-968, Oct. 1978.

[34] Schafer, R. W. & Rabiner, L. R., "A Digital Signal Processing Aproach to Interpolation", Procedure of the IEEE, Vol. 61, No.6, pag. 692 - 702, Jun. 1973.

[35] N. Ahmed, T. Natarajan, and K.R.Rao, "Discrete cosine trasnform", IEEE Trasn. Comput., Vol. C-23, pp. 90-93, Jan. 1974.

[36] Y. M. Le Pannerer, "The RACE European projects on coding', II intl. Workshop on packet Video, Torino, Italy, Sep. 8-9, 1988.

[37] F. Bellifemine, H. Ferreyra, R. Picco, L. Sobrero, "Vector quantization for HDTV-DCT coding", II Intl. Workshop on Signal Processing of HDTV, L`Aquila, Italy, Elsevier Science Publishers, North Holland, pp. 239-247, Feb.-Mar., 1988.

[38] K. H. Tzou, "Compatible HDTV coding for broadband ISDN," GLOBECOM 88, Global Commun., Conf., pp. 743 - 749, Hollywood, FL, Nov. 1988.

[39] D. L. Gall, H. Gaggioni, and C. T. Chen, "Transmission of HDTV signals under 140 Mbits/s using a sub-band decomposition and discrete cosine transform coding," II Intl. Workshop on Signal Processing of HDTV, L`Aquila, ltaly, Elsevier Science Publishers, North Holland, pp. 287 - 293, Feb. – Mar. 1988.

[40] G. Barbieri, F. Molo, and J. L. Tejerina, "A modular and flexible video codec architecture for application to TV and HDTV," Intl. TV Symp. and Techl., Exhibition, Broadcast session, pp. 410 - 420, Montreux, Switzerland, Jun. 17 - 22, 1989.

[41] Telettra, "DCT algorithm for broadcast quality encoding of NTSC television for transmission at 44.736 Megabits/second (DS3)," Document number: Tl.Y1.l/88-035, Aug. 2, 1988.

[42] S. Cucchi and F. Molo, "DCT based television codec for DS3 digital transmission," l30th SMPTE Technical Conf., Preprint No. 130-12, New York, NY, Oct. 1988.

[43] M. Muratori and M. Stroppiana, "Bit rate reduction techniques for coding standard and high definition TV signals," Young Researchers Seminar, MIT, Boston, MA, Oct. 1988.

[44] CMTT/2-DCT Group chairman report. "Digital transmission of component-coded television signals at 30 - 34 Mbit/s and 45 Mbit/s using discrete cosine transform," CMTT/2-66, Jul. 1988.

[45] M. Barbero and M. Stroppiana, "Coding of digital TV signal: systems for redundancy reduction based on the discrete cosine transform," Translation from the artical Codifica del segnale televisivo numerica: sistemi di riduzione della ridondnza mediante l'uso della transformata cosseno discrete «Elettronica e Telecomunicazioni». 1, 1989.

[46] CCIR Rec. 601, "Encoding parameters of digital television for studios", CCIR Tec. and Rep., Intl. Telecommun. Union, Vol. XI-part 1, Plenary Assembly, Geneva, Switzerland, 1982.

[47] V. Thomas, "DCT coding for TV contributions applications," PCS 88, Picture Coding Symp., pp. 13.14-1 through 13.14-2, Torino, Italy, Sept. 1988.

[48] G. Madec, "A 15 Mbit/s codec for 4:2:2 TV signals," PCS 88, Picture Coding Symp., pp. 14.7-1, Torino, Italy, Sept. 1988.

[49] N. Doi. H. Hanyu, M. Izumita, S. Mita, Y. Eto, and H. Imai, "Adaptive DCT coding of the home digital VTR," GLOBECOM 88, Global Commun, Conf., pp. 1073—1079, Hollywood, FL, Nov. - Dec. 1988.

[50] E. Peters, "A 15 Mbit/s codec for component video signals,' PCS 88, Picture Coding Symp., pp. 15.7-1 through 15.7-2, Torino, Italy, Sept. 1988.

[51] L. Chiariglione, "Standardization of moving picture coding for interactive applications," GLOBECOM 89. IEEE Global Commun., Conf., pp. 559 - 563, Dallas, TX, Nov. 27-30, 1989.

[52] N. Ahmed and T. Natarajan, "Some aspects of adaptive transform coding of multispectral data," 16th Asilomar Conf. on Circuits, Systems and Computers, pp. 583—587, Pacific Grove, CA, Nov. 22—24, 1976.

[53] T. Gioutsos and S. Werness, "Transform coding of synthetic aperture radar (SAR) images", ICASSP 87, Intl. Conf. on Acousti., Speech, and Signal Porcess., pp 1370 – 1373, Dallas, TX, Apr. 6 – 9, 1987.

[54] D. J. Mulvaney, D.E. Newland, and K. F. Gill, "A comparison of orthogonal transforms in their apllication to surface texture analusis", Proc. Inst. of Mechanical Engineers, vol. 200, Part C, pp. 407 – 414, 1986.

[55] S. D. H. Saunders and C. J. Gillham, "Transmitting IR images at very low data rate", Proc. IEE 2nd Intl. Conf. on Image Processing and Its applications, pp. 196 – 199, London, Uk, Jun. 1986.

[56] C. Sidney, Ramesh A. Gonipath, and Haitao Guo, "Wavelet and Wavelet Transforms", Prentice Hall, 1997.

[57] Charles F. Chui, "An introduction to wavelets", Academic Press Inc., San Diego, 1992.

[58] A. G. Tescher and R. V. Cox, "An adaptive transform coding algorithm", Intl. Conf. on Commun., pp/ 47-20 through 47-23, Philadelphia, PA, 14-16, 1976.

[59] O. E. Bessett and W. B. Schaming, "A two dimensional discrete cosine transform video bandwidth compression system", NAECON, pp. 1146 – 1152, Dayton, OH, May 1980.

[60] J.W. Modestino, D.G.Daut, and A. L.Vickers, "Combined source channel coding of images using the block cosine transform", IEEE Trans. commun., Vol COM-29, pp. 1261-1274, Sept. 1981.

[61] H. C. Reeve and J. 5. Lim, "Reduction of blocking effect in image coding," ICASSP 83, Intl. Conf.**.** on Acoust., Speech, and Signal Process., pp. 1212 - 1215, Boston, MA, Apr. 1983.

[62] A. K. Jain and P. M. Farrelle, "Recursive block coding", l6th Asilomar Conf. on Circuits, Systems and Computers, pp. 431 - 436, Pacific Grove, CA, Nov. 1982.

[63] M. Schlichte, "Block-overlap transform coding of image signals," Siemens Forsch.-u Entwickl.-Ber. Bd. 13. pp. 100 - 104, Springer-Verlag, 1984.

[64] P. M. Farrelle and A. K. Jain, "Recursive block coding - a new approach to transform coding," IEEE Trans. Commun., vol. COM-34, pp. 112 - 117, Feb. 1986.

[65] M. Miyahara and K. Kotani, "Block distortion in orthogonal transform coding - analysis, minimization and distortion measure," IEEE Trans. Commun., vol. COM-33, pp. 90 - 96. Jan. 1985.

[66] L. T. Watson, R. M. Haralick and O. A. Zuniga, "Constrained transform coding and surface fitting," IEEE Trans. Commun., vol. COM-3l, pp. 7 17 - 726, May 1983.

[67] N. M. Nasrabadi and R. A. King, "Computationally eflicient adaptive block-transform coding," Signal Process. II, Theories and Applications, EURASIP, pp. 729 – 733, Erlangen, W. Germany, Sept. 12 - 16, 1983.

[68] H. M. Hang and B. G. Haskell. "Interpolative vector quantization of color images," IEEE Trans. Commun., vol. COM-36, pp. 465 - 470, April 1988.

[69] K. H. Tzou, "Post-filtering of transform coded images," SPIE, Advances in Image Processing, vol. 974, pp. 121 - 126, San Diego, CA, Aug. 1988.

[70] K. Nemoto and T. Omachi, "Average separation method for DCT coding using smooth interpolation", PCS 88, Picture Coding Symp., pp. 8.12-1 through 8.12-2, Torino, Italy, Sept. 1988.

[71] P. M. Cassereau, D. H. Staelin and G. D. Jager, "Encoding of images based on lapped orthogonal transform," IEEE Trans. Commun., vol. COM-37, pp. 189 - 193, Feb. 1989.

[72] S. Venkataraman, V. R. Kanchan, K.R. Rao and M. Mohanty, "Discrete transforms via the Walsh-Hadamard transform", Signal Process., vol. 14, pp. 371 – 382, 1988.

[73] H. S. Malvar and R. Duarte, "Transform/subband coding of speech with the lapped ortogonal transform", ISCAS 89, Intl. Symp. on Circuits and Systems, pp. 1268 – 1271, Portland, OR, May, 1989.

[74] H. S. Malvar, "The LOT: a link between block transform coding and multirate filter bankx", Intl. Symp. on Circuits and Systems, pp. 835 – 838, Espoo. Finland, Jun. 7 – 9, 1988.

[75] P. Yip and K. R. Rao, "Fast DIT algorithms for DST`s and DCT`s", Circuits, Systems and Signal Process., vol. 3, No. 4, pp. 387 – 408, 1984.

[76] K. N. Ngan, K. S. Leong, and H. Singh, "A HVS-weighted transform coding scheme with adaptive quantization", SPIE Visual commun. and Image Process., vol. 1001, pp. 702 – 708, Cambridge, MA, Nov. 9 – 11, 1988.

[77] K. S. Leong, "Adaptive cosine transform image coding incorporating human visual system model", M. S. Thesis, National Univ. of Singapore, Dec. 1987.

[78] K. N. Ngan, K. S. Leong, and H. Singh, "Adaptive cosine transform coding of images in perceptual domain", IEEE Trans., Acoust., Speech and Signal Process., vol. ASSP-37, pp. 1743 – 1750, Nov. 1989.

[79] Description of Ref. Model 5 (RMS), CCITT SGXV, Working Party XV/4, specialists group on coding for visual telephony, Document 375. Sept. 1988.

[80] J. Maeng and D. Hein, "A low-rate video coding based on DCT/VQ," SPIE, Visual Commun., and Image Processing, vol. 1199, pp. 267 - 273, Philadelphia, PA, Nov. 5 - 10, 1989.

[81] Report of the thirteenth meeting in Paris CCITT SGXV, Working Party XV/1, Specialists Group on Coding for Visual Telephony, Document no. 395R, Sept. 22, 1988.

[82] Description on Ref. Model (RM6), CCITT SGXV, Working Party XV/1, Specialists group on coding for visual telephony, Document 396, Oct. 20. 1988.

[83] J. Guichard, "Motion video coding in CCITT SGXV—hardware trials," GLOBECOM 88, Global Commun., Conf., pp. 37 - 42, Hollywood, FL, Nov. - Dec. 1988.

[84] NTT, KDD, NEC, and FUJITSU, "Accuracy of DCT calculation," Document no. 255, CCITT SGXV, Working Party XV/1, Specialists group on coding for visual telephony, Oct. 1987.

[85] Report of the fifteenth meeting in Oslo CCITT SGXV, Working Party XV/l, Specialists group on coding for visual telephony, Document no. 499R, Mar. 10, 1989.

[86] S. Okubo, "Video codec standardization in CCITT study group XV," Signal Proccss: Image Commun., vol. 1, pp. 45 - 54, Jun. 1989.

[87] Report of the fifteenth meeting in Stuttgart, CCITT SGXV, Working Party XV/1, Specialists group on coding for visual telephony, Document no. 540R, Jun. 16, 1989.

[88] J. P. Hudson, "Report of the joint photographic experts group meeting," 25 - 27, Jan. 1988, KTAS, Copenhagen, Denmark, ISO/IEC/JTC1/SC2/WG8 N175, Rev. 2, May 1988.

[89] G. P. Hudson, "PICA-Photovideotex image compression algorithms, ESPRIT 86: Results and Achievements," North Holland, Elsevier Science Publishers, 1987.

[90] G. P. Hudson, "Esprit 563-PICA, Photovideotex image compression algorithms-toward international standardisation," ISO/TC97/SC2/WG8 N564, Brussels, Belgium, Sept. 28 - 30, 1987.

[91] T. P. Hudson and H. Yasuda, "The selection of a still picture compression technique for international standardisation," PCS 88, Picture Coding Symp., pp. 9.1-1 through 9.1-2, Torino, Italy, Sept. 1988.

[92] G. Wallace. R. Vivian, and H. Poulsen, "Subjective assessment of JPEG compression techniques," PCS 88, Picture Coding Symp., pp. 9.2-1 through 9.2-2, Torino, Italy, Sept. 1988.

[93] A. Leger, B. Niss, J. Vaaben, L. Chiarighione, H. Lohscheller, and S. Gicquel, "Adaptive discrete cosine transform coding scheme (ADCT) for still picture compression standard," PCS-88, Picture Coding Symp., pp. 9.3-1 through 9.3-2, Torino, ltaly, Sept. 1988.

[94] B. Macq and P. Delogne, "Progressive transmission of pictures by transform coding," PCS 88, Picture Coding Symp., pp. 6.4-1 through 6.4-2, Torino, Italy, Sept. 1988.

[95] G. P. Hudson, H. Yasuda, and I. Sebestyen, "The international standardisation of a still picture compression technique," GLOBECOM 88, Global Commun., Conf. pp. 1016 - 1021, Hollywood, FL. Nov. - Dec. 1988.

[96] Leger, J. L. Mitchell, and Y. Yamazaki, "Still picture compression algorithms evaluated for international standardisation," GLOBECOM 88, Global Commun., Conf., pp. 1028 - 1032, Hollywood, FL, Nov. - Dec. 1988.

[97] G. K. Wallace, R. Vivian, and H. Poulsen, "Subjective testing results for still picture compression algorithms for international standardisation," GLOBECOM 88, Global Commun., Conf., pp. 1022 - 1027, Hollywood, FL, Nov. - Dec. l988.

[98] I. Sebestyen, C. F. Touchton, and H. Yasuda, "Application and service requirements for communication of still images," PCS 88, Picture Coding Symp., pp. 9.6-1 through 9.6-2, Torino, ltaly, Sept. 1988.

[99] J. L. Mitchell, W. B. Pennebaker, and C. A. Gonzales, "The standardization of color photographic image data compression," SPIE, Digital image processing applications, vol. 1075, pp. 101 - 106, Los Angeles, CA, Jan. 1989.

[100] H. Yasuda, "Standardization activities on multimedia coding in ISO," Signal Process: Image Commun., vol. 1, pp. 3 - 16, Jun. 1989.

[101] L. Contin, L. Corgnier, and L. Masera, "Interframe extension of ISO still picture coding algorithm and applications to packet network", II Intl. Workshop on Packet Video, Torino, Italy, Sept. 8 - 9, 1988

[102] N. Ahmed and K. R. Rao, "Orthogonal transforms for digital signal processing", New York, NY: Springer, 1975.

[103] K. R. Rao, "Discrete transforms and their applications", New York, NY: Van Nostrand Reinhold, 1985.

[104] G. Kummerfeldt, F. May, and W. wolf, "Coding television signals at 320 and 64 Kbit/s", 2nd International Technical Symposium on Optical and Electro-Optical Applied Science and Engineering, SPIE, vol. 594, pp. 119 – 128, Cannes, France, Dec. 1985.

[105] J. Ameye, J. Bursens, S. Desmet, K. Vanhoof, G. Tu, J. Rommelaere, and A. Oostevlinck, "Image coding using the human visual system", Intl. Workshop on image coding, The Korea Inst. of Commun. Sciences, pp. 229 – 308, Seoul, Korea, Aug. 1987.

[106] W.K. Pratt and W.H.Chen, "Slant transform image coding", IEEE Trans. Commun., vol COM-22, pp. 1075-1093, Aug. 1974.

[107] K. N. Ngan, "Image display techniques using the cosine transform", IEEE Trans. Acoust., Speech, and Signal Process., vol. ASSP-32, pp. 173 – 177, Feb. 1984.

[108] N. S. Jayant, "Waveform Quantization and Coding", IEEE Press, New York.

[109] N. S. Jayant and Peter Noll, "Digital Coding of Waveforms – Principles andApplications to Speech and Video", Prentice Hall, INC.

[110] J. Max, "Quantization for Minimun Distortion", IEEE Trans. on Information Theory, Vol. IT-6, March 1960, pp. 7 – 12. Also, S. P. Lloyd, "Least Squares Quantization in PCM", IEEE Trans. on Information Theory, pp. 129 – 136, Mar. 1982.

[111] Gersho, A. and Gray R. M., "Vector Quantization and Signal Compression", Kluwer Academic Publishers, 1991.

[112] H. Lohscheller, "A subjectively adapted image communication system", IEEE Trans. on Commun., Com-32(12):1316 – 1322, Dec. 1984.

[113] Y. Lind, A. Buzo, and R. M. Gray, "An algorithm for vector quantizer design," IEEE Trans. commum., vol. COM-28, pp. 84-95, Jan. 1980.

[114] N. M. Nasrabadi and R. A. King, "Image coding using vector quantization: A review," IEEE Trans. Commun., vol. COM-36, pp. 957 - 971, Aug. 1988.

[115] H. C. Tseng and T. R. Fischer, "Transform and hybrid transform/DPCM coding of images using pyramid vector quantization," IEEE Trans. Commun., vol. COM-35, pp. 79 - 86, Jan. 1987.

[116] M. Goldberg, P. R. Boucher and S. Shlien, "Image compression using adaptive vector quantization," IEEE Trans. Commun., vol. COM-34, pp. 180 - 187, Feb. 1986.

[117] B. Ramamurthi and A. Gersho, "Classified vector quantization of images," IEEE Trans. Commun., vol. COM-34, pp. 1105 - 1115, Nov. 1986.

[118] C. Labit and J. P. Marescq, "Temporal adaptive vector quantization for image sequence coding," SPIE, Advances in Image Process., vol. 804, pp. 371 - 378, The Hague, Netherlands, Mar. 31-Apr. 3, 1987.

[119] T. Moriya and H. Suda, "An 8 Kbit/s transform coder for noisy channels," Intl. Conf. on Acoust., Speech, and Signal Process., pp. 196 – 199, Glasgow, Scotland, May 23 – 16, 1989.

[120] S. Venkataraman and K. R. Rao, "Applications of vector quantizers and BTC in image coding,"GLOBECOM'85, IEEE Global commun. Conf., pp. 602 – 608, New Orleans, LA, Dec. 2-5, 1985.

[121] C. H. Yim and J. K. Kim, "A simple DCT-CVQ based on two DCT coefficients," PCS 88, Picture Coding Symp., pp. 8.11-1 through 8.11-2, Torino, Italy, Sept. 1988.

[122] K. Aizawa, H. Harashima and H. Miyakawa, "Vector quantization of picture signals in DCT domain", IECE Japan, Tech. Rep., IE84-86, Dec. 1984.

[123] K. Aizawa, H. Harashima and H. Miyakawa, "Vector quantization of picture using discrete cosine transform (DCT-VQ)", J. Inst. TV Engrs of Japan, vol. 39, 10, pp. 920 – 925, Oct. 1985.

[124] K. Aizawa, H. Harashima, and H. Miyakawa, "Adaptive discrete cosine transform coding with vector quantization for color images," ICASSP 86, Intl. Conf. on Acoust. Speech Signal Process., pp. 985 - 988, Tokyo, Japan, Apr. 7 - 11, 1986.

[125] W. H. Chen and C. H. Smith, "Adaptive coding of color images using cosine transform," ICC 76, Intl. Conf. on Commun., pp. 47-7 through 47-13, Philadelphia, PA, Jun., 1976.

[126] W. H. Chen and C. H. Smith, "Adaptive coding of monochrome and color images," IEEE Trans. Commun., vol COM-25, pp. 1285 – 1292, Nov. 1977.

[127] T. Omachi, Y. Takashima, and H. Okada, "DCT-VQ coding scheme using categorization with adaptive band partition," PCS 87, Picture Coding Symp., pp. 161 - 162, Stockholm, Sweden, Jun. 9 - 11, 1987.

[128] J. I. Gimlett, "Use of 'activity'classes in adaptive transform image coding," IEEE Trans. Commun., vol COM-23, pp. 785 – 786, Jul. 1975.

[129] M. Gilge, "Adaptive transform coding of four-color printed images," PCS 86, Picture Coding Symp., pp. 72-73, Tokyo, Japan, Apr. 2-4, 1986.

[130] M. Gilge, "Adaptive transform coding of four-color printed images," ICASSP 87, Intl. Conf. on Acoust., Speech, and Signal Process., pp. 1374-1377, Dallas, TX, Apr. 6-9, 1987.

[131] J. K. Wu and R. E. Burge, "Adaptive bit allocation for image compression,"Comput. Graphics and Image Process, vol. 19, pp. 392 – 400, 1982.

[132] Y. Kato, N. Mukawa, and S. Okubo, "A motion picture coding algoritm using adaptive DCT encoding based on coefficient power distribution classification," IEEE Journal on selected Areas in commun., vol. Sac-5, pp. 1090 – 1099, Aug. 1987.

[133] O. Franceschi, Y. Shtarkov and R. Forchheimer, "An adaptive source coding method for still images,"PCS 88, Picture coding symp., pp. 6.5-1 through 6.5-2, Torino, Italy, Sept. 1988.

[134] Y. S. Ho and A. Gersho, "Classified transform coding of images using vector quantization," Intl. Conf. on Acoust., Specch, and Signal Process., pp. 1890 - 1893, Glasgow, Scotland, May 23 - 26, 1989.

[135] O. Chantelou and C. Remus, "Adaptive transform coding of HDTV pictures," II Intl. Workshop on Signal Processing of HDTV, L'Aquila, Italy, Elsevier Science Publishers, North Holland, pp. 231 – 238, Feb. – Mar. 1988.

[136] H. Holzlwimmer, A. v. Brandt, and W. Tengler, "A 64 kbit/s motion compensated transform coder using vector quantization with scene adaptive codebook,"ICC 87, Intl. Conf. on Commun., pp. 151 – 156, Seatle, WA, Jun. 1987.

[137] A. Gersho and B. Ramamurthi, "*Image coding using vector quantization*", IEEE Int. Conf. Acoust., Speech, Signal Processing, pp. 428-431, May 1982.

[138] B. Ramamurthi and A. Gersho, "*Image Vector Quantization with a perceptually-based cell classifier*", IEEE Int. Conf. Acoust., Speech, Signal Processing, pp. 32.10.1-32.10.4, Mar. 1984.

[139] J. O. Limb, "*Distortion criteria of the human viewer*", IEEE Trans. Syst., Man, Cybern., vol. SMC-9, pp. 778-793, Dec. 1979

[140] Y. Linde, A. Buzo, and R.M. Gray, "*An algorithm for vector quantizer desing*", IEEE Trans. on Commun., vol. COM-28, pp. 84-95, Jan. 1980.

[141] H. Takeo, T. Saito, H. Harashima and Miyakawa, "*A study of mismatch in vector quantization of picture signals*", IECE Japan, Tech. Rep., IE83-103, Dec. 1983.

[142] Malvar, H. S. and D. H. Staelin "*The LOT: Transform Coding Without Blocking Effect*", IEEE Trans. on Acoustics, Speech, and Signal Processing, vol. 37, Apr. 1989.

[143] Malvar, H. S. and D. H. Staelin "*Reduction of Blocking Effect in Image Coding with Lapped Orthogonal Transform*", IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing, New York, pp. 781-748, Apr. 1988.

[144] S. C. Chan and K. L. Ho., "*A New Two-Dimensional Fast Cosine Transform."* IEEE Trans. on Signal Processing, vol. 39, pp. 481-485, Feb. 1991.

[145] R. M. Haralick, "*A storage efficient way to implement the discrete cosine transform,*" IEEE Trans. comput., vol. c-25,pp. 764-765, Jul. 1976.

[146] W. H. Chen, C. H. Smith, and S. C. Fralick, "*A fast computational algorithm for the discrete cosine transform,*" IEEE Trans. Commun., vol. COM-25, pp. 1004-1009, Sept. 1977.

[147] M. J. Narasimha and A. M. Peterson, "*On the computational of discrete cosine transform,*" IEEE Trans. Commun., vol. COM-26, pp. 934-936, Jun. 1978.

[148] B. D. Tseng and W. C. Miller, "*On computing the discrete cosine transform,*" IEEE Trans. Comput., vol. C-27, pp. 966-968, Oct. 1978.

[149] J. Makhoul, "*A fast cosine transform in one and two dimensions,*" IEEE Trans. Acoust., Speech, and Signal Processing, vol. ASSP-28, pp. 27-34, Feb. 1980.

[150] L. G. Byeong, "*A New Algorithm to Compute the Discrete Cosine Transform,*" IEEE Trans. Acoust., Speech, and Signal Processing, vol. ASSP-32, NO. 6, pp. 1243-1245, Dec. 1984.

[151] H. D. Ferreyra, R. Picco, and L. A. Sobrero, "*Analysis and project considerations on 2D-DCT adaptive coding,"* Digital Signal Processing–87, V. Cappelini and A. G. Constantinides (eds.), Elsevier Science Publishers, North Holland, 1987.

[152] L. D. Davisson, "*Rate-Distortion Theory and Application,"* Proceedings of IEEE, Vol. 60, pp. 800-828, Jul. 1972.

[153] J. Pearl, H. C. Andrews, and W. K. Pratt, "*Performance Measure for Transform Data Coding,*" IEEE Trans. on Commun., Vol. COM-20, No. 3, pp. 411-515, Jun. 1972.

[154] S. P. Lloyd, "*Least Square Quantization in PCM,"* IEEE Trans. on Information Theory, vol. IT-28, no. 2, pp. 129-137, Mar. 1982.

[155] J. Max, *"Quantization for Minimum Distortion,"* IRE Trans. on Information Theory, vol. IT- 6, no. 1, pp. 7-12, Mar 1960.

[156] N. Ahmed, T. Natarjan, and R. Rao, *"Discrete Cosine Transform,"* IEEE Trans. on Comp., pp. 90-93, Jan. 1974.

[157] N. Ahmed, T. Natarajan, andK.R.Rao, "Discrete cosine transform", IEEE Trans. Comput., vol. C-23, pp.90-93, Jan. 1974.

[158] M. J. Narashima and A.M.Peterson, "On the computation of the discrete cosine transform", IEEE Trans, Commun., vol. COM-26, pp. 934-946, Jun. 1978,

[159] P.Duhamed, "Implementation of 'Split-Radix' FFT algorithms for complex, real and real- symmetric data", IEEE Trans. Acoust., Speech, and Signal Process., vol. ASSP-34, pp. 285-295, Apr. 1986.

[160] M. Vetterli and H.Nussbaumer, "Simple FFT and DCT algorithms with reduced number of operations", Signal Processing, vol. 6, pp. 267-278, Aug. 1984.

[161] M.S.Corrington, "Implementation of fast cosine transforms using real arithmetic", NAECON, Dayton, OH, May 16-18, 1978.

[162] Z.Wang, "Reconsideration of a fast computational algorithm for the discrete cosine transform", IEEE Trans. Commun., vol. COM-31, pp. 121-123, Jan. 1983.

[163] B.G.Lee, "A new algorithm for the discrete cosine transform", IEEE Trans. Acoust., Spreech, and Signal Process., vol. ASSP-32, pp. 1243-1245, Dec. 1984.

[164] B.G.Lee, "FCT – a fast cosine transform", Intl. Conf. On Acoust., Speech,and Signal Process., pp.28A3.1-28A3.3, San Diego, CA, Mar. 1984.

[165] N. Suehiro and M.Hatori, "Fast algorithms for the DFT and other sinusoidal transforms", IEEE Trans. Acoust., Speech, and Signal Process., vol. ASSP-34, pp. 642-644, Jun. 1986.

[166] P. Yip and K.R.Rao, "Fast DIT algorithms for DST's andDCT's", Circuits, Systems and Signal Process., vol. 3, No. 4, pp. 387-408,1984.

[167] P. Yip and K.R.Rao, "Fast DIF algorithms for DCT's andDST's", Intl., Conf. on Acoust., Speech, and Signal Process., pp. 776-779, Tampa, FL, Mar. 26-29, 1985.

[168] P. Yip and K.R.Rao, "The decimation-in-frequency algorithms for a family of discrete sine and cosine transforms", Circuits, Systems, and Signal Process., pp. 4-19, 1988.

[169] D. Hein and N.Ahmed, "On a real-time Walsh-Hadamard cosine transform image processor", IEEE Trans. Electromag. Compat., vol. EMC-20, pp. 453-457, Aug. 1978.

[170] H.W.Jones, D.N.Hein, and S.C.Knauer, "The Karhunen-Loeve, discrete cosine and related transforms via the Hadamard transform", Intl. Telemetering Conf., pp. 87-98, Los Angeles, CA, Nov. 14-16, 1978.

[171] S. Venkataraman et al., "Discrete transform via the Walsh-Hadamard transform", 25th Midwest Symp. Circuits andSystems, Puebla, Mexico, Proc. Pp. 74-78, Aug. 1983.

[172] H.S.Malvar, "Fast computation of the discrete cosine transform and the discrete Hartley transform", IEEE Trans. Acoust., Speech, and Signal Process., vol. ASSP-35, pp. 1484–1485, Oct. 1987.

[173] M.S.Malvar, "Fast computation of the discrete cosine transform through the fast Hartley transform", Electronics Letters, vol. 22, pp. 352-353, Mar. 27, 1986.

[174] H.S.Malvar, Corrections to, "Fast computation of discrete cosine transform and the discrete Hartley transform", IEEE Trans. Acoust., Speech, and Signal Process., vol. ASSP-36, pp.610, Apr. 1988.

[175] V. Nagesha, "Comments on fast computation of the discrete cosine transform and discrete Hartley transform", IEEE Trans. Acoust., Speech, and Signal Process., col ASSP-37, pp. 439-440, Mar. 1989.

[176] P. P. N. Yang and M.J.Narasimha, "Prime factor decomposition of the discrete cosine transform", Intl. Conf. on Acoust., Speech, and Signal Process., pp. 772-775, Tampa, FL, Mar. 26-29, 1985.

[177] H.S.Hou, "A fast recursive algorithm for computing the discrete cosine transform", IEEE Trans. Acoust., Speech, and Signal Process., vol. ASSP-35, pp. 1455-1461, Oct. 1987.

[178] A. Ligtenberg and J.H.O'Neill, "A single chip solution for na 8 by 8 two dimensional DCT", ISCAS 87, Intl. Symp. on Circuits and Systems, pp. 1128-1131, Philadelphia, PA, May, 5-7, 1987.

[179] A. Ligtenberg , R.H.Wright, and J.H.O'Neill, "A VLSI orthogonal transform chip for real-time image compression", Visual Commun. And Image Process. II, SPIE, Cambridge, MA, Oct. 1987.

[180] C.Loeffler, A.Ligtenberg, and G.S.Moschytz, "Algorithm-architecture mapping for custom DSP chips", Intl., Symp. on Circuits and Systems, pp. 1953-1956, Espoo, Finland, Jun. 1988.

[181] C. Loeffler, A.Ligtenberg, and G.S.Moschytz, "Practical fast 1-D DCT algoriths with 11 multiplications", Intl. Conf. on Acoust., Speech, and Signal Process. Pp. 988-991, Glasgow, Scotland, May 23-26, 1989.

[182] C. Sun and P. Yip, "Split radix algorithms for DCT and DST", Asilomar Conf. on Signals, Systems, and Computers, pp. 508-512, Pacific Grove, CA, Nov. 1989.

APÊNDICE A

Quantizadores Ótimos

Este apêndice refere-se aos quantizadores ótimos de Lloyd-Max com coeficientes de decisão ***d[n]*** e de reconstrução ***r[n]*** para a realização da quantização dos coeficientes de uma transformada para codificação de imagens. Os coeficientes deverão estar no formato *float* quando da realização da operação de quantização. Mostra-se primeiramente os quantizadores para a função densidade de probabilidade gaussiana, para diversos níveis de quantização com ***n bits*** (***n*** variando de 2 a 8 bits) obtendo-se o quantizador de Lloyd-Max com ***N*** níveis que podem variar de 4 a 256 níveis de quantização e posteriormente os quantizadores para a função densidade de probabilidade laplaciana para os mesmos níveis.

Quantizadores Lloyd-Max para a função gaussiana

```
gaus_d_1bit[2] = {-20.000000, 0.000000 }
gaus_r_1bit[2] = { -0.797885, 0.797885 }


gaus_d_2bit[4]={-20.000000, -0.981830, -0.000000, 0.981830 }
gaus_r_2bit[4]={ -1.510729, -0.452931, 0.452931, 1.510729 }


gaus_d_3bit[8] = {
            -20.000000, -1.749115, -1.050993, -0.501148,
           0.000000,  0.501148,  1.050993,  1.749115 }
gaus_r_3bit[8] = {
             -2.153125, -1.345105, -0.756881, -0.245415,
              0.245415,  0.756881,  1.345105,  2.153125 }


gaus_d_4bit[16] = {
      -20.000000, -2.403336, -1.846245, -1.439799, -1.101693,
       -0.801527, -0.523808, -0.258950, -0.000000,  0.258950,
        0.523808,  0.801527,  1.101693,  1.439799,  1.846245,
        2.403336 }
gaus_r_4bit[16] = {
```

```
       -2.734968, -2.071703, -1.620786, -1.258813, -0.944573,
       -0.658481, -0.389134, -0.128766,  0.128766,  0.389134,
        0.658481,  0.944573,  1.258813,  1.620786,  2.071703,
        2.734968 }


gaus_d_5bit[32] = {
      -20.000000, -2.980666, -2.509647, -2.178737, -1.913631,
       -1.687404, -1.486866, -1.304453, -1.135370, -0.976349,
       -0.825036, -0.679651, -0.538796, -0.401324, -0.266263,
       -0.132752, -0.000000,  0.132752,  0.266263,  0.401324,
        0.538796,  0.679651,  0.825036,  0.976349,  1.135370,
        1.304453,  1.486866,  1.687404,  1.913631,  2.178737,
        2.509647,  2.980666 }
gaus_r_5bit[32] = {
      -3.265176, -2.696156, -2.323138, -2.034337, -1.792924,
      -1.581883, -1.391848, -1.217058, -1.053682, -0.899016,
      -0.751055, -0.608247, -0.469344, -0.333304, -0.199222,
      -0.066282,  0.066282,  0.199222,  0.333304,  0.469344,
       0.608247,  0.751055,  0.899016,  1.053682,  1.217058,
       1.391848,  1.581883,  1.792924,  2.034337,  2.323138,
       2.696156,  3.265176 }


gaus_d_6bit[64] = {
     -20.000000, -3.637592, -3.237185, -2.962279, -2.746196,
      -2.564869, -2.406559, -2.264580, -2.134731, -2.014193,
      -1.900975, -1.793618, -1.691021, -1.592332, -1.496880,
      -1.404130, -1.313645, -1.225069, -1.138106, -1.052509,
      -0.968072, -0.884618, -0.801997, -0.720082, -0.638762,
      -0.557939, -0.477529, -0.397457, -0.317656, -0.238064,
      -0.158626, -0.079288,  0.000000,  0.079288,  0.158626,
       0.238064,  0.317656,  0.397457,  0.477529,  0.557939,
       0.638762,  0.720082,  0.801997,  0.884618,  0.968072,
       1.052509,  1.138106,  1.225069,  1.313645,  1.404130,
       1.496880,  1.592332,  1.691021,  1.793618,  1.900975,
       2.014193,  2.134731,  2.264580,  2.406559,  2.564869,
       2.746196,  2.962279,  3.237185,  3.637592 }
gaus_r_6bit[64] = {
      -3.881718, -3.393465, -3.080905, -2.843652, -2.648740,
      -2.480998, -2.332120, -2.197039, -2.072424, -1.955963,
      -1.845988, -1.741249, -1.640793, -1.543871, -1.449890,
      -1.358370, -1.268920, -1.181217, -1.094994, -1.010025,
      -0.926119, -0.843117, -0.760878, -0.679286, -0.598237,
      -0.517640, -0.437417, -0.357497, -0.277815, -0.198314,
      -0.118939, -0.039638,  0.039638,  0.118939,  0.198314,
       0.277815,  0.357497,  0.437417,  0.517640,  0.598237,
       0.679286,  0.760878,  0.843117,  0.926119,  1.010025,
       1.094994,  1.181217,  1.268920,  1.358370,  1.449890,
```

```
        1.543871,  1.640793,  1.741249,  1.845988,  1.955963,
        2.072424,  2.197039,  2.332120,  2.480998,  2.648740,
        2.843652,  3.080905,  3.393465,  3.881718 }


gaus_d_7bit[128] = {
      -20.000000, -5.356783, -5.068614, -4.873842, -4.721321,
       -4.592769, -4.479365, -4.376160, -4.280103, -4.189194,
       -4.102065, -4.017749, -3.935548, -3.854950, -3.775573,
       -3.697131, -3.619406, -3.542234, -3.465489, -3.389074,
       -3.312915, -3.236954, -3.161148, -3.085463, -3.009872,
       -2.934354, -2.858895, -2.783482, -2.708105, -2.632758,
       -2.557433, -2.482128, -2.406838, -2.331560, -2.256293,
       -2.181035, -2.105785, -2.030540, -1.955302, -1.880068,
       -1.804839, -1.729613, -1.654392, -1.579173, -1.503957,
       -1.428744, -1.353534, -1.278325, -1.203119, -1.127915,
       -1.052713, -0.977512, -0.902313, -0.827115, -0.751919,
       -0.676724, -0.601530, -0.526336, -0.451144, -0.375952,
       -0.300761, -0.225570, -0.150380, -0.075190, -0.000000,
        0.075190,  0.150380,  0.225570,  0.300761,  0.375952,
        0.451144,  0.526336,  0.601530,  0.676724,  0.751919,
        0.827115,  0.902313,  0.977512,  1.052713,  1.127915,
        1.203119,  1.278325,  1.353534,  1.428744,  1.503957,
        1.579173,  1.654392,  1.729613,  1.804839,  1.880068,
        1.955302,  2.030540,  2.105785,  2.181035,  2.256293,
        2.331560,  2.406838,  2.482128,  2.557433,  2.632758,
        2.708105,  2.783482,  2.858895,  2.934354,  3.009872,
        3.085463,  3.161148,  3.236954,  3.312915,  3.389074,
        3.465489,  3.542234,  3.619406,  3.697131,  3.775573,
        3.854950,  3.935548,  4.017749,  4.102065,  4.189194,
        4.280103,  4.376160,  4.479365,  4.592769,  4.721321,
        4.873842,  5.068614,  5.356783 }
gaus_r_7bit[128] = {
      -5.533863, -5.179703, -4.957525, -4.790159, -4.652484,
      -4.533053, -4.425677, -4.326643, -4.233563, -4.144825,
      -4.059304, -3.976193, -3.894903, -3.814997, -3.736149,
      -3.658113, -3.580700, -3.503769, -3.427209, -3.350938,
      -3.274891, -3.199017, -3.123279, -3.047646, -2.972097,
      -2.896612, -2.821179, -2.745786, -2.670425, -2.595090,
      -2.519776, -2.444479, -2.369196, -2.293925, -2.218662,
      -2.143408, -2.068161, -1.992920, -1.917684, -1.842453,
      -1.767225, -1.692002, -1.616781, -1.541564, -1.466350,
      -1.391138, -1.315929, -1.240722, -1.165517, -1.090313,
      -1.015112, -0.939912, -0.864714, -0.789517, -0.714321,
      -0.639126, -0.563933, -0.488740, -0.413548, -0.338357,
      -0.263166, -0.187975, -0.112785, -0.037595,  0.037595,
       0.112785,  0.187975,  0.263166,  0.338357,  0.413548,
       0.488740,  0.563933,  0.639126,  0.714321,  0.789517,
       0.864714,  0.939912,  1.015112,  1.090313,  1.165517,
```

```
      1.240722,  1.315929,  1.391138,  1.466350,  1.541564,
      1.616781,  1.692002,  1.767225,  1.842453,  1.917684,
      1.992920,  2.068161,  2.143408,  2.218662,  2.293925,
      2.369196,  2.444479,  2.519776,  2.595090,  2.670425,
      2.745786,  2.821179,  2.896612,  2.972097,  3.047646,
      3.123279,  3.199017,  3.274891,  3.350938,  3.427209,
      3.503769,  3.580700,  3.658113,  3.736149,  3.814997,
      3.894903,  3.976193,  4.059304,  4.144825,  4.233563,
      4.326643,  4.425677,  4.533053,  4.652484,  4.790159,
      4.957525,  5.179703,  5.533863 }


gaus_d_8bit[256] = {
     -20.000000, -9.084786, -8.903703, -8.778801, -8.677621,
      -8.588835, -8.507171, -8.429815, -8.355146, -8.282181,
      -8.210313, -8.139155, -8.068460, -7.998070, -7.927881,
      -7.857825, -7.787859, -7.717955, -7.648093, -7.578261,
      -7.508452, -7.438659, -7.368880, -7.299113, -7.229355,
      -7.159606, -7.089864, -7.020131, -6.950404, -6.880684,
      -6.810972, -6.741265, -6.671566, -6.601872, -6.532185,
      -6.462504, -6.392829, -6.323161, -6.253498, -6.183841,
      -6.114190, -6.044545, -5.974906, -5.905272, -5.835644,
      -5.766021, -5.696404, -5.626792, -5.557186, -5.487584,
      -5.417988, -5.348397, -5.278811, -5.209230, -5.139654,
      -5.070083, -5.000517, -4.930955, -4.861398, -4.791845,
      -4.722297, -4.652753, -4.583214, -4.513679, -4.444149,
      -4.374622, -4.305100, -4.235581, -4.166067, -4.096557,
      -4.027050, -3.957547, -3.888048, -3.818552, -3.749061,
      -3.679572, -3.610087, -3.540606, -3.471127, -3.401652,
      -3.332181, -3.262712, -3.193246, -3.123784, -3.054324,
      -2.984867, -2.915413, -2.845962, -2.776513, -2.707067,
      -2.637623, -2.568182, -2.498743, -2.429307, -2.359873,
      -2.290441, -2.221011, -2.151584, -2.082158, -2.012734,
      -1.943312, -1.873892, -1.804474, -1.735057, -1.665642,
      -1.596229, -1.526817, -1.457407, -1.387997, -1.318590,
      -1.249183, -1.179777, -1.110373, -1.040970, -0.971567,
      -0.902166, -0.832765, -0.763365, -0.693966, -0.624568,
      -0.555170, -0.485773, -0.416376, -0.346979, -0.277583,
      -0.208187, -0.138791, -0.069396,  0.000000,  0.069396,
       0.138791,  0.208187,  0.277583,  0.346979,  0.416376,
       0.485773,  0.555170,  0.624568,  0.693966,  0.763365,
       0.832765,  0.902166,  0.971567,  1.040970,  1.110373,
       1.179777,  1.249183,  1.318590,  1.387997,  1.457407,
       1.526817,  1.596229,  1.665642,  1.735057,  1.804474,
       1.873892,  1.943312,  2.012734,  2.082158,  2.151584,
       2.221011,  2.290441,  2.359873,  2.429307,  2.498743,
       2.568182,  2.637623,  2.707067,  2.776513,  2.845962,
       2.915413,  2.984867,  3.054324,  3.123784,  3.193246,
       3.262712,  3.332181,  3.401652,  3.471127,  3.540606,
```

```
        3.610087,  3.679572,  3.749061,  3.818552,  3.888048,
        3.957547,  4.027050,  4.096557,  4.166067,  4.235581,
        4.305100,  4.374622,  4.444149,  4.513679,  4.583214,
        4.652753,  4.722297,  4.791845,  4.861398,  4.930955,
        5.000517,  5.070083,  5.139654,  5.209230,  5.278811,
        5.348397,  5.417988,  5.487584,  5.557186,  5.626792,
        5.696404,  5.766021,  5.835644,  5.905272,  5.974906,
        6.044545,  6.114190,  6.183841,  6.253498,  6.323161,
        6.392829,  6.462504,  6.532185,  6.601872,  6.671566,
        6.741265,  6.810972,  6.880684,  6.950404,  7.020131,
        7.089864,  7.159606,  7.229355,  7.299113,  7.368880,
        7.438659,  7.508452,  7.578261,  7.648093,  7.717955,
        7.787859,  7.857825,  7.927881,  7.998070,  8.068460,
        8.139155,  8.210313,  8.282181,  8.355146,  8.429815,
        8.507171,  8.588835,  8.677621,  8.778801,  8.903703,
        9.084786 }


gaus_r_8bit[256] = {
     -9.195524, -8.974048, -8.833358, -8.724244, -8.630998,
     -8.546672, -8.467670, -8.391961, -8.318330, -8.246032,
     -8.174593, -8.103716, -8.033205, -7.962935, -7.892826,
     -7.822824, -7.752894, -7.683015, -7.613171, -7.543352,
     -7.473552, -7.403767, -7.333994, -7.264232, -7.194478,
     -7.124733, -7.054996, -6.985266, -6.915543, -6.845826,
     -6.776117, -6.706414, -6.636717, -6.567027, -6.497343,
     -6.427665, -6.357994, -6.288328, -6.218668, -6.149014,
     -6.079366, -6.009724, -5.940088, -5.870457, -5.800831,
     -5.731211, -5.661597, -5.591988, -5.522384, -5.452785,
     -5.383192, -5.313603, -5.244020, -5.174441, -5.104867,
     -5.035299, -4.965734, -4.896175, -4.826620, -4.757070,
     -4.687524, -4.617983, -4.548446, -4.478913, -4.409384,
     -4.339860, -4.270340, -4.200823, -4.131311, -4.061802,
     -3.992298, -3.922797, -3.853299, -3.783806, -3.714315,
     -3.644829, -3.575346, -3.505866, -3.436389, -3.366916,
     -3.297445, -3.227978, -3.158514, -3.089053, -3.019595,
     -2.950139, -2.880687, -2.811237, -2.741789, -2.672344,
     -2.602902, -2.533462, -2.464025, -2.394589, -2.325156,
     -2.255726, -2.186297, -2.116870, -2.047446, -1.978023,
     -1.908602, -1.839183, -1.769765, -1.700350, -1.630935,
     -1.561523, -1.492111, -1.422702, -1.353293, -1.283886,
     -1.214480, -1.145075, -1.075671, -1.006268, -0.936866,
     -0.867465, -0.798065, -0.728666, -0.659267, -0.589869,
     -0.520471, -0.451074, -0.381677, -0.312281, -0.242885,
     -0.173489, -0.104093, -0.034698,  0.034698,  0.104093,
      0.173489,  0.242885,  0.312281,  0.381677,  0.451074,
      0.520471,  0.589869,  0.659267,  0.728666,  0.798065,
      0.867465,  0.936866,  1.006268,  1.075671,  1.145075,
      1.214480,  1.283886,  1.353293,  1.422702,  1.492111,
```

```
      1.561523,  1.630935,  1.700350,  1.769765,  1.839183,
      1.908602,  1.978023,  2.047446,  2.116870,  2.186297,
      2.255726,  2.325156,  2.394589,  2.464025,  2.533462,
      2.602902,  2.672344,  2.741789,  2.811237,  2.880687,
      2.950139,  3.019595,  3.089053,  3.158514,  3.227978,
      3.297445,  3.366916,  3.436389,  3.505866,  3.575346,
      3.644829,  3.714315,  3.783806,  3.853299,  3.922797,
      3.992298,  4.061802,  4.131311,  4.200823,  4.270340,
      4.339860,  4.409384,  4.478913,  4.548446,  4.617983,
      4.687524,  4.757070,  4.826620,  4.896175,  4.965734,
      5.035299,  5.104867,  5.174441,  5.244020,  5.313603,
      5.383192,  5.452785,  5.522384,  5.591988,  5.661597,
      5.731211,  5.800831,  5.870457,  5.940088,  6.009724,
      6.079366,  6.149014,  6.218668,  6.288328,  6.357994,
      6.427665,  6.497343,  6.567027,  6.636717,  6.706414,
      6.776117,  6.845826,  6.915543,  6.985266,  7.054996,
      7.124733,  7.194478,  7.264232,  7.333994,  7.403767,
      7.473552,  7.543352,  7.613171,  7.683015,  7.752894,
      7.822824,  7.892826,  7.962935,  8.033205,  8.103716,
      8.174593,  8.246032,  8.318330,  8.391961,  8.467670,
      8.546672,  8.630998,  8.724244,  8.833358,  8.974048,
      9.195524 }
```

Quantizadores Lloyd-Max para a função laplaciana

```
lapl_d_1bit[2] = {-100.000000, 0.000000 }
lapl_r_1bit[2] = {  -0.701778, 0.701778 }


lapl_d_2bit[4]={-100.000000, -1.120161, 0.000000, 1.120161 }
lapl_r_2bit[4]={  -1.822276, -0.418046, 0.418046, 1.822276 }


lapl_d_3bit[8] = {
     -100.000000, -2.369937, -1.249229, -0.531984,
        0.000000,  0.531984,  1.249229,  2.369937 }
lapl_r_3bit[8] = {
       -3.072430, -1.667445, -0.831013, -0.232955,
        0.232955,  0.831013,  1.667445,  3.072430 }


lapl_d_4bit[16] = {
     -100.000000, -3.728556, -2.606046, -1.886731, -1.352340,
       -0.925900, -0.570701, -0.266270,  0.000000,  0.266270,
        0.570701,  0.925900,  1.352340,  1.886731,  2.606046,
```

```
        3.728556 }
lapl_r_4bit[16] = {
       -4.431911, -3.025201, -2.186890, -1.586573, -1.118107,
       -0.733693, -0.407708, -0.124832,  0.124832,  0.407708,
        0.733693,  1.118107,  1.586573,  2.186890,  3.025201,
        4.431911 }


lapl_d_5bit[32] = {
     -100.000000, -5.131228, -4.009276, -3.291191, -2.758341,
       -2.333536, -1.979871, -1.676701, -1.411277, -1.175163,
       -0.962486, -0.768994, -0.591508, -0.427593, -0.275338,
       -0.133219,  0.000000,  0.133219,  0.275338,  0.427593,
        0.591508,  0.768994,  0.962486,  1.175163,  1.411277,
        1.676701,  1.979871,  2.333536,  2.758341,  3.291191,
        4.009276,  5.131228 }
lapl_r_5bit[32] = {
       -5.834531, -4.427926, -3.590627, -2.991754, -2.524927,
       -2.142144, -1.817597, -1.535805, -1.286749, -1.063577,
       -0.861396, -0.676592, -0.506424, -0.348761, -0.201914,
       -0.064524,  0.064524,  0.201914,  0.348761,  0.506424,
        0.676592,  0.861396,  1.063577,  1.286749,  1.535805,
        1.817597,  2.142144,  2.524927,  2.991754,  3.590627,
        4.427926,  5.834531 }


lapl_d_6bit[64] = {
       -100.000000, -7.008477, -5.883158, -5.160668, -4.622165,
      -4.190438, -3.828602, -3.516050, -3.240080, -2.992314,
      -2.766943, -2.559782, -2.367730, -2.188428, -2.020051,
      -1.861163, -1.710616, -1.567486, -1.431018, -1.300592,
      -1.175694, -1.055894, -0.940829, -0.830194, -0.723725,
      -0.621197, -0.522415, -0.427207, -0.335422, -0.246925,
      -0.161594, -0.079320,  0.000000,  0.079320,  0.161594,
       0.246925,  0.335422,  0.427207,  0.522415,  0.621197,
       0.723725,  0.830194,  0.940829,  1.055894,  1.175694,
       1.300592,  1.431018,  1.567486,  1.710616,  1.861163,
       2.020051,  2.188428,  2.367730,  2.559782,  2.766943,
       2.992314,  3.240080,  3.516050,  3.828602,  4.190438,
       4.622165,  5.160668,  5.883158,  7.008477 }
lapl_r_6bit[64] = {
      -7.713250, -6.303703, -5.462612, -4.858724, -4.385607,
      -3.995269, -3.661935, -3.370165, -3.109995, -2.874633,
      -2.659252, -2.460312, -2.275148, -2.101709, -1.938394,
      -1.783932, -1.637300, -1.497671, -1.364365, -1.236820,
      -1.114569, -0.997219, -0.884439, -0.775948, -0.671502,
      -0.570893, -0.473938, -0.380477, -0.290367, -0.203482,
      -0.119706, -0.038933,  0.038933,  0.119706,  0.203482,
       0.290367,  0.380477,  0.473938,  0.570893,  0.671502,
```

```
       0.775948,  0.884439,  0.997219,  1.114569,  1.236820,
       1.364365,  1.497671,  1.637300,  1.783932,  1.938394,
       2.101709,  2.275148,  2.460312,  2.659252,  2.874633,
       3.109995,  3.370165,  3.661935,  3.995269,  4.385607,
       4.858724,  5.462612,  6.303703,  7.713250 }


lapl_d_7bit[128] = {
      -100.000000, -13.564157, -12.419733, -11.670579, -11.097513,
       -10.623559, -10.212215,  -9.843333,  -9.504715,  -9.188509,
     -8.889441,  -8.603860,  -8.329182,  -8.063549,  -7.805606,
     -7.554356,  -7.309061,  -7.069171,  -6.834273,  -6.604054,
     -6.378280,  -6.156770,  -5.939384,  -5.726016,  -5.516581,
     -5.311012,  -5.109255,  -4.911264,  -4.717004,  -4.526440,
     -4.339545,  -4.156293,  -3.976659,  -3.800622,  -3.628159,
     -3.459249,  -3.293872,  -3.132006,  -2.973630,  -2.818724,
     -2.667266,  -2.519233,  -2.374603,  -2.233353,  -2.095461,
     -1.960901,  -1.829648,  -1.701678,  -1.576964,  -1.455479,
     -1.337196,  -1.222086,  -1.110120,  -1.001269,  -0.895500,
     -0.792784,  -0.693086,  -0.596374,  -0.502614,  -0.411770,
     -0.323806,  -0.238686,  -0.156370,  -0.076822,   0.000000,
      0.076822,   0.156370,   0.238686,   0.323806,   0.411770,
      0.502614,   0.596374,   0.693086,   0.792784,   0.895500,
      1.001269,   1.110120,   1.222086,   1.337196,   1.455479,
      1.576964,   1.701678,   1.829648,   1.960901,   2.095461,
      2.233353,   2.374603,   2.519233,   2.667266,   2.818724,
      2.973630,   3.132006,   3.293872,   3.459249,   3.628159,
      3.800622,   3.976659,   4.156293,   4.339545,   4.526440,
      4.717004,   4.911264,   5.109255,   5.311012,   5.516581,
      5.726016,   5.939384,   6.156770,   6.378280,   6.604054,
      6.834273,   7.069171,   7.309061,   7.554356,   7.805606,
      8.063549,   8.329182,   8.603860,   8.889441,   9.188509,
      9.504715,   9.843333,  10.212215,  10.623559,  11.097513,
     11.670579,  12.419733,  13.564157 }
lapl_r_7bit[128] = {
     -14.276693, -12.851621, -11.987846, -11.353311, -10.841716,
     -10.405402, -10.019027,  -9.667639,  -9.341792,  -9.035226,
      -8.743655,  -8.464065,  -8.194300,  -7.932798,  -7.678413,
      -7.430298,  -7.187824,  -6.950518,  -6.718027,  -6.490082,
      -6.266479,  -6.047061,  -5.831707,  -5.620324,  -5.412837,
      -5.209186,  -5.009323,  -4.813206,  -4.620801,  -4.432079,
      -4.247011,  -4.065574,  -3.887744,  -3.713499,  -3.542818,
      -3.375680,  -3.212064,  -3.051948,  -2.895313,  -2.742136,
      -2.592395,  -2.446070,  -2.303136,  -2.163571,  -2.027351,
      -1.894451,  -1.764846,  -1.638510,  -1.515418,  -1.395541,
      -1.278851,  -1.165321,  -1.054920,  -0.947618,  -0.843383,
      -0.742184,  -0.643988,  -0.548760,  -0.456467,  -0.367072,
      -0.280540,  -0.196831,  -0.115909,  -0.037734,   0.037734,
       0.115909,   0.196831,   0.280540,   0.367072,   0.456467,
```

```
        0.548760,   0.643988,   0.742184,   0.843383,   0.947618,
        1.054920,   1.165321,   1.278851,   1.395541,   1.515418,
        1.638510,   1.764846,   1.894451,   2.027351,   2.163571,
        2.303136,   2.446070,   2.592395,   2.742136,   2.895313,
        3.051948,   3.212064,   3.375680,   3.542818,   3.713499,
        3.887744,   4.065574,   4.247011,   4.432079,   4.620801,
        4.813206,   5.009323,   5.209186,   5.412837,   5.620324,
        5.831707,   6.047061,   6.266479,   6.490082,   6.718027,
        6.950518,   7.187824,   7.430298,   7.678413,   7.932798,
        8.194300,   8.464065,   8.743655,   9.035226,   9.341792,
        9.667639,  10.019027,  10.405402,  10.841716,  11.353311,
       11.987846,  12.851621,  14.276693 }


lapl_d_8bit[256] = {
     -100.000000, -38.871844, -37.661231, -36.818089, -36.125082,
      -35.508204, -34.934176, -34.386015, -33.854453, -33.334241,
      -32.822337, -32.316961, -31.817058, -31.322007, -30.831434,
      -30.345122, -29.862939, -29.384808, -28.910683, -28.440538,
      -27.974356, -27.512130, -27.053853, -26.599524, -26.149140,
      -25.702701, -25.260207, -24.821655, -24.387048, -23.956383,
      -23.529660, -23.106878, -22.688038, -22.273136, -21.862173,
      -21.455147, -21.052057, -20.652900, -20.257675, -19.866380,
      -19.479013, -19.095572, -18.716054, -18.340456, -17.968776,
      -17.601010, -17.237157, -16.877212, -16.521172, -16.169034,
      -15.820794, -15.476448, -15.135993, -14.799423, -14.466735,
      -14.137924, -13.812986, -13.491916, -13.174708, -12.861358,
      -12.551860, -12.246210, -11.944400, -11.646426, -11.352280,
      -11.061958, -10.775453, -10.492757, -10.213865, -9.938768,
       -9.667461,  -9.399934,  -9.136182,  -8.876194, -8.619965,
       -8.367484,  -8.118744,  -7.873735,  -7.632448,  -7.394874,
       -7.161003,  -6.930825,  -6.704330,  -6.481506,  -6.262344,
       -6.046831,  -5.834956,  -5.626708,  -5.422073,  -5.221039,
       -5.023593,  -4.829722,  -4.639412,  -4.452647,  -4.269415,
       -4.089699,  -3.913484,  -3.740753,  -3.571491,  -3.405680,
       -3.243303,  -3.084341,  -2.928777,  -2.776589,  -2.627760,
       -2.482268,  -2.340093,  -2.201212,  -2.065604,  -1.933246,
       -1.804114,  -1.678184,  -1.555431,  -1.435828,  -1.319350,
       -1.205970,  -1.095659,  -0.988388,  -0.884129,  -0.782849,
       -0.684519,  -0.589105,  -0.496576,  -0.406897,  -0.320032,
       -0.235948,  -0.154606,  -0.075969,   0.000000,   0.075969,
        0.154606,   0.235948,   0.320032,   0.406897,   0.496576,
        0.589105,   0.684519,   0.782849,   0.884129,   0.988388,
        1.095659,   1.205970,   1.319350,   1.435828,   1.555431,
        1.678184,   1.804114,   1.933246,   2.065604,   2.201212,
        2.340093,   2.482268,   2.627760,   2.776589,   2.928777,
        3.084341,   3.243303,   3.405680,   3.571491,   3.740753,
        3.913484,   4.089699,   4.269415,   4.452647,   4.639412,
        4.829722,   5.023593,   5.221039,   5.422073,   5.626708,
```

```
         5.834956,   6.046831,   6.262344,   6.481506,   6.704330,
         6.930825,   7.161003,   7.394874,   7.632448,   7.873735,
         8.118744,   8.367484,   8.619965,   8.876194,   9.136182,
         9.399934,   9.667461,   9.938768,  10.213865,  10.492757,
        10.775453,  11.061958,  11.352280,  11.646426,  11.944400,
        12.246210,  12.551860,  12.861358,  13.174708,  13.491916,
        13.812986,  14.137924,  14.466735,  14.799423,  15.135993,
        15.476448,  15.820794,  16.169034,  16.521172,  16.877212,
        17.237157,  17.601010,  17.968776,  18.340456,  18.716054,
        19.095572,  19.479013,  19.866380,  20.257675,  20.652900,
        21.052057,  21.455147,  21.862173,  22.273136,  22.688038,
        23.106878,  23.529660,  23.956383,  24.387048,  24.821655,
        25.260207,  25.702701,  26.149140,  26.599524,  27.053853,
        27.512130,  27.974356,  28.440538,  28.910683,  29.384808,
        29.862939,  30.345122,  30.831434,  31.322007,  31.817058,
        32.316961,  32.822337,  33.334241,  33.854453,  34.386015,
        34.934176,  35.508204,  36.125082,  36.818089,  37.661231,
        38.871844 }
lapl_r_8bit[256] = {
     -39.610408, -38.133280, -37.189181, -36.446996, -35.803169,
     -35.213239, -34.655112, -34.116917, -33.591989, -33.076492,
     -32.568183, -32.065739, -31.568378, -31.075635, -30.587234,
     -30.103011, -29.622868, -29.146748, -28.674618, -28.206457,
     -27.742255, -27.282004, -26.825702, -26.373346, -25.924935,
     -25.480468, -25.039945, -24.603366, -24.170730, -23.742036,
     -23.317284, -22.896473, -22.479602, -22.066670, -21.657676,
     -21.252618, -20.851495, -20.454305, -20.061045, -19.671715,
     -19.286312, -18.904832, -18.527275, -18.153637, -17.783915,
     -17.418106, -17.056208, -16.698216, -16.344128, -15.993940,
     -15.647648, -15.305249, -14.966737, -14.632109, -14.301361,
     -13.974488, -13.651484, -13.332347, -13.017069, -12.705647,
     -12.398074, -12.094345, -11.794455, -11.498396, -11.206164,
     -10.917752, -10.633153, -10.352361, -10.075369,  -9.802168,
      -9.532753,  -9.267116,  -9.005248,  -8.747141,  -8.492788,
      -8.242180,  -7.995308,  -7.752162,  -7.512734,  -7.277014,
      -7.044992,  -6.816658,  -6.592001,  -6.371011,  -6.153676,
      -5.939986,  -5.729927,  -5.523488,  -5.320657,  -5.121421,
      -4.925766,  -4.733678,  -4.545145,  -4.360150,  -4.178680,
      -4.000718,  -3.826249,  -3.655257,  -3.487725,  -3.323636,
      -3.162971,  -3.005712,  -2.851841,  -2.701338,  -2.554182,
      -2.410354,  -2.269832,  -2.132593,  -1.998616, -1.867877,
      -1.740352,  -1.616016,  -1.494845,  -1.376812,  -1.261889,
      -1.150051,  -1.041267,  -0.935509,  -0.832748,  -0.732951,
      -0.636087,  -0.542124,  -0.451028,  -0.362765,  -0.277300,
      -0.194596,  -0.114616,  -0.037323,   0.037323,   0.114616,
       0.194596,   0.277300,   0.362765,   0.451028,   0.542124,
       0.636087,   0.732951,   0.832748,   0.935509,   1.041267,
       1.150051,   1.261889,   1.376812,   1.494845,   1.616016,
       1.740352,   1.867877,   1.998616,   2.132593,   2.269832,
```

```
 2.410354,   2.554182,   2.701338,   2.851841,   3.005712,
 3.162971,   3.323636,   3.487725,   3.655257,   3.826249,
 4.000718,   4.178680,   4.360150,   4.545145,   4.733678,
 4.925766,   5.121421,   5.320657,   5.523488,   5.729927,
 5.939986,   6.153676,   6.371011,   6.592001,   6.816658,
 7.044992,   7.277014,   7.512734,   7.752162,   7.995308,
 8.242180,   8.492788,   8.747141,   9.005248,   9.267116,
 9.532753,   9.802168,  10.075369,  10.352361,  10.633153,
10.917752,  11.206164,  11.498396,  11.794455,  12.094345,
12.398074,  12.705647,  13.017069,  13.332347,  13.651484,
13.974488,  14.301361,  14.632109,  14.966737,  15.305249,
15.647648,  15.993940,  16.344128,  16.698216,  17.056208,
17.418106,  17.783915,  18.153637,  18.527275,  18.904832,
19.286312,  19.671715,  20.061045,  20.454305,  20.851495,
21.252618,  21.657676,  22.066670,  22.479602,  22.896473,
23.317284,  23.742036,  24.170730,  24.603366,  25.039945,
25.480468,  25.924935,  26.373346,  26.825702,  27.282004,
27.742255,  28.206457,  28.674618,  29.146748,  29.622868,
30.103011,  30.587234,  31.075635,  31.568378,  32.065739,
32.568183,  33.076492,  33.591989,  34.116917,  34.655112,
35.213239,  35.803169,  36.446996,  37.189181,  38.133280,
39.610408 }
```

APÊNDICE B

ALGORITMOS RÁPIDOS PARA O CÁLCULO DA DCT-II

1. Introdução

O desenvolvimento de algoritmos eficientes para a computação da DCT (mais especificamente da DCT-II) teve início logo após Ahmed et al. **[157]** terem publicado seus trabalhos sobre a DCT. Era natural que nas tentativas iniciais para computação da DCT fossem utilizadas como base algoritmos rápidos da Transformada de Fourier (FFT). Embora a DCT-II não tivesse sido desenvolvida como uma versão discretizada da Transformada Cosseno de Fouriar (FCT), foram exploradas suas relações com a Transformada Discreta de Fourier (DFT) para os desenvolvimentos iniciais de seus algoritmos computacionais. Haralick **[145]** reportou a computação de uma DCT de N pontos usando a FFT de 2N pontos. Outros trabalhos seguindo a mesma linha surgiram logo em seguida: Narashima and Peterson **[158]**, Tseng e Miller **[148]**, Makhoul **[149]**, Duhamed **[159]**, Vetterli and Nussbaumer **[160]**, Chan and Ho **[144]** e vários outros contribuíram para o avanço nessa direção.

Algoritmos rápidos foram também obtidos considerando-se a fatoração da matriz da DCT. Chen, Smith and Fralick **[146]** reportaram tal desenvolvimento. Uma vez que a fatorização de matriz não é única, existem várias outras formas de algoritmos rápidos. Alguns esquemas de fatoração recaem na decimação no tempo, outros na decimação na freqüência. Corrington **[161],** Wang **[162]**, Lee **[163,164]**, Suehiro and Hatori **[165]**, Yip and Rao **[166-168]** e vários outros contribuíram para esse desenvolvimento. Algoritmos computacionais rápidos para a DCT podem também ser obtidos através da computação de outras transformadas discretas como os obtidos por: Hein and Ahmed **[169]**, Jones, Hein and Knauer **[170]**, Venkataraman et al. **[171]**, Malvar **[172-174]** e Nagesha **[175]**. Algoritmos rápidos podem ser também computadas através da decomposição em fatores primos (Yang and Narashima **[176]**), através da computação recursiva (Hou **[177]**) e por rotação planar **[178-181]**, além de outros algoritmos como split-radix **[182]**.

Entre os algoritmos mais eficientes para a computação da DCT está aquele desenvolvido por Chan and Ho **[144]**. Esse algoritmo segue uma lógica similar a utilizada no algoritmo da computação da FFT, exceto pelo fato de o mesmo ser precedido de um esquema de reordenação dos dados de entrada. Na derivação, é feita a suposição de que N é um inteiro múltiplo de 2. A seguir, é feita a exploração desses algoritmos para a computação da Transformada Rápida Cosseno (DFT) desenvolvida por Chan and Ho.

2. Algoritmo de CHAN e HO para a Transformada Rápida Cosseno (FCT)

A DCT-II e a sua inversa IDCT-II unidimensional (1D) de uma seqüência *x(n)* de entrada são dadas por:

$$X(k) = \frac{2}{N} e(k) \sum_{n=0}^{N-1} x(n) \cos\left[\frac{(2n+1)\pi k}{2N}\right] \tag{1}$$

$$x(k) = \sum_{k=0}^{N-1} e(k) X(k) \cos\left[\frac{(2n+1)\pi k}{2N}\right] \tag{2}$$

onde

$$e(k) = \begin{cases} \frac{1}{\sqrt{2}} & ; \quad k = 0 \\ 1 & ; \quad k \neq 0 \end{cases} \qquad k=0,1,2,\ldots,N-1 \tag{3}$$

Considerando-se a seqüência {***x(n)***} de *N* pontos, *N* par e múltiplo de 2, a seqüência {***x(n)***} pode ser reordenada na seguinte forma:

$$\begin{aligned} y(n) &= x(2n) \\ y(N-1-n) &= x(2n+1) \end{aligned} \qquad n = 0,1,2,\ldots,\frac{N}{2}-1 \tag{4}$$

Então, a DCT aplicada na seqüência {***y(n)***} resulta em (exceto pelo fator escalar):

$$X(k)=\sum_{n=0}^{\frac{N}{2}-1} y(n)\cos\left[\frac{(2(2n)+1)\pi k}{2N}\right]+\sum_{n=0}^{\frac{N}{2}-1} y(N-1-n)\cos\left[\frac{(2(2n+1)+1)\pi k}{2N}\right]$$

$$X(k)=\sum_{n=0}^{\frac{N}{2}-1} y(n)\cos\left[\frac{(4n+1)\pi k}{2N}\right]+\sum_{n=0}^{\frac{N}{2}-1} y(N-1-n)\cos\left[\frac{(4n+3)\pi k}{2N}\right] \quad (5)$$

Fazendo $j = N-1-n <=> n = N-1-j$.

para $n = 0 => j = N-1$;

para $n = N/2 - 1 => j = N/2$

então, a segunda equação à direita em (5) resulta em:

$$\sum_{n=0}^{\frac{N}{2}-1} y(N-1-n)\cos\left[\frac{(4n+3)\pi k}{2N}\right] \Leftrightarrow \sum_{j=N-1}^{\frac{N}{2}} y(j)\cos\left[\frac{(4(N-1-j)+3)\pi k}{2N}\right]$$

$$=\sum_{j=N-1}^{\frac{N}{2}} y(j)\cos\left[\frac{(4N-4-4j+3)\pi k}{2N}\right] \Rightarrow \sum_{j=N-1}^{\frac{N}{2}} y(j)\cos\left[\frac{(-(4j+1)+4N)\pi k}{2N}\right]$$

Como a função cosseno é periódica, então, $\cos(\Theta+4N)=\cos(\Theta)$, portanto:

$$\sum_{j=N-1}^{\frac{N}{2}} y(j)\cos\left[\frac{(-(4j+1)+4N)\pi k}{2N}\right]=\sum_{j=N-1}^{\frac{N}{2}} y(j)\cos\left[\frac{-(4j+1)\pi k}{2N}\right]$$

Como $\cos(-\Theta)=\cos(\Theta)$, então:

$$\sum_{j=N-1}^{\frac{N}{2}} y(j)\cos\left[\frac{-(4j+1)\pi k}{2N}\right] = \sum_{j=N-1}^{\frac{N}{2}} y(j)\cos\left[\frac{(4j+1)\pi k}{2N}\right] \quad (6)$$

Invertendo-se o somatório da Equação (6) tem-se que:

$$\sum_{j=N-1}^{\frac{N}{2}} y(j)\cos\left[\frac{(4j+1)\pi k}{2N}\right] = \sum_{j=\frac{N}{2}}^{N-1} y(j)\cos\left[\frac{(4j+1)\pi k}{2N}\right] \tag{7}$$

Substituindo "*j*" por "*n*" na Equação (7) e levando-se na Equação (5) tem-se:

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} y(n)\cos\left[\frac{(4n+1)\pi k}{2N}\right] + \sum_{n=\frac{N}{2}}^{N-1} y(n)\cos\left[\frac{(4n+1)\pi k}{2N}\right] \tag{8}$$

A Equação (8) resulta em:

$$X(k) = \sum_{n=0}^{N-1} y(n)\cos\left[\frac{(4n+1)\pi k}{2N}\right] \tag{9}$$

Fazendo a decimação em freqüência da Equação (9), expressando-a em elementos pares e ímpares, tem-se:

Termos Pares:

$$X(2k) = \sum_{n=0}^{N-1} y(n)\cos\left[\frac{(4n+1)\pi 2k}{2N}\right] \tag{10}$$

Termos ímpares:

$$X(2k+1) = \sum_{n=0}^{N-1} y(n)\cos\left[\frac{(4n+1)\pi(2k+1)}{2N}\right] \tag{11}$$

Da Equação (10), quando $n = m + N/2$, então tem-se:

$$\cos\left[\frac{(4n+1)\pi 2k}{2N}\right] = \cos\left[\frac{(4(m+\frac{N}{2})+1)\pi 2k}{2N}\right] = \cos\left[\frac{(4m+2N+1)\pi 2k}{2N}\right] = \cos\left[\frac{(4m+1)\pi 2k}{2N}\right] \tag{12}$$

Portanto, para "*n*" e "*n+N/2*" os valores dos cossenos são iguais. Logo, a Equação (10) pode ser reescrita como:

$$X(2k) = \sum_{n=0}^{\frac{N}{2}-1} \left[y(n) + y\left(n + \frac{N}{2}\right) \right] \cos\left[\frac{(4n+1)\pi k}{2\left(\frac{N}{2}\right)} \right] \tag{13}$$

Da Equação (11) tem-se que:

$$\cos\left[\frac{(4n+1)\pi(2k+1)}{2N} \right] = \cos\left[\frac{(4n+1)\pi 2k + (4n+1)\pi}{2N} \right] \tag{14}$$

Para $n = m+N/2$ tem-se:

$$\cos\left[\frac{4\left(n+\frac{N}{2}\right)+1)\pi 2k + \left(4\left(n+\frac{N}{2}\right)+1\right)\pi}{2N} \right] = \cos\left[\frac{(4n+2N+1)\pi 2k + (4n+2N+1)\pi}{2N} \right]$$

$$\cos\left[\frac{(4n+2N+1)\pi 2k + (4n+1)\pi + \pi 2N}{2N} \right] = \cos\left[\frac{(4n+1)\pi 2k + (4n+1)\pi}{2N} + \pi \right] = \cos\left[\frac{(4n+1)\pi(4k+1)}{2N} + \pi \right] \tag{15}$$

Portanto, para $n = m+N/2$ tem-se que:

$$\cos\left[\frac{(4n+1)\pi(2k+1)}{2N} + \pi \right] = -\cos\left[\frac{(4n+1)\pi(4k+1)}{2N} \right] \tag{16}$$

Logo, a Equação (11) pode ser reescrita como:

$$X(2k+1) = \sum_{n=0}^{\frac{N}{2}-1} \left[y(n) - y\left(n + \frac{N}{2}\right) \right] \cos\left[\frac{(4n+1)\pi(2k+1)}{2\left(\frac{N}{2}\right)} \right] \tag{17}$$

Mas:

$$\cos\left[\frac{(4n+1)\pi(2k+1)}{2N} \right] = \cos[\Theta(2k+1)] = 2\cos[\Theta]\cos[\Theta 2k] - \cos[\Theta(2k-1)] \tag{18}$$

Assim, a Equação (17) pode ser reescrita como:

$$X(2k+1) = \sum_{n=0}^{\frac{N}{2}-1} \left[y(n) - y\left(n + \frac{N}{2}\right) \right] \{2\cos[\Theta]\cos[\Theta 2k] - \cos[\Theta(2k-1)]\} \tag{19}$$

$$X(2k+1)=\sum_{n=0}^{\frac{N}{2}-1}\left[y(n)-y(n+\tfrac{N}{2})\right]\left\{2\cos[\Theta]\cos[\Theta 2k]\right\}-\underbrace{\sum_{n=0}^{\frac{N}{2}-1}\left[y(n)-y(n+\tfrac{N}{2})\right]\left\{\cos[\Theta(2k-1)]\right\}}_{X(2k-1)}$$

$$X(2k+1)=\sum_{n=0}^{\frac{N}{2}-1}\left[y(n)-y(n+\tfrac{N}{2})\right]\left\{2\cos[\Theta]\cos[\Theta 2k]\right\}-X(2k-1) \tag{20}$$

$$X(2k+1)=\sum_{n=0}^{\frac{N}{2}-1}\left[y(n)-y(n+\tfrac{N}{2})\right]\left\{2\cos[\tfrac{(4n+1)\pi}{2N}]\cos[\tfrac{(4n+1)\pi k}{2(\frac{N}{2})}]\right\}-X(2k-1) \tag{21}$$

Das Equações (17) e (21), fazendo:

$$\begin{cases} x_{00}(n)=y(n)+y(n+\tfrac{N}{2}) \\ x_{01}(n)=\left[y(n)-y(n+\tfrac{N}{2})\right]\left(2\cos\left[\tfrac{(4n+1)\pi}{2N}\right]\right) \end{cases} \tag{22}$$

obtém-se:

$$2\cos\left(\tfrac{(4n+1)\pi}{2N}\right)=2C_{2N}^{(4n+1)} \tag{23}$$

As Equações (22) representam as operações de borboleta (*butterfly*) para o primeiro estágio da FCT. Portanto,

$$X(2k)=\sum_{n=0}^{\frac{N}{2}-1}x_{00}(n)\cos\left[\tfrac{(4n+1)\pi k}{2(\frac{N}{2})}\right] \tag{24}$$

$$X(2k+1)=\sum_{n=0}^{\frac{N}{2}-1}x_{01}(n)\cos\left[\tfrac{(4n+1)\pi k}{2(\frac{N}{2})}\right]-X(2k-1) \tag{25}$$

Fazendo:

$$\begin{cases} Y_{00}(k)=X(2k) \\ Y_{01}(k)=X(2k+1)+X(2k-1) \end{cases} \tag{26}$$

Temos que as Equações (24) e (25) podem ser reescritas como:

$$Y_{00}(k) = \sum_{n=0}^{\frac{N}{2}-1} x_{00}(n)\cos\left[\frac{(4n+1)\pi k}{2(\frac{N}{2})}\right] \Rightarrow FCT\,de\,\tfrac{N}{2}\,pontos \tag{27}$$

$$Y_{01}(k) = \sum_{n=0}^{\frac{N}{2}-1} x_{01}(n)\cos\left[\frac{(4n+1)\pi k}{2(\frac{N}{2})}\right] \Rightarrow FCT\ de\ \tfrac{N}{2}\ pontos \tag{28}$$

Dividindo as Equações (27) e (28) em seqüências pares e ímpares, tem-se:

Para a Equação (27):

$$Y_{00}(2k) = \sum_{n=0}^{\frac{N}{4}-1}\left[x_{00}(n) + x_{00}(n+\tfrac{N}{4})\right]\cos\left[\frac{(4n+1)\pi k}{2(\frac{N}{4})}\right] \tag{29}$$

$$Y_{00}(2k+1) = \sum_{n=0}^{\frac{N}{4}-1}\left\{\left[x_{00}(n) - x_{00}(n+\tfrac{N}{4})\right]2\cos\left[\frac{(4n+1)\pi}{2(\frac{N}{2})}\right]\right\}\left\{\cos\left[\frac{(4n+1)\pi k}{2(\frac{N}{4})}\right]\right\} - Y_{00}(2k-1) \tag{30}$$

$$\begin{cases} x_{10}(n) = \left[x_{00}(n) + x_{00}(n+\frac{N}{4})\right] \\ x_{11}(n) = \left[x_{00}(n) - x_{00}(n+\frac{N}{4})\right]\left(2\cos\left[\frac{(4n+1)\pi}{2(\frac{N}{2})}\right]\right) \end{cases} \tag{31}$$

$$2\cos\left(\frac{(4n+1)\pi}{2(\frac{N}{2})}\right) = 2C^{(4n+1)}_{2(\frac{N}{2})} \tag{32}$$

Para a Equação (28):

$$Y_{01}(2k) = \sum_{n=0}^{\frac{N}{4}-1}\left[x_{01}(n) + x_{01}(n+\tfrac{N}{4})\right]\cos\left[\frac{(4n+1)\pi k}{2(\frac{N}{4})}\right] \tag{33}$$

$$Y_{01}(2k+1) = \sum_{n=0}^{\frac{N}{4}-1}\left\{\left[x_{01}(n) - x_{01}(n+\tfrac{N}{4})\right]2\cos\left[\frac{(4n+1)\pi}{2(\frac{N}{2})}\right]\right\}\left\{\cos\left[\frac{(4n+1)\pi k}{2(\frac{N}{4})}\right]\right\} - Y_{01}(2k-1) \tag{34}$$

De (33) e (34), tem-se:

$$\begin{cases} x_{12}(n) = \left[x_{01}(n) + x_{01}\left(n+\frac{N}{4}\right)\right] \\ x_{13}(n) = \left[x_{01}(n) - x_{01}\left(n+\frac{N}{4}\right)\right]\left(2\cos\left[\frac{(4n+1)\pi}{2\left(\frac{N}{2}\right)}\right]\right) \end{cases} \tag{35}$$

As Equações (31) e (35) representam as operações de borboleta (*butterfly*) para o segundo estágio da FCT. As Equações (29), (30), (33) e (34) resultam em:

$$\begin{cases} Y_{00}(2k) = \sum_{n=0}^{\frac{N}{4}-1}\left[x_{10}(n)\right]\cos\left[\frac{(4n+1)\pi k}{2\left(\frac{N}{4}\right)}\right] \\ Y_{00}(2k+1) + Y_{00}(2k-1) = \sum_{n=0}^{\frac{N}{4}-1}\left[x_{11}(n)\right]\cos\left[\frac{(4n+1)\pi k}{2\left(\frac{N}{4}\right)}\right] \end{cases} \tag{36}$$

$$\begin{cases} Y_{01}(2k) = \sum_{n=0}^{\frac{N}{4}-1}\left[x_{12}(n)\right]\cos\left[\frac{(4n+1)\pi k}{2\left(\frac{N}{4}\right)}\right] \\ Y_{01}(2k+1) + Y_{01}(2k-1) = \sum_{n=0}^{\frac{N}{4}-1}\left[x_{13}(n)\right]\cos\left[\frac{(4n+1)\pi k}{2\left(\frac{N}{4}\right)}\right] \end{cases} \tag{37}$$

Fazendo:

$$\begin{cases} Y_{10}(k) = Y_{00}(2k) \\ Y_{11}(k) = Y_{00}(2k+1) + Y_{00}(2k-1) \end{cases} \tag{38}$$

$$\begin{cases} Y_{12}(k) = Y_{01}(2k) \\ Y_{13}(k) = Y_{01}(2k+1) + Y_{01}(2k-1) \end{cases} \tag{39}$$

Temos que as Equações (36) e (37) podem ser reescritas como:

$$\begin{cases} Y_{10}(k) = \sum_{n=0}^{\frac{N}{4}-1}\left[x_{10}(n)\right]\cos\left[\frac{(4n+1)\pi k}{2\left(\frac{N}{4}\right)}\right] \\ Y_{11}(k) = \sum_{n=0}^{\frac{N}{4}-1}\left[x_{11}(n)\right]\cos\left[\frac{(4n+1)\pi k}{2\left(\frac{N}{4}\right)}\right] \end{cases} \tag{40}$$

$$\begin{cases} Y_{12}(k) = \sum_{n=0}^{\frac{N}{4}-1} [x_{12}(n)] cos\left[\frac{(4n+1)\pi k}{2(\frac{N}{4})}\right] \\ Y_{13}(k) = \sum_{n=0}^{\frac{N}{4}-1} [x_{13}(n)] cos\left[\frac{(4n+1)\pi k}{2(\frac{N}{4})}\right] \end{cases} \quad (41)$$

Dividindo as Equações (40) e (41) em seqüências pares e ímpares, tem-se novas equações que possibilitarão o desenvolvimento de novos estágios de borboletas (*butterfly*). Esses desenvolvimentos continuarão até que o último estágio de borboletas (*butterfly*) seja obtido, dependendo do número N de elementos da seqüência {***x(n)***} a ser transformada.

3. Exemplo

Para N = 8 teremos o seguinte desenvolvimento para a obtenção dos estágios de borboletas (*butterfly*) a partir das Equações (38) e (39):

$$Y_{10}(k) = \left[x_{10}(0) cos\left(\frac{\pi k}{4}\right) + x_{10}(1) cos\left(\frac{5\pi k}{4}\right)\right] \quad (42)$$

$$\begin{cases} Y_{10}(0) = [x_{10}(0) + x_{10}(1)] \\ Y_{10}(1) = [x_{10}(0) - x_{10}(1)] cos\left(\frac{\pi}{4}\right) \Rightarrow [x_{10}(0) - x_{10}(1)] C^{1}_{2(\frac{8}{4})} \end{cases} \quad (43)$$

$$Y_{11}(k) = \left[x_{11}(0) cos\left(\frac{\pi k}{4}\right) + x_{11}(1) cos\left(\frac{5\pi k}{4}\right)\right] \quad (44)$$

$$\begin{cases} Y_{11}(0) = [x_{11}(0) + x_{11}(1)] \\ Y_{11}(1) = [x_{11}(0) - x_{11}(1)] cos\left(\frac{\pi}{4}\right) \Rightarrow [x_{11}(0) - x_{11}(1)] C^{1}_{2(\frac{8}{4})} \end{cases} \quad (45)$$

$$Y_{12}(k) = \left[x_{12}(0) cos\left(\frac{\pi k}{4}\right) + x_{12}(1) cos\left(\frac{5\pi k}{4}\right)\right] \quad (46)$$

$$\begin{cases} Y_{12}(0) = [x_{12}(0) + x_{12}(1)] \\ Y_{12}(1) = [x_{12}(0) - x_{12}(1)] cos\left(\frac{\pi}{4}\right) \Rightarrow [x_{12}(0) - x_{12}(1)] C^{1}_{2(\frac{8}{4})} \end{cases} \quad (47)$$

$$Y_{13}(k) = \left[x_{13}(0)\cos\left(\frac{\pi k}{4}\right) + x_{13}(1)\cos\left(\frac{5\pi k}{4}\right)\right] \tag{48}$$

$$\begin{cases} Y_{13}(0) = \left[x_{13}(0) + x_{13}(1)\right] \\ Y_{13}(1) = \left[x_{13}(0) - x_{13}(1)\right]\cos\left(\frac{\pi}{4}\right) \Rightarrow \left[x_{13}(0) - x_{13}(1)\right] C^{1}_{2\left(\frac{8}{4}\right)} \end{cases} \tag{49}$$

Para relacionar os valores de saída do gráfico da FCT com os valores da entrada, é necessário trabalhar com as equações no sentido para traz.

Exemplo:

Das Equações (38) e (26) pode-se escrever:

$$\left.\begin{array}{l} Y_{10}(k) = Y_{00}(2k) \\ Y_{00}(k) = X(2k) \\ Y_{00}(2k) = X(4k) \end{array}\right\rangle \Leftrightarrow Y_{10}(k) = X(4k) \tag{50}$$

$$\left.\begin{array}{l} Y_{11}(k) = Y_{00}(2k+1) + Y_{00}(2k-1) \\ Y_{00}(k) = X(2k) \\ Y_{00}(2k+1) = X(2(2k+1)) = X(4k+2) \\ Y_{00}(2k-1) = X(2(2k-1)) = X(4k-2) \end{array}\right\rangle \Leftrightarrow Y_{11}(k) = X(4k+2) + X(4k-2) \tag{51}$$

Das Equações (39) e (26) pode-se escrever:

$$\left.\begin{array}{l} Y_{12}(k) = Y_{01}(2k) \\ Y_{01}(k) = X(2k+1) + X(2k-1) \\ Y_{01}(2k) = X(2(2k)+1) + X(2(2k)-1) \end{array}\right\rangle \Leftrightarrow Y_{12}(k) = X(4k+1) + X(4k-1) \tag{52}$$

$$\left.\begin{array}{l} Y_{13}(k) = Y_{01}(2k+1) + Y_{01}(2k-1) \\ Y_{01}(k) = X(2k+1) + X(2k-1) \\ Y_{01}(2k+1) = X(2(2k+1)+1) + X(2(2k+1)-1) \\ Y_{01}(2k-1) = X(2(2k-1)+1) + X(2(2k-1)-1) \end{array}\right\rangle \Leftrightarrow \begin{array}{l} Y_{13}(k) = X(4k+3) + X(4k+1) \\ \qquad\qquad + X(4k-1) + X(4k-3) \end{array}$$

(53)

Portanto,

$$
\begin{aligned}
&Y_{10}(0) \overset{0}{\Rightarrow} X(0) \\
&Y_{10}(1) \overset{1}{\Rightarrow} X(4) \\
&Y_{11}(0) \overset{2}{\Rightarrow} X(2)+X(-2) = 2X(2) \\
&Y_{11}(1) \overset{3}{\Rightarrow} X(6)+X(2) \\
&Y_{12}(0) \overset{4}{\Rightarrow} X(1)+X(-1) = 2X(1) \\
&Y_{12}(1) \overset{5}{\Rightarrow} X(5)+X(3) \\
&Y_{13}(0) \overset{6}{\Rightarrow} X(3)+X(1)+X(-1)+X(-3) = 2[X(3)+X(1)] \\
&Y_{13}(1) \overset{7}{\Rightarrow} X(7)+X(5)+X(3)+X(1)
\end{aligned}
\tag{54}
$$

Fazendo a ordenação bit-reverso tem-se:

$$
\begin{aligned}
&(0)\,000 \rightarrow 000\,(0) \quad (4)\,100 \rightarrow 001\,(1) \\
&(1)\,001 \rightarrow 100\,(4) \quad (5)\,101 \rightarrow 101\,(5) \\
&(2)\,010 \rightarrow 010\,(2) \quad (6)\,110 \rightarrow 011\,(3) \\
&(3)\,011 \rightarrow 110\,(6) \quad (7)\,111 \rightarrow 111\,(7)
\end{aligned}
\tag{55}
$$

Portanto, as Equações (54) resultam em:

$$
\begin{aligned}
&Y_{10}(0) \overset{0}{\Rightarrow} X(0) \\
&Y_{12}(0) \overset{4}{\Rightarrow} X(1)+X(-1) = 2X(1) \\
&Y_{11}(0) \overset{2}{\Rightarrow} X(2)+X(-2) = 2X(2) \\
&Y_{13}(0) \overset{6}{\Rightarrow} X(3)+X(1)+X(-1)+X(-3) = 2[X(3)+X(1)] \\
&Y_{10}(1) \overset{1}{\Rightarrow} X(4) \\
&Y_{12}(1) \overset{5}{\Rightarrow} X(5)+X(3) \\
&Y_{11}(1) \overset{3}{\Rightarrow} X(6)+X(2) \\
&Y_{13}(1) \overset{7}{\Rightarrow} X(7)+X(5)+X(3)+X(1)
\end{aligned}
\tag{56}
$$

Para a obtenção dos valores de saída, correspondentes aos valores ordenados da seqüência de entrada transformada em coeficientes FCT, realiza-se subtrações recursivas, obtendo-se:

$$
\begin{aligned}
&Y_{10}(0) \overset{0}{\Rightarrow} X(0) \\
&Y_{12}(0) \overset{4}{\Rightarrow} 2X(1)\left(*\tfrac{1}{2}\right) = X(1) \\
&Y_{11}(0) \overset{2}{\Rightarrow} 2X(2)\left(*\tfrac{1}{2}\right) = X(2) \\
&Y_{13}(0) \overset{6}{\Rightarrow} 2[X(3)+X(1)] - \left[X(3)+2X(1)\right] = X(3) \\
&Y_{10}(1) \overset{1}{\Rightarrow} X(4) \\
&Y_{12}(1) \overset{5}{\Rightarrow} X(5)+X(3)\left(-X(3)\right) = X(5) \\
&Y_{11}(1) \overset{3}{\Rightarrow} X(6)+X(2)\left(-X(2)\right) = X(6) \\
&Y_{13}(1) \overset{7}{\Rightarrow} X(7)+X(5)+X(3)+X(1) - \left[X(5)+X(3)+2X(1)\right] = X(7)
\end{aligned}
\tag{57}
$$

Os diagramas de fluxo para a realização da FCT de uma seqüência {***x(n)***} de entrada para $N = 8$, 16 e 32 são mostrados nas **Figuras** seguintes.

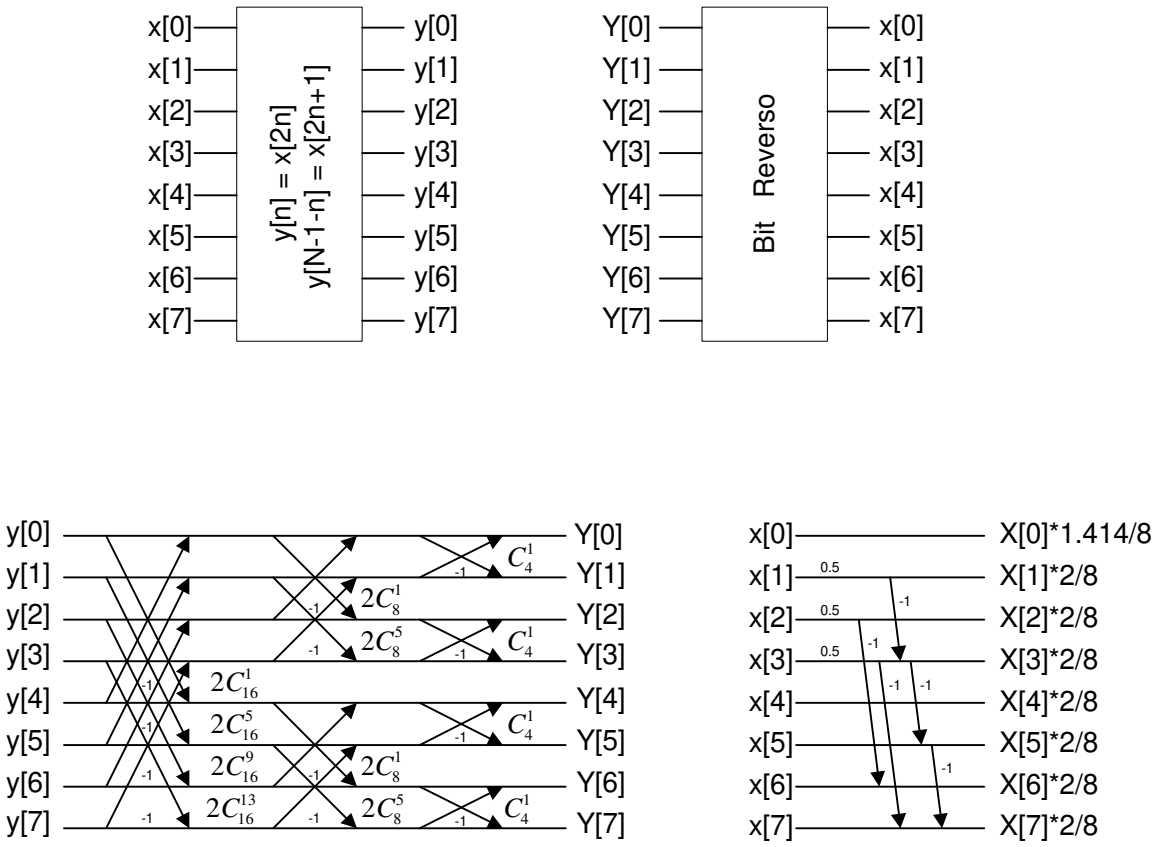


Fig. APB-1 – Fluxo da FCT de uma sequência **{*x[n]*}** para entradas **N = 8**

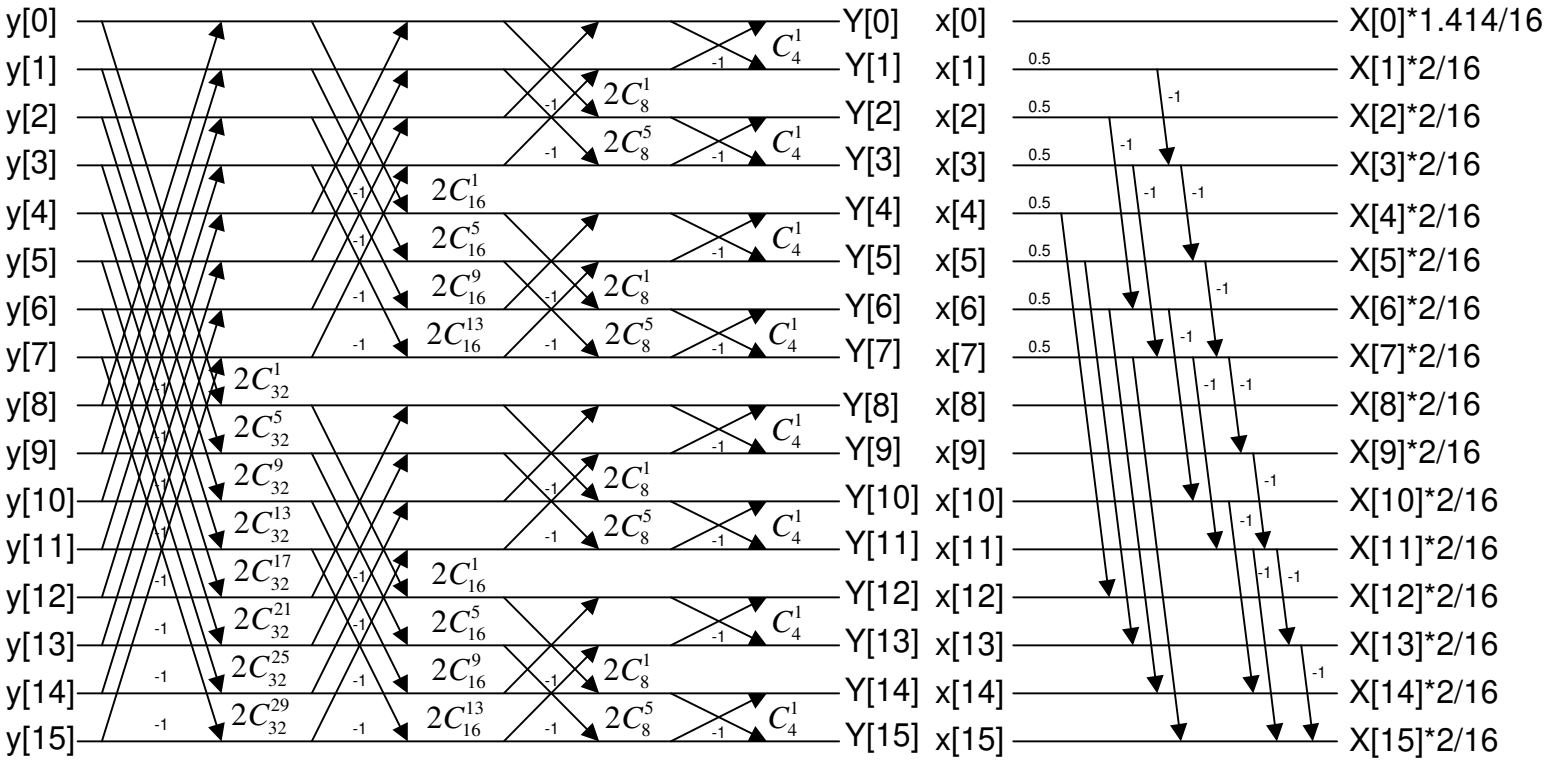


Fig. APB-2 – Fluxo da FCT de uma sequência **{*x[n]*}** para entradas **N = 16**

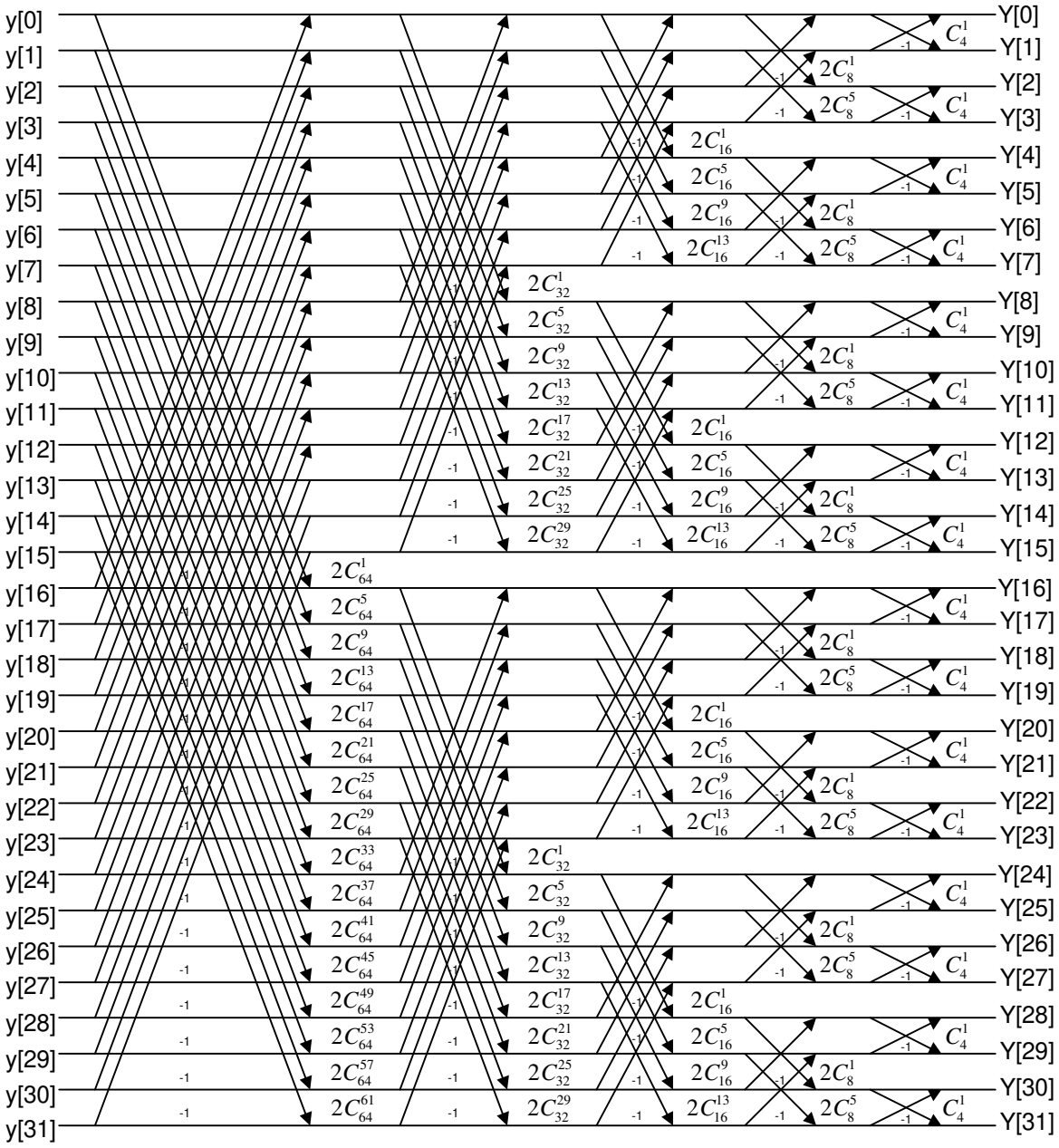


Fig. APB-3 – Fluxo da FCT de uma sequência **{*x[n]*}** para entradas **N = 32**

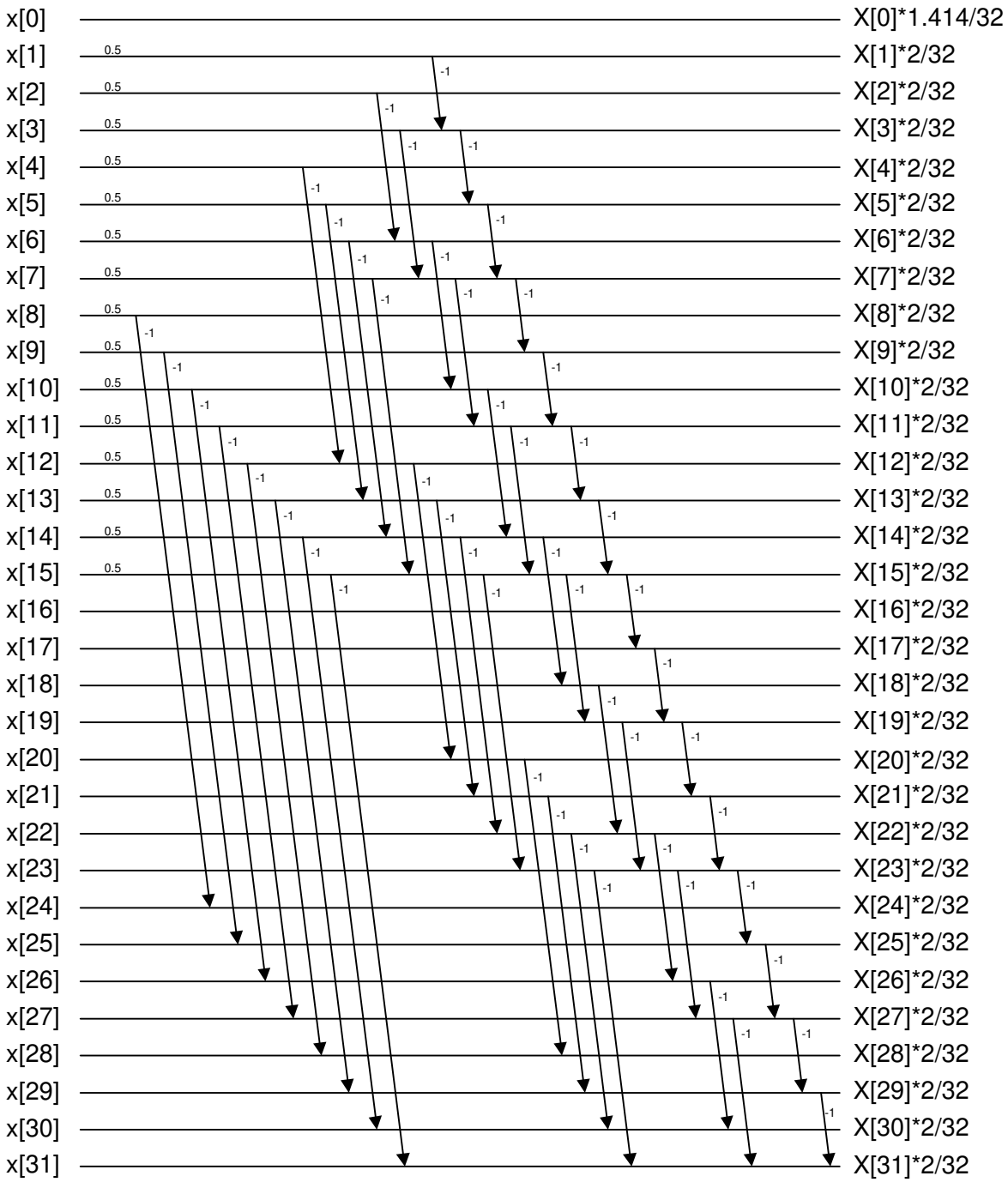


Fig. APB-3 – Continuação

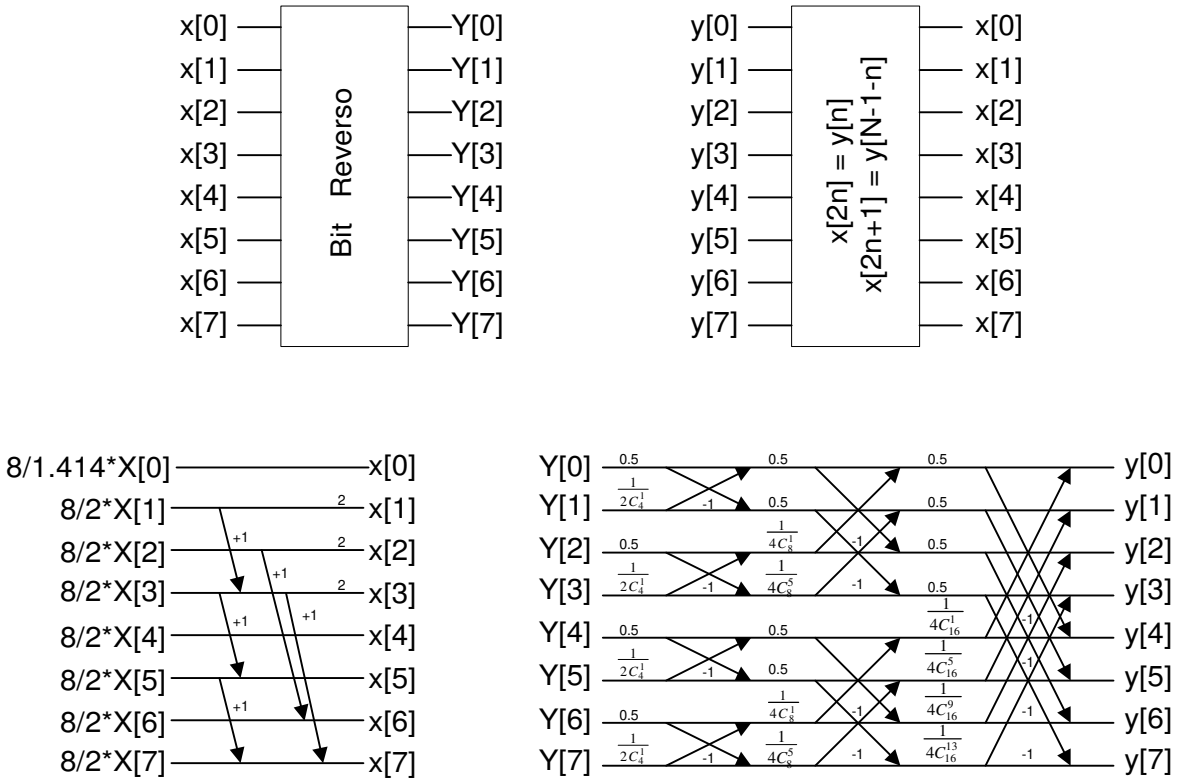


Fig. APB-4 – Fluxo da IFCT de uma sequência **{*x[n]*}** para entradas **N = 8**

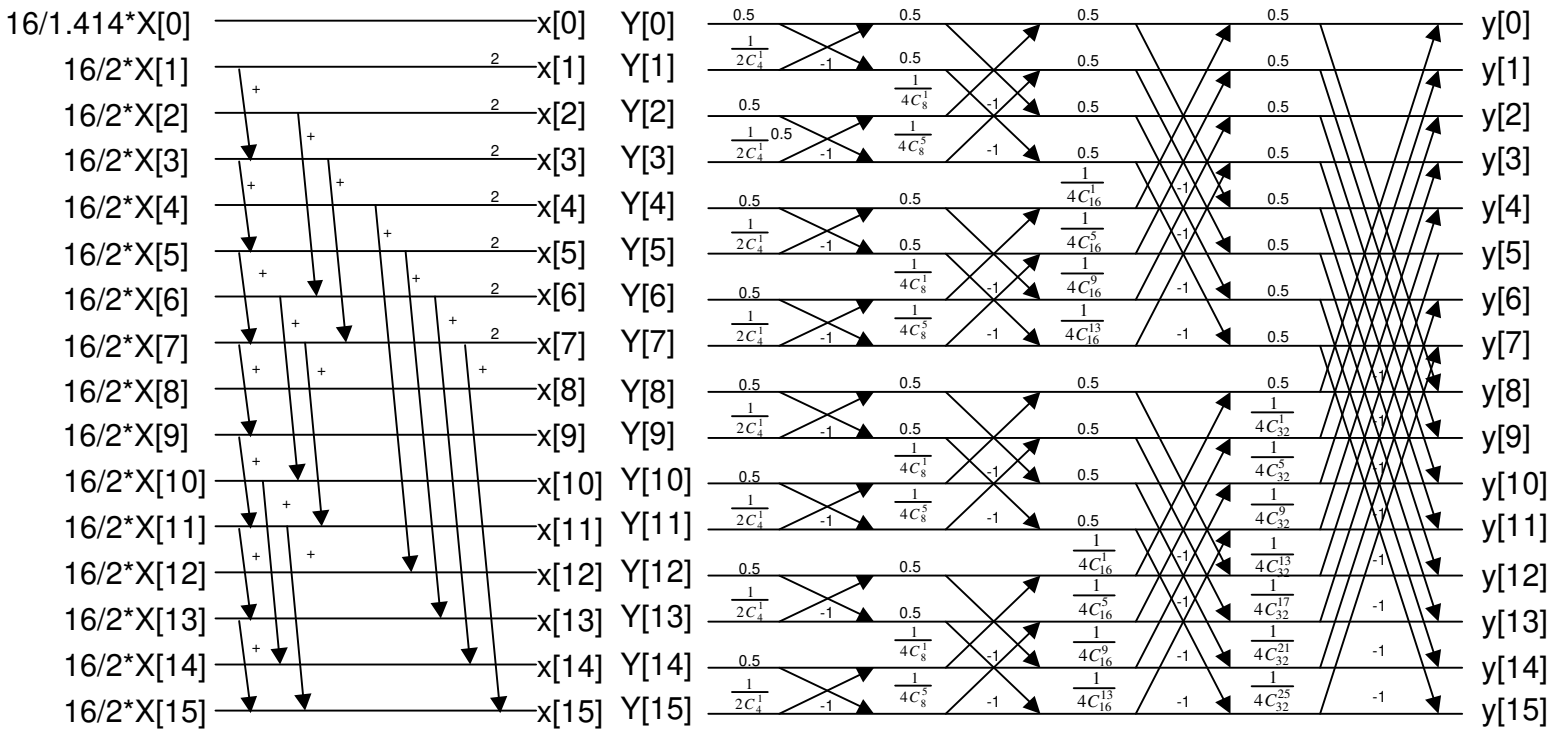


Fig. APB-5 – Fluxo da IFCT de uma sequência ***{x[n]}*** para entradas **N = 16**

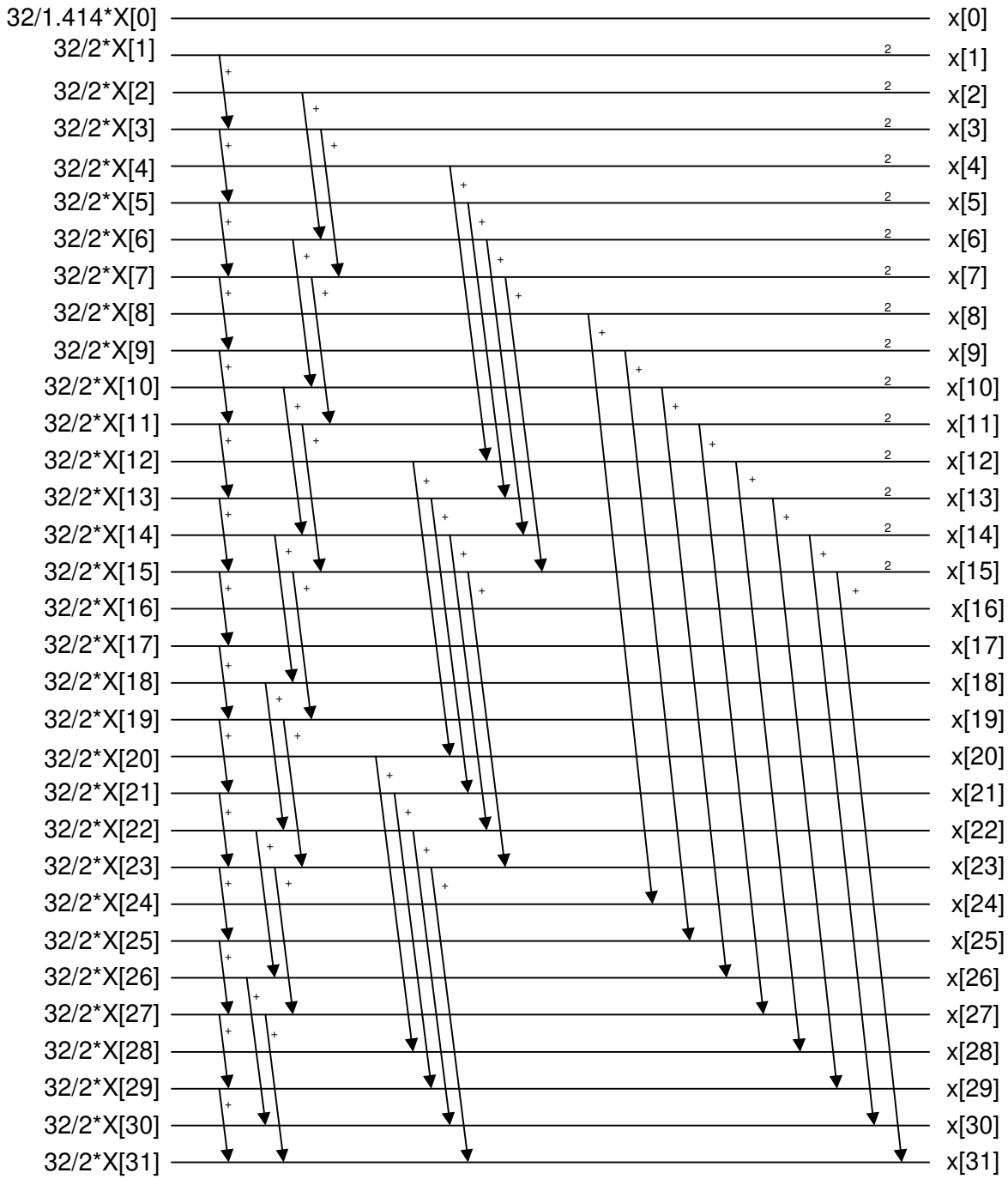


Fig. APB-6 – Fluxo da IFCT de uma sequência **{*x[n]*}** para entradas **N = 32**

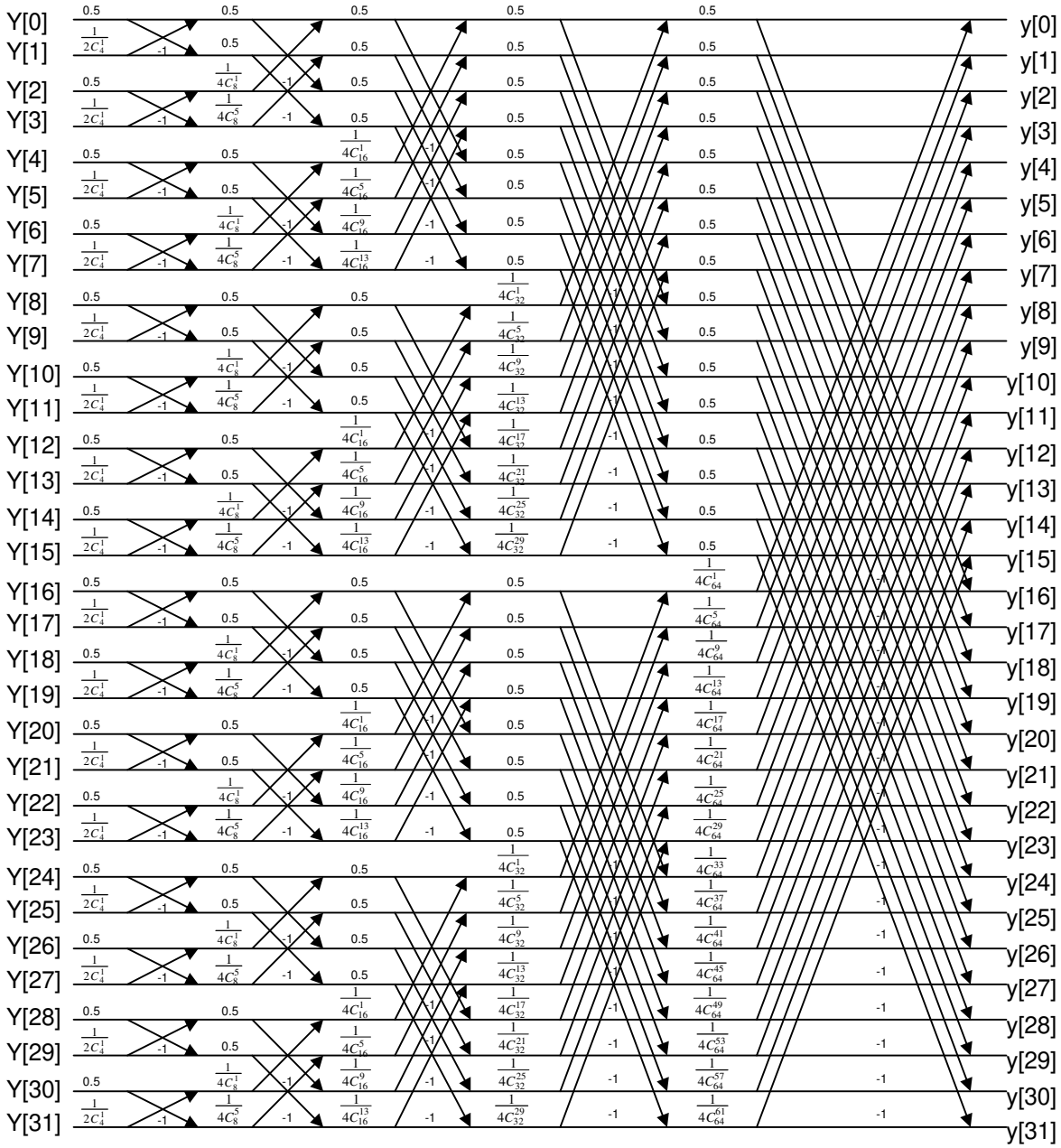


Fig. APB-6 – Continuação

APÊNDICE C

LISTAGEM DOS PROGRAMAS

Para a simulação do sistema proposto, foram desenvolvidos vários programas em linguagem **C**. Todos os programas foram desenvolvidos pelo autor. Os programas foram desenvolvidos em forma de rotinas formando um projeto como um todo. Para a utilização dos programas, os mesmos devem ser compilados e ligados para a geração de um programa executável. Os programas foram desenvolvidos no padrão ANSI (*American National Standard Institute*), podendo portanto ser compilados em qualquer plataforma da linha computadores (PC), e com pequenos ajustes, em qualquer plataforma da linha **UNIX**. Alguns dos programas foram compilados e testados na plataforma **UNIX** e **LINUX** e funcionaram sem nenhuma modificação.

As imagens a serem executadas pelo sistema devem estar no formato RAW (sem formatação) e com tamanhos (linha x coluna) que sejam múltiplos dos tamanhos dos blocos da DCT utilizada.

O pacote de programas está dividido em programas auxiliares e programas principais. Os programas principais são os programas que contém a função *main( )*, e somente esses podem gerar os programas executáveis. Cada programa principal usa apenas alguns programas auxiliares.

Os programas desenvolvidos para a simulação do sistema são listados a seguir.

```
/*********************************************************************
ABRE_ARQ.C  Abre um arquivo para leitura ou escrita
 *********************************************************************
abre_ler        abre um arquivo para leitura
abre_escreve    abre um arquivo para escrita
 *********************************************************************
 *********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "maniparq.h"

/*********************************************************************
 *********************************************************************
abre_ler  abre um arquivo no modo binário para leitura e retorna um
          ponteiro para a estrutura alocada pela função. Alocação de
          erro ou tipo impróprio causa exit(1). Um arquivo com problema
          retorna um ponteiro NULL.

ARQUIVO *abre_ler(char *nome_arq)
*********************************************************************/
ARQUIVO *abre_ler(char *nome_arq)
    {
      ARQUIVO *arquivo;

     arquivo = (ARQUIVO *)malloc(sizeof(ARQUIVO));
     if(!arquivo) {
       printf("\nErro de abertura na alocação de estrutuda do \
              arquivo %s \n",nome_arq);
       exit(1);
      }

     arquivo->ptr_arq = fopen(nome_arq,"rb");
     if(arquivo->ptr_arq==NULL) {
      printf("\nO arquivo e %s não existe \n",nome_arq);
      return(NULL);
      }
     if(!arquivo->ptr_arq) {
      printf("\nErro de abertura de %s em abre_ler() \n",nome_arq);
      return(NULL);
      }

     arquivo->nome = (char *)malloc(strlen(nome_arq) + 1);
     if(!arquivo->nome) {
       printf("\nErro na alocação do nome do arquivo em abre_ler() \n");
       exit(1);
       }
     strcpy(arquivo->nome,nome_arq);

     return(arquivo);
   }
```

```
/*********************************************************************
 *********************************************************************
abre_escreve   abre um arquivo no modo binário para escrita e retorna
           um ponteiro para a estrutura alocada pela função. Alocação
           de erro ou tipo inpróprio causa exit(1). Um arquivo ruim
           retorna um ponteiro NULL.

ARQUIVO *abre_escreve(char *nome_arq)
*********************************************************************/
ARQUIVO *abre_escreve(char *nome_arq)
   {
     ARQUIVO *arquivo;

     arquivo = (ARQUIVO *)malloc(sizeof(ARQUIVO));
     if(!arquivo) {
       printf("\nErro de alocação na abertura da estrura do \
              arquivo %s \n",nome_arq);
       exit(1);
      }

     arquivo->ptr_arq = fopen(nome_arq,"wb");
     if(!arquivo->ptr_arq) {
      printf("\nErro de abertura de %s em abre_escreve() \n",nome_arq);
      return(NULL);
      }

     arquivo->nome = (char *)malloc(strlen(nome_arq) + 1);
     if(!arquivo->nome) {
       printf("\nErro na alocação do nome do arquivo em abre_ler() \n");
       exit(1);
       }
     strcpy(arquivo->nome,nome_arq);

     return(arquivo);
   }
/*********************************************************************/
```

```
/*-------------------------------------------------------------------
ADCOMOI.C  Codificação adaptativa de imagens monocromaticas através da
         determinação da energia AC de cada bloco de coeficientes da
         transformada DCT2D para a determinação das classes dos blocos.
         Descrição da técnica:
      1 - Determinação da energia AC
            EAC = soma[(Xc[uv])^2] - Xc[00]  u,v = 0,.....N
            Xc[uv] = coef. [uv] da DCT
            N = tamanho da coluna do bloco DCT

-------------------------------------------------------------------*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <process.h>
#include <io.h>
#include <conio.h>
#include <math.h>
#include "maniparq.h"
/*******************************************************************/
main()
{
 MATRIX *Ximage,*EAC,*DCTAC,*DEAC,*CLASSE,*SCEAC;
 MATRIX *VAR,*BIJ;
 MATRIX *KK;
 float **kk;
 unsigned int n;
 int i,j,k,l,p,q,N,NN,M,col,coluna,lin,linha,tipo,cl0,cl1,cl2,cl3;
 int NA,coef,**eac,*nbloc,status,MIN,MAX,*HIST,lim1,lim2,lim3;
 long int ecl0,ecl1,ecl2,ecl3;
 int bitcl0,bitcl1,bitcl2,bitcl3;
 float B,**xif,**dctac,**var,*prod,soma,soma2,PROB;
 float sh,sv,sd,fn,fn0,fn1,fn2,fn3;
 char *nome_arq,*arq,arq_temp[14],tam_elemf,tam_elemc,ch;
 char **classe,**deac,**sceac,**bij;
 char *Xac;
 int Xdc;
 ARQUIVO *arq_in;
 FILE *arqdat,*arqhdr,*arqvdc;
/*-------------------------------------------------------------------*/
printf("\nPrograma ADCOMOI para codificação adaptativa de imagens");
printf("\nmonocromaticas através da determinação da energia AC de cada");
printf("\nbloco da transformada DCT2D e da classificação dos blocos");
printf("\nem bandas de energia e cada banda em sub-bandas");
/*-------------------------------------------------------------------*/
do {
    nome_arq = ler_str("nome do arquivo com os coeficientes DCT2D ");
    arq_in = abre_ler(nome_arq);
    printf("\nDimensoes do arquivo:");
    linha = ler_int("numero de linhas",1,512);
    coluna = ler_int("numero de colunas",1,1024);
 } while(!arq_in);
/*-------------------------------------------------------------------*/
```

```
do {
     N = 8;
     n = (int)(log10(N)/log10(2));
     NN = 1 << n;
     if(NN!=N) {
       printf("\nO número de entrada deve ser potencia de 2 ");
       exit(1);
      }
     if((N>coluna)||(N>linha)) {
      printf("\nO Bloco da transformada não pode ser maior que a imagem
");
      exit(1);
      }
     if((linha%N)||(coluna%N)) {
      printf("\nO número de linhas ou colunas não eh multiplo de %d ",N);
      exit(1);
      }
 } while(!N);
/*--------------------------------------------------------------------*/
   NA = N*N;
   M = coluna/N;
   B = (float)((linha/N)*(coluna/N));
/*--------------------------------------------------------------------*/
   nome_arq = ler_str("nome do arquivo de saída SEM EXTENÇÃO: ");
   arq = strchr(nome_arq,'.');
   if(arq != NULL) {
      k = strcspn(nome_arq,".");
      nome_arq[k] = '\0';
     }
  /*-----------------------------------------------------*/
   arq = strcpy(arq_temp,nome_arq);
   arq = strcat(arq_temp,".dat");
   arqdat = fopen(arq,"wb");
  /*-----------------------------------------------------*/
   arq = strcpy(arq_temp,nome_arq);
   arq = strcat(arq_temp,".hdr");
   arqhdr = fopen(arq,"wb");
  /*-----------------------------------------------------*/
   arq = strcpy(arq_temp,nome_arq);
   arq = strcat(arq_temp,".vdc");
   arqvdc = fopen(arq,"wb");
  /*-----------------------------------------------------*/
/*-------------------------------------------------------------*/
    tam_elemc = sizeof(char);
    tam_elemf = sizeof(float);
    Ximage = aloca_mat(N,coluna,tam_elemf);
    xif = (float **)Ximage->ptr_mat;
    EAC = aloca_mat(M,M,sizeof(int));
    eac = (int **)EAC->ptr_mat;
/*--------------------------------------------------------------------*/
/*--------------------------------------------------------------------*/
   for(i = 0; i < M; i++)
       for(j = 0; j < M; j++)
         eac[i][j] = 0;
   MIN = 32767;
   MAX = -32768;
```

```
   for(lin = 0; lin < linha; lin+=N) {
      for(i = 0; i < N; i++)
       ler_registro((char *)Ximage->ptr_mat[i],arq_in,coluna,tam_elemf);
      for(col = 0; col < coluna; col+=N) {
       soma2 = 0.0;
       for(i = 0; i < N; i++) {
          for(j = 0; j < N; j++) {
             if((i==0)&&(j==0)) {
                continue;
             }
             else {
                soma2 += (float)pow((double)xif[i][col+j],2.0);
             }
           }
         }
       eac[lin/N][col/N] = ROUND(soma2);
       if(ROUND(soma2)<MIN)    MIN = ROUND(soma2);
       if(ROUND(soma2)>MAX)    MAX = ROUND(soma2);
       }
    }
   if(MIN != 0)     MAX = ROUND((float)MAX/(float)MIN);
   libera_mat(Ximage);
/*-------------------------------------------------------------------------*/
   HIST = (int *)calloc(MAX+1,sizeof(int));
   if(HIST == NULL) {
      printf("\nNão existe memória suficiente para HISTOGRAMA");
      exit(1);
     }
   for(i = 0; i < M; i++)
      for(j = 0; j < M; j++)  {
         l = (int)((float)eac[i][j]/(float)MIN);
         HIST[ l ]++;
      }
/*-------------------------------------------------------------------------*/
   PROB = 0.0;
   lim1=lim2=lim3=0;
   for(i = MIN/MIN; i <= MAX; i++)   {
       if(HIST[i] == 0)
          continue;
       else  {
          PROB += (float)HIST[i]/B;
          if(PROB <= 0.25)  { lim1 = i; }
          if(PROB > 0.25 && PROB <= 0.50) { lim2 = i;  }
          if(PROB > 0.50 && PROB <= 0.75) { lim3 = i; }
       }
    }
   free(HIST);
/*-------------------------------------------------------------------------*/
   CLASSE = aloca_mat(M,M,tam_elemc);
   classe = (char **)CLASSE->ptr_mat;
   for(i = 0; i < M; i++)
      for(j = 0; j < M; j++) {
         l = (int)((float)eac[i][j]/(float)MIN);
       if(l >= 0 && l <= lim1)    k = 0;
       if(l > lim1 && l <= lim2)  k = 1;
       if(l > lim2 && l <= lim3)  k = 2;
       if(l > lim3)    k = 3;
```

```
       classe[i][j] = k;
       }
/*-------------------------------------------------------------------------*/
   ecl0=ecl1=ecl2=ecl3=0L;
   bitcl0=bitcl1=bitcl2=bitcl3=0;
   cl0=cl1=cl2=cl3=0;
   for(i = 0; i < M; i++)
      for(j = 0; j < M; j++) {
       if(classe[i][j] == 0)  { ecl0 += eac[i][j]; cl0+=1; }
       if(classe[i][j] == 1)  { ecl1 += eac[i][j]; cl1+=1; }
       if(classe[i][j] == 2)  { ecl2 += eac[i][j]; cl2+=1; }
       if(classe[i][j] == 3)  { ecl3 += eac[i][j]; cl3+=1; }
       }
    bitcl0 = (int)pow((double)ecl0,(1.0/3.0));
    bitcl1 = (int)pow((double)ecl1,(1.0/3.0));
    bitcl2 = (int)pow((double)ecl2,(1.0/3.0));
    bitcl3 = (int)pow((double)ecl3,(1.0/3.0));
/*-------------------------------------------------------------------------*/
   libera_mat(EAC);
/*-------------------------------------------------------------------------*/
   rewind(arq_in->ptr_arq);
   Ximage = aloca_mat(N,coluna,tam_elemf);
   xif = (float **)Ximage->ptr_mat;
   DCTAC = aloca_mat(N,N,tam_elemf);
   dctac = (float **)DCTAC->ptr_mat;
   DEAC = aloca_mat(M,M,sizeof(char));
   deac = (char **)DEAC->ptr_mat;
   for(lin = 0; lin < linha; lin+=N) {
      for(i = 0; i < N; i++)
       ler_registro((char *)Ximage->ptr_mat[i],arq_in,coluna,tam_elemf);
      for(col = 0; col < coluna; col+=N) {
       sh=sv=sd=0.0;
       for(i = 0; i < N; i++) {
          for(j = 0; j < N; j++) {
            if((i==0)&&(j==0)) { dctac[i][j] = xif[i][col+j]; }
            else { dctac[i][j] = (float)pow((double)xif[i][col+j],2.0); }
           }
         }
       for(i = 0; i < 3; i++) {
          for(j = 0; j < N; j++) {
             if((i==0)&&(j==0)) {  continue; }
             else {  sh += dctac[i][j];
                   sv += dctac[j][i]; }
           }
         }
       sd = dctac[0][1]+dctac[0][2]+dctac[1][0]+dctac[1][1]+dctac[1][2];
       sd +=dctac[2][0]+dctac[2][1]+dctac[2][2]+dctac[2][3]+dctac[3][2];
       sd +=dctac[3][3]+dctac[3][4]+dctac[4][3]+dctac[4][4]+dctac[4][5];
       sd +=dctac[5][4]+dctac[5][5]+dctac[5][6]+dctac[6][5]+dctac[6][6];
       sd +=dctac[6][7]+dctac[7][6]+dctac[7][7];
       if(sh > sv && sh > sd) { deac[lin/N][col/N] = 0; }
       if(sv > sh && sv > sd) { deac[lin/N][col/N] = 1; }

       if(sd > sh && sd > sv) { deac[lin/N][col/N] = 2; }
       if(sh==sv  && sv==sd)  { deac[lin/N][col/N] = 0; }
       }
    }
```

```
   libera_mat(Ximage);
   libera_mat(DCTAC);
/*-------------------------------------------------------------------*/
   nbloc = (int *)calloc(3*4,sizeof(int));
   SCEAC = aloca_mat(M,M,tam_elemc);
   sceac = (char **)SCEAC->ptr_mat;
   for(i = 0; i < M; i++)
      for(j = 0; j < M; j++) {
       if(deac[i][j] == 0) { if(classe[i][j] == 0 ) sceac[i][j] = 0;
                             if(classe[i][j] == 1 ) sceac[i][j] = 1;
                             if(classe[i][j] == 2 ) sceac[i][j] = 2;
                             if(classe[i][j] == 3 ) sceac[i][j] = 3;
           }
       if(deac[i][j] == 1) { if(classe[i][j] == 0 ) sceac[i][j] = 4;
                             if(classe[i][j] == 1 ) sceac[i][j] = 5;
                             if(classe[i][j] == 2 ) sceac[i][j] = 6;
                             if(classe[i][j] == 3 ) sceac[i][j] = 7;
           }
       if(deac[i][j] == 2) { if(classe[i][j] == 0 ) sceac[i][j] = 8;
                             if(classe[i][j] == 1 ) sceac[i][j] = 9;
                             if(classe[i][j] == 2 ) sceac[i][j] = 10;
                             if(classe[i][j] == 3 ) sceac[i][j] = 11;
           }
       nbloc[ sceac[i][j] ] += 1;
       }
   libera_mat(DEAC);
   libera_mat(CLASSE);
/*-------------------------------------------------------------------*/
   VAR = aloca_mat(12,NA,tam_elemf);
   var = (float **)VAR->ptr_mat;
   Ximage = aloca_mat(N,coluna,tam_elemf);
   xif = (float **)Ximage->ptr_mat;
   rewind(arq_in->ptr_arq);
   for(i = 0; i < 12; i++) for(j = 0; j < NA; j++)  var[i][j] = 0.0;
   for(lin = 0; lin < linha; lin+=N) {
      k = (int)(lin/N);
      for(i = 0; i < N; i++)
        ler_registro((char *)Ximage->ptr_mat[i],arq_in,coluna,tam_elemf);
      for(col = 0; col < coluna; col+=N) {
        l = (int)(col/N);
        p = sceac[k][l];
        q = 0;
        for(i = 0; i < N; i++) {
            for(j = 0; j < N; j++) {
              var[p][q] += (float)pow((double)xif[i][col+j],2.0);
              q++;
             }
         }
       }
    }
   libera_mat(Ximage);

   for(i = 0; i < 12; i++) {
       for(j = 0; j < NA; j++) {
         var[i][j] /= (float)nbloc[i];
      }
    }
```

```
  free(nbloc);

/*-------------------------------------------------------------------------*/
   prod = (float *)calloc(3*4,sizeof(float));
   for(i = 0; i < 12; i++)
       prod[i] = 1.0;
   for(i = 0; i < 12; i++) {
       for(j = 1; j < NA; j++) {
         prod[i] *= (float)pow(var[i][j],(1.0/(float)(NA-1)));
      }
    }
/*-------------------------------------------------------------------------*/
   BIJ = aloca_mat(12,NA,sizeof(char));
   bij = (char **)BIJ->ptr_mat;
   {
     float bits,bpp;
     for(i = 0; i < 12; i++) {
       bij[i][0] = 8;
       if(i==0 || i==4 || i==8)  bpp = bitcl0;
       if(i==1 || i==5 || i==9)  bpp = bitcl1;
       if(i==2 || i==6 || i==10) bpp = bitcl2;
       if(i==3 || i==7 || i==11) bpp = bitcl3;
       bpp /=(float)(NA-1);
       for(j = 1; j < NA; j++) {
          bits = bpp + 0.5*log10((double)(var[i][j]/prod[i]))/log10(2.0);
          if(bits>8.0)  bits = 8.0;
          bij[i][j] = (char)((bits>0)?ROUND(bits):0);
        }
      }
    }
   for(i = 0; i < 12; i++)  {
      for(j = 0; j < NA; j++)  {
        printf("%d ",(int)bij[i][j]);
        if(j!=0 && (j+1)%8==0)
           printf("\n");
      }
     printf("\n");
     getch();
    }
  free(prod);
/*-------------------------------------------------------------------------*/
  fn0=fn1=fn2=fn3= -32768.0;
  for(i = 0; i < 12; i++) {
     for(j = 1; j < NA; j++) {
      if(bij[i][j] == 1) {
          fn = var[i][j];
          if((i==0 || i==4 || i==8) && (fn0 < fn))   fn0 = fn;
          if((i==1 || i==5 || i==9) && (fn1 < fn))   fn1 = fn;
          if((i==2 || i==6 || i==10) && (fn2 < fn))  fn2 = fn;

          if((i==3 || i==7 || i==11) && (fn3 < fn))  fn3 = fn;
       }
      }
   }
   fn0 = (float)sqrt(fn0);   fn1 = (float)sqrt(fn1);
   fn2 = (float)sqrt(fn2);   fn3 = (float)sqrt(fn3);
   libera_mat(VAR);
```

```
   libera_mat(BIJ);
/*-------------------------------------------------------------------*/

   fwrite(&linha,sizeof(int),1,arqhdr);
   fwrite(&coluna,sizeof(int),1,arqhdr);
   for(i = 0; i < M; i++) {
       status = fwrite((char *)SCEAC->ptr_mat[i],sizeof(char),M,arqhdr);
       if(status != M) {
        printf("\nErro na gravação de registro em %s \n",arqhdr);
        exit(1);
      }
    }
   fwrite(&fn0,sizeof(float),1,arqhdr);
   fwrite(&fn1,sizeof(float),1,arqhdr);
   fwrite(&fn2,sizeof(float),1,arqhdr);
   fwrite(&fn3,sizeof(float),1,arqhdr);
   fclose(arqhdr);
/*-------------------------------------------------------------------*/
   Ximage = aloca_mat(N,coluna,tam_elemf);
   xif = (float **)Ximage->ptr_mat;
   Xac = (char *)calloc(NA-1,tam_elemc);
   KK = aloca_mat(N,coluna,tam_elemf);
   kk = (float **)KK->ptr_mat;
   rewind(arq_in->ptr_arq);
   for(lin = 0; lin < linha; lin+=N) {
      k = (int)(lin/N);
      for(i = 0; i < N; i++)
        ler_registro((char *)Ximage->ptr_mat[i],arq_in,coluna,tam_elemf);
      for(col = 0; col < coluna; col+=N) {
        l = (int)(col/N);
        p = sceac[k][l];
        if(p==0 || p==4 || p==8)   fn = fn0;
        if(p==1 || p==5 || p==9)   fn = fn1;
        if(p==2 || p==6 || p==10)  fn = fn2;
        if(p==3 || p==7 || p==11)  fn = fn3;
        q = 0;
        for(i = 0; i < N; i++) {
            for(j = 0; j < N; j++) {
              if(i == 0 && j == 0) {
                  Xdc = ROUND(xif[i][col+j]/2.0);
                      kk[i][col+j] = (float)Xdc;
                }
              else {
                  Xac[q] = (char)ROUND(xif[i][col+j]/fn);
                      kk[i][col+j] = (float)Xac[q];
                  q++;
               }
             }
          }
         if(putc(Xdc,arqvdc) == EOF) {
           printf("\nErro na gravação de dados em %s \n",arqvdc);
           exit(1);
          }
         status = fwrite((char *)Xac,tam_elemc,NA-1,arqdat);
         if(status != NA-1) {
           printf("\nErro na gravação de registro em %s \n",arqdat);
           exit(1);
```

```
            }
        }


     }
   libera_mat(SCEAC);
   libera_mat(Ximage);
/*-------------------------------------------------------------------------*/

fcloseall();
}
/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/
```

```
/*-------------------------------------------------------------------
ADCOMI1.C  Codificação adaptativa de imagens monocromaticas através da
 determinação da energia AC de cada bloco de coeficientes da transformada
 DCT2D para a determinação das classes dos blocos.
 Descrição da técnica:
      1 - Determinação da energia AC
            EAC = soma[(Xc[uv])^2] - Xc[00]  u,v = 0,.....N
            Xc[uv] = coef. [uv] da DCT
            N = tamanho da coluna do bloco DCT

-------------------------------------------------------------------*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <process.h>
#include <io.h>
#include <conio.h>
#include <math.h>
#include "maniparq.h"
/*******************************************************************/
main()
{
 MATRIX *Ximage,*EAC,*DCTAC,*DEAC,*CLASSE,*SCEAC;
 MATRIX *VAR,*BIJ;
 MATRIX *KK;
 float **kk;
 unsigned int n;
 int i,j,k,l,p,q,N,NN,Ml,Mc,col,coluna,lin,linha,tipo,cl0,cl1,cl2,cl3;
 int NA,coef,**eac,*nbloc,status,MIN,MAX,*HIST,lim1,lim2,lim3;
 long int ecl0,ecl1,ecl2,ecl3;
 int bitcl0,bitcl1,bitcl2,bitcl3;
 float B,**xif,**dctac,**var,*prod,soma,soma2,PROB;
 float sh,sv,sd,fn,fn0,fn1,fn2,fn3,fn4,fn5,fn6,fn7,fn8,fn9,fn10,fn11;
 char *nome_arq,*arq,arq_temp[14],tam_elemf,tam_elemc,ch;
 char **classe,**deac,**sceac,**bij;
 char *Xac,*Yac,sig;
 int Xdc;
 ARQUIVO *arq_in;
 FILE *arqdat,*arqhdr,*arqvdc;
 int zigzag[64] = { 0, 1, 8,16, 9, 2, 3,10,
               17,24,32,25,18,11, 4, 5,
               12,19,26,33,40,48,41,34,
               27,20,13, 6, 7,14,21,28,
               35,42,49,56,57,50,43,36,
               29,22,15,23,30,37,44,51,
               58,59,52,45,38,31,39,46,
               53,60,61,54,47,62,55,63 };
int vertical[64] = {0, 8,16,24,32,40,48,56,
                1, 9,17,25,33,41,49,57,
                2,10,18,26,34,42,50,58,
                3,11,19,27,35,43,51,59,
                4,12,20,28,36,44,52,60,
                5,13,21,29,37,45,53,61,
                6,14,22,30,38,46,54,62,
                7,15,23,31,39,47,55,63 };
```

```
/*--------------------------------------------------------------------*/
printf("\nPrograma ADCOMI1 para codificação adaptativa de imagens");
printf("\nmonocromaticas através da determinação da energia AC de cada");
printf("\nbloco da transformada DCT2D e da classificação dos blocos");
printf("\nem bandas de energia e cada banda em sub-bandas");
/*--------------------------------------------------------------------*/
do {
    nome_arq = ler_str("nome do arquivo com os coeficientes DCT2D ");
    arq_in = abre_ler(nome_arq);
    printf("\nDimensoes do arquivo:");
    linha = ler_int("número de linhas",1,512);
    coluna = ler_int("número de colunas",1,1024);
 } while(!arq_in);
/*--------------------------------------------------------------------*/
do {
     N = 8;
     n = (int)(log10(N)/log10(2));
     NN = 1 << n;
     if(NN!=N) {
       printf("\nO número de entrada deve ser potencia de 2 ");
       exit(1);
      }
     if((N>coluna)||(N>linha)) {
      printf("\nO Bloco da transformada não pode ser maior que a imagem
");
      exit(1);
      }
     if((linha%N)||(coluna%N)) {
      printf("\nO número de linhas ou colunas não eh mutiplo de %d ",N);
      exit(1);
      }
 } while(!N);
/*--------------------------------------------------------------------*/
   NA = N*N;
   Mc = coluna/N;
   Ml = linha/N;
   B = (float)((linha/N)*(coluna/N));
/*--------------------------------------------------------------------*/
   printf("\nGERA ARQUIVO DE SAÍDA, COM A ENERGIA DE CADA BLOCO");
   nome_arq = ler_str("nome do arquivo de saída SEM EXTENÇÃO: ");
   arq = strchr(nome_arq,'.');
   if(arq != NULL) {
      k = strcspn(nome_arq,".");
      nome_arq[k] = '\0';
     }
  /*-----------------------------------------------------*/
   arq = strcpy(arq_temp,nome_arq);
   arq = strcat(arq_temp,".dat");
   arqdat = fopen(arq,"wb");
  /*-----------------------------------------------------*/
   arq = strcpy(arq_temp,nome_arq);
   arq = strcat(arq_temp,".hdr");
   arqhdr = fopen(arq,"wb");

  /*-----------------------------------------------------*/
   arq = strcpy(arq_temp,nome_arq);
   arq = strcat(arq_temp,".vdc");
```

```
   arqvdc = fopen(arq,"wb");
  /*-------------------------------------------------------*/
/*-----------------------------------------------------------------------*/
    tam_elemc = sizeof(char);
    tam_elemf = sizeof(float);
    Ximage = aloca_mat(N,coluna,tam_elemf);
    xif = (float **)Ximage->ptr_mat;
    EAC = aloca_mat(Ml,Mc,sizeof(int));
    eac = (int **)EAC->ptr_mat;
/*-----------------------------------------------------------------------*/
   for(i = 0; i < Ml; i++)
       for(j = 0; j < Mc; j++)
         eac[i][j] = 0;
   MIN = 32767;
   MAX = -32768;
   for(lin = 0; lin < linha; lin+=N) {
      for(i = 0; i < N; i++)
        ler_registro((char *)Ximage->ptr_mat[i],arq_in,coluna,tam_elemf);
      for(col = 0; col < coluna; col+=N) {
       soma2 = 0.0;
       for(i = 0; i < N; i++) {
          for(j = 0; j < N; j++) {
             if((i==0)&&(j==0)) {
                continue;
             }
             else {
                soma2 += (float)pow((double)xif[i][col+j],2.0);
             }
           }
         }
       eac[lin/N][col/N] = ROUND(soma2);
       if(ROUND(soma2)<MIN)   MIN = ROUND(soma2);
       if(ROUND(soma2)>MAX)   MAX = ROUND(soma2);
       }
    }
   if(MIN != 0)    MAX = ROUND((float)MAX/(float)MIN);
   libera_mat(Ximage);
/*-----------------------------------------------------------------------*/
   HIST = (int *)calloc(MAX+1,sizeof(int));
   if(HIST == NULL) {
      printf("\nNão existe memória suficiente para HISTOGRAMA");
      exit(1);
     }
   for(i = 0; i < Ml; i++)
      for(j = 0; j < Mc; j++)  {
         l = (int)((float)eac[i][j]/(float)MIN);
         HIST[ l ]++;
      }
/*-----------------------------------------------------------------------*/
   PROB = 0.0;
   lim1=lim2=lim3=0;
   for(i = MIN/MIN; i <= MAX; i++)   {

       if(HIST[i] == 0)
          continue;
       else  {
          PROB += (float)HIST[i]/B;
```

```
          if(PROB <= 0.25)  { lim1 = i; }
          if(PROB > 0.25 && PROB <= 0.50) { lim2 = i;  }
          if(PROB > 0.50 && PROB <= 0.75) { lim3 = i; }
       }
     }
   free(HIST);
/*-------------------------------------------------------------------------*/
   CLASSE = aloca_mat(Ml,Mc,tam_elemc);
   classe = (char **)CLASSE->ptr_mat;
   for(i = 0; i < Ml; i++)
      for(j = 0; j < Mc; j++) {
         l = (int)((float)eac[i][j]/(float)MIN);
       if(l >= 0 && l <= lim1)    k = 0;
       if(l > lim1 && l <= lim2)  k = 1;
       if(l > lim2 && l <= lim3)  k = 2;
       if(l > lim3)    k = 3;
       classe[i][j] = k;
       }
/*-------------------------------------------------------------------------*/
   ecl0=ecl1=ecl2=ecl3=0L;
   bitcl0=bitcl1=bitcl2=bitcl3=0;
   cl0=cl1=cl2=cl3=0;
   for(i = 0; i < Ml; i++)
      for(j = 0; j < Mc; j++) {
       if(classe[i][j] == 0)  { ecl0 += eac[i][j]; cl0+=1; }
       if(classe[i][j] == 1)  { ecl1 += eac[i][j]; cl1+=1; }
       if(classe[i][j] == 2)  { ecl2 += eac[i][j]; cl2+=1; }
       if(classe[i][j] == 3)  { ecl3 += eac[i][j]; cl3+=1; }
       }
    bitcl0 = (int)pow((double)ecl0,(1.0/3.0));
    bitcl1 = (int)pow((double)ecl1,(1.0/3.0));
    bitcl2 = (int)pow((double)ecl2,(1.0/3.0));
    bitcl3 = (int)pow((double)ecl3,(1.0/3.0));
/*-------------------------------------------------------------------------*/
   libera_mat(EAC);
/*-------------------------------------------------------------------------*/
   rewind(arq_in->ptr_arq);
   Ximage = aloca_mat(N,coluna,tam_elemf);
   xif = (float **)Ximage->ptr_mat;
   DCTAC = aloca_mat(N,N,tam_elemf);
   dctac = (float **)DCTAC->ptr_mat;
   DEAC = aloca_mat(Ml,Mc,sizeof(char));
   deac = (char **)DEAC->ptr_mat;
   for(lin = 0; lin < linha; lin+=N) {
      for(i = 0; i < N; i++)
        ler_registro((char *)Ximage->ptr_mat[i],arq_in,coluna,tam_elemf);
      for(col = 0; col < coluna; col+=N) {
       sh=sv=sd=0.0;
        for(i = 0; i < N; i++) {
          for(j = 0; j < N; j++) {
             if((i==0)&&(j==0)) { dctac[i][j] = xif[i][col+j]; }

             else { dctac[i][j] = (float)pow((double)xif[i][col+j],2.0);
}
           }
         }
       for(i = 0; i < 3; i++) {
```

```
          for(j = 0; j < N; j++) {
             if((i==0)&&(j==0)) {  continue; }
             else {  sh += dctac[i][j];
                   sv += dctac[j][i]; }
           }
         }
       sd = dctac[0][1]+dctac[0][2]+dctac[1][0]+dctac[1][1]+dctac[1][2];
       sd +=dctac[2][0]+dctac[2][1]+dctac[2][2]+dctac[2][3]+dctac[3][2];
       sd +=dctac[3][3]+dctac[3][4]+dctac[4][3]+dctac[4][4]+dctac[4][5];
       sd +=dctac[5][4]+dctac[5][5]+dctac[5][6]+dctac[6][5]+dctac[6][6];
       sd +=dctac[6][7]+dctac[7][6]+dctac[7][7];
       if(sh > sv && sh > sd) { deac[lin/N][col/N] = 0; }
       if(sv > sh && sv > sd) { deac[lin/N][col/N] = 1; }
       if(sd > sh && sd > sv) { deac[lin/N][col/N] = 2; }
       if(sh==sv  && sv==sd)  { deac[lin/N][col/N] = 0; }
       }
     }
   libera_mat(Ximage);
   libera_mat(DCTAC);
/*-------------------------------------------------------------------*/
   nbloc = (int *)calloc(3*4,sizeof(int));
   SCEAC = aloca_mat(Ml,Mc,tam_elemc);
   sceac = (char **)SCEAC->ptr_mat;
   for(i = 0; i < Ml; i++)
      for(j = 0; j < Mc; j++) {
       if(deac[i][j] == 0) { if(classe[i][j] == 0 ) sceac[i][j] = 0;
                             if(classe[i][j] == 1 ) sceac[i][j] = 1;
                             if(classe[i][j] == 2 ) sceac[i][j] = 2;
                             if(classe[i][j] == 3 ) sceac[i][j] = 3;
           }
       if(deac[i][j] == 1) { if(classe[i][j] == 0 ) sceac[i][j] = 4;
                             if(classe[i][j] == 1 ) sceac[i][j] = 5;
                             if(classe[i][j] == 2 ) sceac[i][j] = 6;
                             if(classe[i][j] == 3 ) sceac[i][j] = 7;
           }
       if(deac[i][j] == 2) { if(classe[i][j] == 0 ) sceac[i][j] = 8;
                             if(classe[i][j] == 1 ) sceac[i][j] = 9;
                             if(classe[i][j] == 2 ) sceac[i][j] = 10;
                             if(classe[i][j] == 3 ) sceac[i][j] = 11;
           }
       nbloc[ sceac[i][j] ] += 1;
       }
   libera_mat(DEAC);
   libera_mat(CLASSE);
/*-------------------------------------------------------------------*/
   VAR = aloca_mat(12,NA,tam_elemf);
   var = (float **)VAR->ptr_mat;
   Ximage = aloca_mat(N,coluna,tam_elemf);
   xif = (float **)Ximage->ptr_mat;
   rewind(arq_in->ptr_arq);

   for(i = 0; i < 12; i++) for(j = 0; j < NA; j++)  var[i][j] = 0.0;
   for(lin = 0; lin < linha; lin+=N) {
      k = (int)(lin/N);
      for(i = 0; i < N; i++)
        ler_registro((char *)Ximage->ptr_mat[i],arq_in,coluna,tam_elemf);
      for(col = 0; col < coluna; col+=N) {
```

```
        l = (int)(col/N);
        p = sceac[k][l];
        q = 0;
        for(i = 0; i < N; i++) {
            for(j = 0; j < N; j++) {
              var[p][q] += (float)pow((double)xif[i][col+j],2.0);
              q++;
             }
          }
       }
    }
   libera_mat(Ximage);
   for(i = 0; i < 12; i++) {
       for(j = 0; j < NA; j++) {
         var[i][j] /= (float)nbloc[i];
      }
   }
  free(nbloc);
/*-------------------------------------------------------------------------*/
   prod = (float *)calloc(3*4,sizeof(float));
   for(i = 0; i < 12; i++)
       prod[i] = 1.0;
   for(i = 0; i < 12; i++) {
       for(j = 1; j < NA; j++) {
         prod[i] *= (float)pow(var[i][j],(1.0/(float)(NA-1)));
      }
    }
/*-------------------------------------------------------------------------*/
   BIJ = aloca_mat(12,NA,sizeof(char));
   bij = (char **)BIJ->ptr_mat;
   {
     float bits,bpp;
     for(i = 0; i < 12; i++) {
       bij[i][0] = 8;
       if(i==0 || i==4 || i==8)  bpp = bitcl0;
       if(i==1 || i==5 || i==9)  bpp = bitcl1;
       if(i==2 || i==6 || i==10) bpp = bitcl2;
       if(i==3 || i==7 || i==11) bpp = bitcl3;
       bpp /=(float)(NA-1);
       for(j = 1; j < NA; j++) {
          bits = bpp + 0.5*log10((double)(var[i][j]/prod[i]))/log10(2.0);
          if(bits>8.0)  bits = 8.0;
          bij[i][j] = (char)((bits>0)?ROUND(bits):0);
        }
      }
    }
  free(prod);
/*-------------------------------------------------------------------------*/
  fn0=fn1=fn2=fn3=fn4=fn5= -32768.0;

  fn6=fn7=fn8=fn9=fn10=fn11= -32768.0;
  for(i = 0; i < 12; i++) {
     for(j = 1; j < NA; j++) {
      if(bij[i][j] == 1) {
          fn = var[i][j];
          if((i==0)&&(fn0 < fn))    fn0 = fn;
          if((i==1)&&(fn1 < fn))    fn1 = fn;
```

```
          if((i==2)&&(fn2 < fn))    fn2 = fn;
          if((i==3)&&(fn3 < fn))    fn3 = fn;
          if((i==4)&&(fn4 < fn))    fn4 = fn;
          if((i==5)&&(fn5 < fn))    fn5 = fn;
          if((i==6)&&(fn6 < fn))    fn6 = fn;
          if((i==7)&&(fn7 < fn))    fn7 = fn;
          if((i==8)&&(fn8 < fn))    fn8 = fn;
          if((i==9)&&(fn9 < fn))    fn9 = fn;
          if((i==10)&&(fn10 < fn))  fn10 = fn;
          if((i==11)&&(fn11 < fn))  fn11 = fn;
       }
      }
   }
  fn0=(float)sqrt(fn0); fn1=(float)sqrt(fn1); fn2=(float)sqrt(fn2);
  fn3=(float)sqrt(fn3); fn4=(float)sqrt(fn4); fn5=(float)sqrt(fn5);
  fn6=(float)sqrt(fn6); fn7=(float)sqrt(fn7); fn8=(float)sqrt(fn8);
  fn9=(float)sqrt(fn9); fn10=(float)sqrt(fn10); fn11=(float)sqrt(fn11);
  libera_mat(VAR);
  libera_mat(BIJ);
/*-------------------------------------------------------------------*/
   fwrite(&linha,sizeof(int),1,arqhdr);
   fwrite(&coluna,sizeof(int),1,arqhdr);
   for(i = 0; i < Ml; i++) {
       status = fwrite((char *)SCEAC->ptr_mat[i],sizeof(char),Mc,arqhdr);
       if(status != Mc) {
        printf("\nErro na gravação de registro em %s \n",arqhdr);
        exit(1);
      }
    }
   fwrite(&fn0,sizeof(float),1,arqhdr);
   fwrite(&fn1,sizeof(float),1,arqhdr);
   fwrite(&fn2,sizeof(float),1,arqhdr);
   fwrite(&fn3,sizeof(float),1,arqhdr);
   fwrite(&fn4,sizeof(float),1,arqhdr);
   fwrite(&fn5,sizeof(float),1,arqhdr);
   fwrite(&fn6,sizeof(float),1,arqhdr);
   fwrite(&fn7,sizeof(float),1,arqhdr);
   fwrite(&fn8,sizeof(float),1,arqhdr);
   fwrite(&fn9,sizeof(float),1,arqhdr);
   fwrite(&fn10,sizeof(float),1,arqhdr);
   fwrite(&fn11,sizeof(float),1,arqhdr);
   fclose(arqhdr);
/*-------------------------------------------------------------------*/
   Ximage = aloca_mat(N,coluna,tam_elemf);
   xif = (float **)Ximage->ptr_mat;
   Xac = (char *)calloc(NA-1,tam_elemc);
   Yac = (char *)calloc(NA-1,tam_elemc);
   KK = aloca_mat(N,coluna,tam_elemf);

   kk = (float **)KK->ptr_mat;
   rewind(arq_in->ptr_arq);
   for(lin = 0; lin < linha; lin+=N) {
      k = (int)(lin/N);
      for(i = 0; i < N; i++)
        ler_registro((char *)Ximage->ptr_mat[i],arq_in,coluna,tam_elemf);
      for(col = 0; col < coluna; col+=N) {
        l = (int)(col/N);
```

```
        p = sceac[k][l];
        if(p==0)  fn = fn0;    if(p==1)   fn=fn1;
        if(p==2)  fn = fn2;    if(p==3)   fn=fn3;
        if(p==4)  fn = fn4;    if(p==5)   fn=fn5;
        if(p==6)  fn = fn6;    if(p==7)   fn=fn7;
        if(p==8)  fn = fn8;    if(p==9)   fn=fn9;
        if(p==10) fn = fn10;   if(p==11)  fn=fn11;
        q = 0;
        for(i = 0; i < N; i++) {
            for(j = 0; j < N; j++) {
              if(i == 0 && j == 0) {
                  Xdc = ROUND(xif[i][col+j]/2.0);
                  kk[i][col+j] = (float)Xdc;
                 }
              else {
                  Xac[q] = (char)ROUND(xif[i][col+j]/fn);
                  kk[i][col+j] = (float)Xac[q];
                  q++;
               }
             }
          }
        for(q = 0; q < NA-1; q++) {
            if(p==0 || p==1 || p==2 || p==3)
               Yac[q] = Xac[q];
            if(p==4 || p==5 || p==6 || p==7)
               Yac[q] = Xac[ vertical[q] ];
            if(p==8 || p==9 || p==10 || p==11)
               Yac[q] = Xac[ zigzag[q] ];
          }
           if(putc(Xdc,arqvdc) == EOF) {
             printf("\nErro na gravação de dados em %s \n",arqvdc);
             exit(1);
            }
           status = fwrite((char *)Yac,tam_elemc,NA-1,arqdat);
           if(status != NA-1) {
             printf("\nErro na gravação de registro em %s \n",arqdat);
             exit(1);
            }
       }
    }
   libera_mat(SCEAC);
   libera_mat(Ximage);
/*-------------------------------------------------------------------*/
fcloseall();
}
/*-------------------------------------------------------------------*/
/*-------------------------------------------------------------------*/
```

```
/*****************************************************************
ALOC_MAT  aloca e libera espaço de memória para matriz
 *****************************************************************
aloca_mat    aloca espaço para matriz, retornando um ponteiro para
          a estrutura MATRIX
libera_mat   libera o espaço de memória e o ponteiro da estrutura
          MATRIX
*****************************************************************
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "maniparq.h"
/*****************************************************************
 *****************************************************************
aloca_mat  Aloca espaço para uma matriz de tamanho (linha x coluna)
         e tipo dado pelo tamanho do elemento, podendo ser do tipo
         int, float ou double (2, 4 ou 8 bytes). Provoca exit(1)
         caso não possa ser alocado espaço para a mesma.

MATRIX *aloca_mat(unsigned int linha,unsigned int coluna,int tamanho)
*****************************************************************/
MATRIX *aloca_mat(unsigned int linha,unsigned int coluna,int tamanho)
      {
       int i;
       MATRIX *AA;
       AA = (MATRIX *)calloc(1,sizeof(MATRIX));
       if(!AA) {
           printf("\nErro na alocação de espaço para a estrutura \n");
           exit(1);
         }

       AA->linha = linha;
       AA->coluna = coluna;
       AA->tamanho = tamanho;

      switch(tamanho) {
            case sizeof(unsigned char): {
              unsigned char **char_mat;
      char_mat = (unsigned char **)calloc(linha,sizeof(unsigned char *));

              if(!char_mat) {
                 printf("\nErro na alocação de ponteiros de linhas \
                       na matriz %dx%d \n",linha,coluna);
                 exit(1);
               }
       for(i = 0; i < linha; i++) {
          char_mat[i] = (unsigned char *)calloc(coluna,sizeof(unsigned \\
                        char));
          if(!char_mat[i]) {
              printf("\nErro na alocação de ponteiros de \
                 colunas em %d na matrix %dx%d \n",i,linha,coluna);
               exit(1);
             }
          }
         AA->ptr_mat = (char **)char_mat;
         break;
```

```
        }
      case sizeof(short): {
        short **int_mat;
        int_mat = (short **)calloc(linha,sizeof(short *));
        if(!int_mat) {
           printf("\nErro na alocação de ponteiros de linhas \
                 na matriz %dx%d \n",linha,coluna);
           exit(1);
         }
        for(i = 0; i < linha; i++) {
           int_mat[i] = (short *)calloc(coluna,sizeof(short));
           if(!int_mat[i]) {
            printf("\nErro na alocação de ponteiros de \
              colunas em %d na matrix %dx%d \n",i,linha,coluna);
            exit(1);
            }
         }
        AA->ptr_mat = (char **)int_mat;
        break;
       }
      case sizeof(float): {
        float **float_mat;
        float_mat = (float **)calloc(linha,sizeof(float *));
        if(!float_mat) {
           printf("\nErro na alocação de ponteiros de linhas \
                 na matriz %dx%d \n",linha,coluna);
           exit(1);
          }
        for(i = 0; i < linha; i++) {
           float_mat[i] = (float *)calloc(coluna,sizeof(float));
           if(!float_mat[i]) {
            printf("\nErro na alocação de ponteiros de \
             colunas em %d na matrix %d x %d \n",i,linha,coluna);
            exit(1);
            }
         }
        AA->ptr_mat = (char **)float_mat;
        break;
       }
      case sizeof(double): {
        double **double_mat;
        double_mat = (double **)calloc(linha,sizeof(double *));
        if(!double_mat) {
            printf("\nErro na alocação de ponteiros de linhas \
                 na matriz %dx%d \n",linha,coluna);
            exit(1);
          }
        for(i = 0; i < linha; i++) {
           double_mat[i] = (double *)calloc(coluna,sizeof(double));
           if(!double_mat[i]) {
            printf("\nErro na alocação de ponteiros de \

              colunas em %d na matrix %dx%d \n",i,linha,coluna);
            exit(1);
            }
         }
        AA->ptr_mat = (char **)double_mat;
```

```
              break;
             }
            default:
              printf("\nErro na alocação da matriz: tipo não \
                      suportado \n");
              exit(1);
         }
       return(AA);
      }
/*************************************************************************
 *************************************************************************
libera_mat   libera a área da matriz alocada para dados, bem como
           ponteiros e estruturas mapa MATRIX. Emite mensagem de
           Erro e provoca exit(1) caso ocorra uma passagem de estrutura
           impropria (ponteiro NULL ou matriz de tamanho zero ).

void libera_mat(MATRIX *AA)
*************************************************************************
/
void libera_mat(MATRIX *AA)
      {
       int i;
       char **livre;

       if(!AA || !AA->ptr_mat || !AA->linha || !AA->coluna) {
          printf("\nErro: a estrutura passada eh invalida \n");
          exit(1);
        }

       livre = AA->ptr_mat;

       for(i = 0; i < AA->linha; i++)
           free(livre[i]);

       free((char *)livre);
       livre = NULL;

       free((char *)AA);
       }
/*************************************************************************/
```

```
/*-------------------------------------------------------------------
DADCOMI1.C  Decodificação adaptativa de imagens monocromaticas codificada
      pelo programa ADCOMI1.C
-------------------------------------------------------------------*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <process.h>
#include <io.h>
#include <conio.h>
#include <math.h>
#include "maniparq.h"
/*******************************************************************/
main()
{
  MATRIX *Ximage,*SCEAC;
  int i,j,k,l,p,q,N,M,NA,lin,col,linha,coluna,status,Xdc;
  char **sceac,*Xac,*Yac;
  float **xif;
  float fn,fn0,fn1,fn2,fn3,fn4,fn5,fn6,fn7,fn8,fn9,fn10,fn11;
  char *nome_arq,*arq,arq_temp[14],tam_elemf,tam_elemc,ch;
  ARQUIVO *arq_out;
  FILE *arqhdr,*arqdat,*arqvdc;
  int zigzag[64] = { 0, 1, 8,16, 9, 2, 3,10,
                17,24,32,25,18,11, 4, 5,
                12,19,26,33,40,48,41,34,
                27,20,13, 6, 7,14,21,28,
                35,42,49,56,57,50,43,36,
                29,22,15,23,30,37,44,51,
                58,59,52,45,38,31,39,46,
                53,60,61,54,47,62,55,63 };

  int vertical[64] = {0, 8,16,24,32,40,48,56,
                  1, 9,17,25,33,41,49,57,
                  2,10,18,26,34,42,50,58,
                  3,11,19,27,35,43,51,59,
                  4,12,20,28,36,44,52,60,
                  5,13,21,29,37,45,53,61,
                  6,14,22,30,38,46,54,62,
                  7,15,23,31,39,47,55,63 };
/*-------------------------------------------------------------------*/
printf("\nPrograma DADCOMI1 para decodificação adaptativa de imagens");
printf("\nmonocromaticas codificada pelo programa ADCOMI1");
/*-------------------------------------------------------------------*/
  printf("\nLer TRES Arquivos com extensao .HDR,.DAT e .VDC da imagem ");
  printf("\nCODIFICADA. Certifique-se da existencia desses arquivos.");
  nome_arq = ler_str("nome do arquivo SEM EXTENÇÃO para DECODIFICAÇÃO ");
   arq = strchr(nome_arq,'.');
   if(arq != NULL) {
      k = strcspn(nome_arq,".");
      nome_arq[k] = '\0';
     }
   /*-----------------------------------------------------*/
   arq = strcpy(arq_temp,nome_arq);
   arq = strcat(arq_temp,".hdr");
   arqhdr = fopen(arq,"rb");
```

```
   /*-----------------------------------------------------*/
   arq = strcpy(arq_temp,nome_arq);
   arq = strcat(arq_temp,".dat");
   arqdat = fopen(arq,"rb");
   /*-----------------------------------------------------*/
   arq = strcpy(arq_temp,nome_arq);
   arq = strcat(arq_temp,".vdc");
   arqvdc = fopen(arq,"rb");
   /*-----------------------------------------------------*/
/*-------------------------------------------------------------------*/
   do {
      nome_arq = ler_str("nome do arquivo da saída DECODIFICADO ");
      arq_out = abre_escreve(nome_arq);
    } while(!arq_out);
/*-------------------------------------------------------------------*/
   N = 8;
   NA = N*N;
   fread(&linha,sizeof(int),1,arqhdr);
   fread(&coluna,sizeof(int),1,arqhdr);
   M = coluna/N;
   SCEAC = aloca_mat(M,M,sizeof(char));
   sceac = (char **)SCEAC->ptr_mat;
   for(i = 0; i < M; i++) {
       status = fread((char *)SCEAC->ptr_mat[i],sizeof(char),M,arqhdr);
       if(status != M) {
        printf("\nErro na gravação de registro em %s \n",arqhdr);
        exit(1);
      }
    }
   fread(&fn0,sizeof(float),1,arqhdr);
   fread(&fn1,sizeof(float),1,arqhdr);
   fread(&fn2,sizeof(float),1,arqhdr);
   fread(&fn3,sizeof(float),1,arqhdr);
   fread(&fn4,sizeof(float),1,arqhdr);
   fread(&fn5,sizeof(float),1,arqhdr);
   fread(&fn6,sizeof(float),1,arqhdr);
   fread(&fn7,sizeof(float),1,arqhdr);
   fread(&fn8,sizeof(float),1,arqhdr);
   fread(&fn9,sizeof(float),1,arqhdr);
   fread(&fn10,sizeof(float),1,arqhdr);
   fread(&fn11,sizeof(float),1,arqhdr);
   fclose(arqhdr);
/*-------------------------------------------------------------------*/
/*-------------------------------------------------------------------*/
   tam_elemf = sizeof(float);
   Ximage = aloca_mat(N,coluna,tam_elemf);
   xif = (float **)Ximage->ptr_mat;
   tam_elemc = sizeof(char);
   Xac = (char *)calloc(NA-1,tam_elemc);
   Yac = (char *)calloc(NA-1,tam_elemc);

   for(lin = 0; lin < linha; lin+=N) {
      k = (int)(lin/N);

      for(col = 0; col < coluna; col+=N) {
        l = (int)(col/N);
        if((Xdc = getc(arqvdc)) == EOF) {
```

```
            printf("\nErro na leitura do valor DC em %s \n",arqvdc);
            exit(1);
          }
        if((fread((char *)Yac,tam_elemc,NA-1,arqdat)) != NA-1) {
             printf("\nErro na leitura de registro em %s \n",arqdat);
             exit(1);
          }
        p = sceac[k][l];
        if(p==0)  fn = fn0;   if(p==1)  fn=fn1;
        if(p==2)  fn = fn2;   if(p==3)  fn=fn3;
        if(p==4)  fn = fn4;   if(p==5)  fn=fn5;
        if(p==6)  fn = fn6;   if(p==7)  fn=fn7;
        if(p==8)  fn = fn8;   if(p==9)  fn=fn9;
        if(p==10) fn = fn10;  if(p==11) fn=fn11;
        for(q = 0; q < NA-1; q++) {
            if(p==0 || p==1 || p==2 || p==3)
               Xac[q] = Yac[q];
            if(p==4 || p==5 || p==6 || p==7)
               Xac[ vertical[q] ] = Yac[q];
            if(p==8 || p==9 || p==10 || p==11)
               Xac[ zigzag[q] ] = Yac[q];
          }
        q = 0;
        for(i = 0; i < N; i++) {
            for(j = 0; j < N; j++) {
              if(i == 0 && j == 0) {
                  xif[i][col+j] = (float)(Xdc)*2.0;
                }
              else {
                 xif[i][col+j] =  (float)(Xac[q])*fn;
                 q++;
               }
             }
          }
       }
      for(i = 0; i < N; i++)
        grava_registro((char *)Ximage-
>ptr_mat[i],arq_out,coluna,tam_elemf);

    }
   libera_mat(SCEAC);
   libera_mat(Ximage);
/*-------------------------------------------------------------------------*/
fcloseall();

/*-------------------------------------------------------------------------*/
}
```

```
/*-------------------------------------------------------------------
DADCOMI3.C  Decodificação adaptativa de imagens monocromaticas codificada
      pelo programa ADCOMI1.C
-------------------------------------------------------------------*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <process.h>
#include <io.h>
#include <conio.h>
#include <math.h>
#include "maniparq.h"
#include "JPG_H.H"
/*******************************************************************/

static BYTE chbyte;

main()
{
  MATRIX *Ximage,*SCEAC;
  int i,j,k,l,p,q,N,M,NA,lin,col,linha,coluna,status;
  long contabits;
  int_bloco Xdeq,Yzag;
  char **sceac;
  float **xif;
  float fn,fn0,fn1,fn2,fn3,fn4,fn5,fn6,fn7,fn8,fn9,fn10,fn11;
  char *nome_arq,*arq,arq_temp[14],tam_elemf,tam_elemc,ch;
  ARQUIVO *arq_out;
  FILE *arqhdr,*arqdat;
  int zigzag[64] = { 0, 1, 8,16, 9, 2, 3,10,
                17,24,32,25,18,11, 4, 5,
                12,19,26,33,40,48,41,34,
                27,20,13, 6, 7,14,21,28,
                35,42,49,56,57,50,43,36,
                29,22,15,23,30,37,44,51,
                58,59,52,45,38,31,39,46,
                53,60,61,54,47,62,55,63 };

  int vertical[64] = {0, 8,16,24,32,40,48,56,
                  1, 9,17,25,33,41,49,57,
                  2,10,18,26,34,42,50,58,
                  3,11,19,27,35,43,51,59,
                  4,12,20,28,36,44,52,60,
                  5,13,21,29,37,45,53,61,
                  6,14,22,30,38,46,54,62,
                  7,15,23,31,39,47,55,63 };
/*-------------------------------------------------------------------*/
printf("\nPrograma DADCOMI3 para decodificação adaptativa de imagens");
printf("\nmonocromaticas codificada pelo programa ADCOMI1");
/*-------------------------------------------------------------------*/
  printf("\nLer TRES Arquivos com extensao .HDR,.DAT e .VDC da imagem ");
  printf("\nCODIFICADA. Certifique-se da existencia desses arquivos.");
  nome_arq = ler_str("nome do arquivo SEM EXTENÇÃO para DECODIFICAÇÃO ");
   arq = strchr(nome_arq,'.');

   if(arq != NULL) {
```

```
      k = strcspn(nome_arq,".");
      nome_arq[k] = '\0';
     }
   /*-------------------------------------------------------*/
   arq = strcpy(arq_temp,nome_arq);
   arq = strcat(arq_temp,".hdr");
   arqhdr = fopen(arq,"rb");
   /*-------------------------------------------------------*/
   arq = strcpy(arq_temp,nome_arq);
   arq = strcat(arq_temp,".dat");
   arqdat = fopen(arq,"rb");
   /*-------------------------------------------------------*/
/*-----------------------------------------------------------------------*/
   do {
      nome_arq = ler_str("nome do arquivo da saída DECODIFICADO ");
      arq_out = abre_escreve(nome_arq);
    } while(!arq_out);
/*-----------------------------------------------------------------------*/
   N = 8;
   NA = N*N;
   fread(&linha,sizeof(int),1,arqhdr);
   fread(&coluna,sizeof(int),1,arqhdr);
   M = coluna/N;
   SCEAC = aloca_mat(M,M,sizeof(char));
   sceac = (char **)SCEAC->ptr_mat;
   for(i = 0; i < M; i++) {
       status = fread((char *)SCEAC->ptr_mat[i],sizeof(char),M,arqhdr);
       if(status != M) {
        printf("\nErro na gravação de registro em %s \n",arqhdr);
        exit(1);
      }
    }
   fread(&fn0,sizeof(float),1,arqhdr);
   fread(&fn1,sizeof(float),1,arqhdr);
   fread(&fn2,sizeof(float),1,arqhdr);
   fread(&fn3,sizeof(float),1,arqhdr);
   fread(&fn4,sizeof(float),1,arqhdr);
   fread(&fn5,sizeof(float),1,arqhdr);
   fread(&fn6,sizeof(float),1,arqhdr);
   fread(&fn7,sizeof(float),1,arqhdr);
   fread(&fn8,sizeof(float),1,arqhdr);
   fread(&fn9,sizeof(float),1,arqhdr);
   fread(&fn10,sizeof(float),1,arqhdr);
   fread(&fn11,sizeof(float),1,arqhdr);
   fclose(arqhdr);
/*-----------------------------------------------------------------------*/
/*-----------------------------------------------------------------------*/
   tam_elemf = sizeof(float);
   Ximage = aloca_mat(N,coluna,tam_elemf);
   xif = (float **)Ximage->ptr_mat;

   contabits=0L;
   chint = getc(arqdat);
   chbyte = (BYTE)chint;

   for(lin = 0; lin < linha; lin+=N) {
      k = (int)(lin/N);
```

```
      for(col = 0; col < coluna; col+=N) {
        l = (int)(col/N);
        p = sceac[k][l];
        if(p==0)  fn = fn0;   if(p==1)  fn=fn1;
        if(p==2)  fn = fn2;   if(p==3)  fn=fn3;
        if(p==4)  fn = fn4;   if(p==5)  fn=fn5;
        if(p==6)  fn = fn6;   if(p==7)  fn=fn7;
        if(p==8)  fn = fn8;   if(p==9)  fn=fn9;
        if(p==10) fn = fn10;  if(p==11) fn=fn11;

        contabits=DECHUFF(arqdat,&Yzag,contabits,&chbyte);
        for(q = 0; q < NA; q++) {
            if(p==0 || p==1 || p==2 || p==3)
               Xdeq.linear[q] = Yzag.linear[q];
            if(p==4 || p==5 || p==6 || p==7)
               Xdeq.linear[ vertical[q] ] = Yzag.linear[q];
            if(p==8 || p==9 || p==10 || p==11)
               Xdeq.linear[ zigzag[q] ] = Yzag.linear[q];
          }
        q = 0;
        for(i = 0; i < N; i++) {
            for(j = 0; j < N; j++) {
              if(i == 0 && j == 0) {
                  xif[i][col+j] = (float)(Xdeq.linear[q])*2.0;
                  q++;
                 }
              else {
                 xif[i][col+j] = (float)(Xdeq.linear[q])*fn;
                 q++;
                }
             }
          }
       }
      for(i = 0; i < N; i++)
        grava_registro((char *)Ximage-
>ptr_mat[i],arq_out,coluna,tam_elemf);

    }
   libera_mat(SCEAC);
   libera_mat(Ximage);
/*-------------------------------------------------------------------*/
fcloseall();
/*-------------------------------------------------------------------*/
}
```

```
/*-------------------------------------------------------------------
DADCOMI5.C  Decodificação adaptativa de imagens monocromaticas codificada
      pelo programa ADCOMI1.C
-------------------------------------------------------------------*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <process.h>
#include <io.h>
#include <conio.h>
#include <math.h>
#include "maniparq.h"
#include "JPG_H.H"
/*******************************************************************/
float DQUNIFORM(float,char,char);
static BYTE chbyte;

main()
{
  MATRIX *Ximage,*BIJ,*MAXIMO,*SCEAC;
  int i,j,k,l,p,q,N,Ml,Mc,NA,lin,col,linha,coluna,status,xq;
  long contabits;
  int_bloco Xdeq,Yzag;
  char **bij,**max,**sceac;
  float **xif,xf;
  float fn,fn0,fn1,fn2,fn3,fn4,fn5,fn6,fn7,fn8,fn9,fn10,fn11;
  char *nome_arq,*arq,arq_temp[14],tam_elemf,tam_elemc,ch;
  ARQUIVO *arq_out,*arq_in;
  FILE *arqhdr,*arqdat;
  int zigzag[64] = { 0, 1, 8,16, 9, 2, 3,10,
                17,24,32,25,18,11, 4, 5,
                12,19,26,33,40,48,41,34,
                27,20,13, 6, 7,14,21,28,
                35,42,49,56,57,50,43,36,
                29,22,15,23,30,37,44,51,
                58,59,52,45,38,31,39,46,
                53,60,61,54,47,62,55,63 };

  int vertical[64] = {0, 8,16,24,32,40,48,56,
                  1, 9,17,25,33,41,49,57,
                  2,10,18,26,34,42,50,58,
                  3,11,19,27,35,43,51,59,
                  4,12,20,28,36,44,52,60,
                  5,13,21,29,37,45,53,61,
                  6,14,22,30,38,46,54,62,
                  7,15,23,31,39,47,55,63 };
/*-------------------------------------------------------------------*/
printf("\nPrograma DADCOMI5 para decodificação adaptativa de imagens");
printf("\nmonocromáticas codificada pelo programa ADCOMI1");
/*-------------------------------------------------------------------*/
 printf("\nLer TRES Arquivos com extensao .HDR, .DAT e .VDC da imagem ");
 printf("\nCODIFICADA. Certifique-se da existencia desses arquivos.");
do {
    nome_arq = ler_str("nome do arquivo com os coeficientes DCT2D ");

    arq_in = abre_ler(nome_arq);
```

```
    printf("\nDimensoes do arquivo:");
    linha = ler_int("número de linhas",1,512);
    coluna = ler_int("número de colunas",1,1024);
 } while(!arq_in);

  nome_arq = ler_str("nome do arquivo SEM EXTENÇÃO para DECODIFICAÇÃO ");
   arq = strchr(nome_arq,'.');
   if(arq != NULL) {
      k = strcspn(nome_arq,".");
      nome_arq[k] = '\0';
     }
   /*-------------------------------------------------------*/
   arq = strcpy(arq_temp,nome_arq);
   arq = strcat(arq_temp,".hdr");
   arqhdr = fopen(arq,"rb");
   /*-------------------------------------------------------*/
   /*arq = strcpy(arq_temp,nome_arq);*/
   /*arq = strcat(arq_temp,".dat");*/
   /*arqdat = fopen(arq,"rb");*/
   /*-------------------------------------------------------*/
/*-----------------------------------------------------------------*/
   do {
      nome_arq = ler_str("nome do arquivo da saída DECODIFICADO ");
      arq_out = abre_escreve(nome_arq);
    } while(!arq_out);
/*-----------------------------------------------------------------*/
   N = 8;
   NA = N*N;
   fread(&linha,sizeof(int),1,arqhdr);
   fread(&coluna,sizeof(int),1,arqhdr);
   Ml = linha/N;
   Mc = coluna/N;
   SCEAC = aloca_mat(Ml,Mc,sizeof(char));
   sceac = (char **)SCEAC->ptr_mat;
   for(i = 0; i < Ml; i++) {
       status = fread((char *)SCEAC->ptr_mat[i],sizeof(char),Mc,arqhdr);
       if(status != Mc) {
        printf("\nErro na gravação de registro em %s \n",arqhdr);
        exit(1);
       }
    }
   BIJ = aloca_mat(12,NA,sizeof(char));
   bij = (char **)BIJ->ptr_mat;
   for(i = 0; i < 12; i++) {
       status = fread((char *)BIJ->ptr_mat[i],sizeof(char),NA,arqhdr);
       if(status != NA) {
        printf("\nErro na gravação de registro em %s \n",arqhdr);
        exit(1);
       }
    }
   MAXIMO = aloca_mat(12,NA,sizeof(char));
   max = (char **)MAXIMO->ptr_mat;
   for(i = 0; i < 12; i++) {

       status = fread((char *)MAXIMO->ptr_mat[i],sizeof(char),NA,arqhdr);
       if(status != NA) {
        printf("\nErro na gravação de registro em %s \n",arqhdr);
```

```
        exit(1);
      }
    }
   fread(&fn0,sizeof(float),1,arqhdr);
   fread(&fn1,sizeof(float),1,arqhdr);
   fread(&fn2,sizeof(float),1,arqhdr);
   fread(&fn3,sizeof(float),1,arqhdr);
   fread(&fn4,sizeof(float),1,arqhdr);
   fread(&fn5,sizeof(float),1,arqhdr);
   fread(&fn6,sizeof(float),1,arqhdr);
   fread(&fn7,sizeof(float),1,arqhdr);
   fread(&fn8,sizeof(float),1,arqhdr);
   fread(&fn9,sizeof(float),1,arqhdr);
   fread(&fn10,sizeof(float),1,arqhdr);
   fread(&fn11,sizeof(float),1,arqhdr);
   fclose(arqhdr);
/*-------------------------------------------------------------------*/
/*-------------------------------------------------------------------*/
   tam_elemf = sizeof(float);
   Ximage = aloca_mat(N,coluna,tam_elemf);
   xif = (float **)Ximage->ptr_mat;

   contabits=0L;
   chint = getc(arqdat);
   chbyte = (BYTE)chint;
   for(lin = 0; lin < linha; lin+=N) {
      k = (int)(lin/N);
      for(i = 0; i < N; i++)
        ler_registro((char *)Ximage->ptr_mat[i],arq_in,coluna,tam_elemf);
      for(col = 0; col < coluna; col+=N) {
        l = (int)(col/N);
        p = sceac[k][l];
        if(p==0)  fn = fn0;   if(p==1)  fn=fn1;
        if(p==2)  fn = fn2;   if(p==3)  fn=fn3;
        if(p==4)  fn = fn4;   if(p==5)  fn=fn5;
        if(p==6)  fn = fn6;   if(p==7)  fn=fn7;
        if(p==8)  fn = fn8;   if(p==9)  fn=fn9;
        if(p==10) fn = fn10;  if(p==11) fn=fn11;
        q = 0;
        for(i = 0; i < N; i++) {
            for(j = 0; j < N; j++) {
              if(i == 0 && j == 0) {
                  xif[i][col+j] = xif[i][col+j]*2.0*fn;
                  xf = xif[i][col+j];
                 }
              else {
                 if(bij[p][q] == 0)  xif[i][col+j] = 0.0;
                 else {
                   xf = xif[i][col+j];
                   xf = DQUNIFORM(xf,max[p][q],bij[p][q]);
                   xif[i][col+j] = xf*fn;
                   xf = xif[i][col+j];
                   }
               }
              q++;
             }
          }
```

```
       }
      for(i = 0; i < N; i++)
        grava_registro((char *)Ximage-
>ptr_mat[i],arq_out,coluna,tam_elemf);

    }
   libera_mat(SCEAC);
   libera_mat(Ximage);
/*-------------------------------------------------------------------------*/
fcloseall();
/*-------------------------------------------------------------------------*/
}
/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/
float DQUNIFORM(float xq,char max,char bij)
{
  float vq;
  float vt,vt1,vmax,vbij,nb;
  vmax = (float)max;  vbij = (float)bij;
  nb = (float)pow(2.0,vbij-1.0);
  vt = ((xq*vmax)/nb);

  if(vbij==1.0) {
     if(xq ==  0.0)  vq =  0.0;
     if(xq == -1.0)  vq = -0.5;
     if(xq ==  1.0)  vq =  0.5;
   }
  else if(vbij == 2.0) {
     if(xq == -2.0)       vq = -1.5;
     else if(xq == -1.0)  vq = -0.75;
     else if(xq ==  0.0)  vq =  0.0;
     else if(xq ==  1.0)  vq =  0.75;
    }
  else if(vbij == 3.0) {
     vt1 = fabs(xq);
     if(vt1 == 4.0)       vq = 3.5;
     else if(vt1 == 3.0)  vq = 2.5;
     else if(vt1 == 2.0)  vq = 1.5;
     else if(vt1 == 1.0)  vq = 0.5;
     if(xq < 0.0)         vq = -vq;
    }
  else if(vbij > 3.0)
        vq = xq;
  vt = ((vq*vmax)/nb);

  return(vt);
}
```

```
/*********************************************************************
LER.C  ler string, inteiro e float entrados pelo usuario a partir do
       prompt.
 *********************************************************************
ler_str    ler string entrada pelo usuário a partir do prompt
ler_int    ler inteiro entrado pelo usuário dentro de um range
ler_float  ler float idem

 *********************************************************************
 *********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "maniparq.h"
/*********************************************************************
ler_str  ler uma string entrada pelo usuário a partir do prompt e retorna
       um ponteiro para a string passada pelo chamador. Indica erro se
        o espaço da string não for alocado. Limitado em 80 caracter

char *ler_str(char *prompt_str)
*********************************************************************/
char *ler_str(char *prompt_str)
   {
    char *texto;

    texto = (char *)malloc(200);
    if(!texto) {
      printf("\nErro de alocação de string em ler_str() \n");
      exit(1);
    }
    printf("\n\nEntre %s : ",prompt_str);
    gets(texto);

    return(texto);
   }
/*********************************************************************
 ler_int   ler um texto entrado pelo usuário a partir do prompt e retorna
        um inteiro, dentro de um range de valores (inferior e superior)
        passado pelo chamador. Indica erro se o valor passado estiver
        fora do range especificado.

int ler_int(char *prompt_str, int limite_inf, int limite_sup)
*********************************************************************/
int ler_int(char *prompt_str, int limite_inf, int limite_sup)
  {
   char *ler_str();
   int valor, sinal_erro;
   char *cp, *endcp;
   char *str_temp;

   if(limite_inf > limite_sup) {
     printf("\nErro de limites de entrada, limite_inf > limite_sup \n");
     exit(1);
   }

   str_temp = (char *)malloc(strlen(prompt_str) + 200);
```

```
   if(!str_temp) {
     printf("\nErro de alocação de string em ler_int() \n");
     exit(1);
   }
   sprintf(str_temp, "%s [%d...%d]",prompt_str,limite_inf,limite_sup);

   do {
       cp = ler_str(str_temp);
       valor = (int)strtol(cp,&endcp,10);
       sinal_erro = (cp == endcp) || (*endcp != '\0');
       free(cp);
      } while(valor < limite_inf || valor > limite_sup || sinal_erro);

   free(str_temp);
   return(valor);
  }
/*********************************************************************
ler_float  ler um texto entrado pelo usuário a partir do prompt e retorna
         um double, dentro de um range de valores (inferior e superior)
         passado pelo chamador. Indica erro se o valor passado estiver
         fora do range especificado.

double ler_float(char *prompt_str, int limite_inf, int limite_sup)
*********************************************************************/
double ler_float(char *prompt_str, double limite_inf, double limite_sup)
  {
   char *ler_str();
   double valor;
   int sinal_erro;
   char *cp, *endcp;
   char *str_temp;

   if(limite_inf > limite_sup) {
     printf("\nErro de limites de entrada, limite_inf > limite_sup \n");
     exit(1);
   }

   str_temp = (char *)malloc(strlen(prompt_str) + 200);
   if(!str_temp) {
     printf("\nErro de alocação de string em ler_int() \n");
     exit(1);
   }
   sprintf(str_temp, "%s
[%1.2g...%1.2g]",prompt_str,limite_inf,limite_sup);

   do {
       cp = ler_str(str_temp);
       valor = strtod(cp,&endcp);
       sinal_erro = (cp == endcp) || (*endcp != '\0');
       free(cp);
      } while(valor < limite_inf || valor > limite_sup || sinal_erro);

   free(str_temp);
   return(valor);

  }
/*********************************************************************/
```

```
/*********************************************************************
MANIPMAT.C  Rotinas de manipulação de matrizes. Essas rotinas realizam
          operações do tipo soma, subtração, multiplicação ponto a
          ponto e produto direto de duas matrizes, extrai submatrizes,
          transpoen, inverte e extrai o determinante das mesmas.

*********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "maniparq.h"

/*********************************************************************
escala()  multiplica uma matriz por um escalar de valor s. Retorna uma
         nova matriz escalada como um ponteiro para a nova estrutura. O
         escalar "s" eh uma constante double. Matrizes inteiras sao
         convertidas para float quando escaladas.

MATRIX *escalar(MATRIX *AA,double s)
*********************************************************************/
MATRIX *escalar(MATRIX *AA,double s)
     {
       MATRIX *aloca_mat();
       MATRIX *BB;
       int lin_a, col_a;

       lin_a = AA->linha;
       col_a = AA->coluna;

       switch(AA->tamanho) {
           case sizeof(short):
             BB = aloca_mat(lin_a,col_a,sizeof(float));
             ESCALAR_MAT(AA,BB,*,s,lin_a,col_a,0,0,short,float)
             break;
           case sizeof(float):
             BB = aloca_mat(lin_a,col_a,sizeof(float));
             ESCALAR_MAT(AA,BB,*,s,lin_a,col_a,0,0,float,float)
             break;
           case sizeof(double):
             BB = aloca_mat(lin_a,col_a,sizeof(double));
             ESCALAR_MAT(AA,BB,*,s,lin_a,col_a,0,0,double,double)
        }
      return(BB);

     }
```

```
/*********************************************************************
sub_mat()  extrai uma sub_matriz de uma matriz maior dependendo da linha
        e coluna de offset que indicam a posição de inicio da extração.
        Retorna uma nova matriz menor como um ponteiro para a nova
        estrutura com o número de linha e coluna entrado pelo usuário.

MATRIX *sub_mat(MATRIX *AA,int lin_off,int col_off,int lin,int col)
*********************************************************************/
MATRIX *sub_mat(MATRIX *AA,int lin_off,int col_off,int lin,int col)
       {

        MATRIX *aloca_mat();
        MATRIX *BB;

      if((lin+lin_off) > AA->linha || (col+col_off) > AA->coluna) {
        printf("\nErro, a sub_matriz excede o tamanho da matriz
maior\n");
        exit(1);
       }

      BB = aloca_mat(lin,col,AA->tamanho);

        switch(AA->tamanho) {
           case sizeof(short):
             ESCALAR_MAT(AA,BB,*,1,lin,col,lin_off,col_off,short,short)
             break;
           case sizeof(float):
             ESCALAR_MAT(AA,BB,*,1,lin,col,lin_off,col_off,float,float)
             break;
           case sizeof(double):
             ESCALAR_MAT(AA,BB,*,1,lin,col,lin_off,col_off,double,double)
        }
      return(BB);
     }
/*********************************************************************/
/*********************************************************************
somar() função para operar duas matrizes, elemento por elemento,
      usando a macro ELEMENTO_MAT que realiza soma, subtração e
      multiplicação elemento a elemento das matrizes "A" com "B".
      O número de linhas e colunas de ambas matrizes deve ser iguais.
      Retorna um ponteiro para a nova estrutura de matriz escalada "C".
      O tipo retornado depende dos tipos recebidos pela macro.
      Operações do tipo C = A+B, C = A-B, C = A.*B. O operador deve ser
      + (soma), - (subtração) ou * (multiplicação).
MATRIX *somar(MATRIX *AA,MATRIX *BB)
*********************************************************************/
MATRIX *somar(MATRIX *AA,MATRIX *BB)
     {
       MATRIX *aloca_mat();
       MATRIX *CC;
       int lin_a, col_a;

      if(BB->linha !=  AA->linha || BB->coluna != AA->coluna) {
        printf("\nErro, a matriz BB deve ser do mesmo tamanho de AA \n");
        printf("\nAA eh %dx%d e BB eh %dx%d \n"
                  ,AA->linha,AA->coluna,BB->linha,BB->coluna);
        exit(1);
```

```
    }

    lin_a = AA->linha;
    col_a = AA->coluna;

    if(AA->tamanho > BB->tamanho)
         CC = aloca_mat(lin_a,col_a,AA->tamanho);
    else
         CC = aloca_mat(lin_a,col_a,BB->tamanho);

     switch(AA->tamanho) {
        case sizeof(short):
         switch(BB->tamanho) {
           case sizeof(short):
             ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,+,short,short,short)
             break;
           case sizeof(float):
             ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,+,short,float,float)
             break;
           case sizeof(double):
             ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,+,short,double,double)
          }
          break;
        case sizeof(float):
         switch(BB->tamanho) {
           case sizeof(short):
             ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,+,float,short,float)
             break;
           case sizeof(float):
             ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,+,float,float,float)
             break;
           case sizeof(double):
             ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,+,float,double,double)
          }
          break;
        case sizeof(double):
         switch(BB->tamanho) {
           case sizeof(short):
             ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,+,double,short,double)
             break;
           case sizeof(float):
             ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,+,double,float,double)
             break;
           case sizeof(double):
             ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,+,double,double,double)
          }
          break;
       }
     return(CC);

}
```

```
/*********************************************************************
*********************************************************************
subtrair() função para operar duas matrizes, elemento por elemento,
      usando a macro ELEMENTO_MAT que realiza soma, subtração e
      multiplicação elemento a elemento das matrizes "A" com "B".
      O número de linhas e colunas de ambas matrizes deve ser iguais.
      Retorna um ponteiro para a nova estrutura de matriz escalada "C".
      O tipo retornado depende dos tipos recebidos pela macro.
      Operações do tipo C = A+B, C = A-B, C = A.*B. O operador deve ser
      + (soma), - (subtração) ou * (multiplicação).
MATRIX *subtrair(MATRIX *AA,MATRIX *BB)
*********************************************************************/
MATRIX *subtrair(MATRIX *AA,MATRIX *BB)
     {

       MATRIX *aloca_mat();
       MATRIX *CC;
       int lin_a, col_a;
      if(BB->linha !=  AA->linha || BB->coluna != AA->coluna) {
        printf("\nErro, a matriz BB deve ser do mesmo tamanho de AA \n");
        printf("\nMatriz AA eh %dx%d e BB eh %dx%d \n"
                  ,AA->linha,AA->coluna,BB->linha,BB->coluna);
        exit(1);
       }

       lin_a = AA->linha;
       col_a = AA->coluna;

       if(AA->tamanho > BB->tamanho)
            CC = aloca_mat(lin_a,col_a,AA->tamanho);
       else
            CC = aloca_mat(lin_a,col_a,BB->tamanho);

        switch(AA->tamanho) {
           case sizeof(short):
            switch(BB->tamanho) {
              case sizeof(short):
                ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,-,short,short,short)
                break;
              case sizeof(float):
                ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,-,short,float,float)
                break;
              case sizeof(double):
                ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,-,short,double,double)
             }
             break;
           case sizeof(float):
            switch(BB->tamanho) {
              case sizeof(short):
                ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,-,float,short,float)
                break;
              case sizeof(float):
                ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,-,float,float,float)
                break;
              case sizeof(double):
                ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,-,float,double,double)
             }
```

```
            break;
          case sizeof(double):
           switch(BB->tamanho) {
             case sizeof(short):
               ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,-,double,short,double)
               break;
             case sizeof(float):
               ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,-,double,float,double)
               break;
             case sizeof(double):
               ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,-,double,double,double)
           }

            break;
         }
       return(CC);

    }
/*********************************************************************
*********************************************************************
mult_piece() função para operar duas matrizes, elemento por elemento,
     usando a macro ELEMENTO_MAT que realiza soma, subtração e
     multiplicação elemento a elemento das matrizes "A" com "B".
     O número de linhas e colunas de ambas matrizes deve ser iguais.
     Retorna um ponteiro para a nova estrutura de matriz escalada "C".
     O tipo retornado depende dos tipos recebidos pela macro.
     Operações do tipo C = A+B, C = A-B, C = A.*B. O operador deve ser
     + (soma), - (subtração) ou * (multiplicação).
MATRIX *mult_piece(MATRIX *AA,MATRIX *BB)
*********************************************************************/
MATRIX *mult_piece(MATRIX *AA,MATRIX *BB)
    {
      MATRIX *aloca_mat();
      MATRIX *CC;
      int lin_a, col_a;

     if(BB->linha !=  AA->linha || BB->coluna != AA->coluna) {
       printf("\nErro, em multiplicar(), a matriz BB deve ser do mesmo \
                tamanho de AA \n");
       printf("\nMatriz AA eh %dx%d e matriz BB eh %dx%d \n"
                ,AA->linha,AA->coluna,BB->linha,BB->coluna);
       exit(1);
      }

      lin_a = AA->linha;
      col_a = AA->coluna;

      if(AA->tamanho > BB->tamanho)
           CC = aloca_mat(lin_a,col_a,AA->tamanho);
      else
           CC = aloca_mat(lin_a,col_a,BB->tamanho);

       switch(AA->tamanho) {
          case sizeof(short):
            switch(BB->tamanho) {
            case sizeof(short):
              ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,*,short,short,short)
```

```
              break;
             case sizeof(float):
               ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,*,short,float,float)
               break;
             case sizeof(double):
                ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,*,short,double,double)
            }
            break;
           case sizeof(float):
             switch(BB->tamanho) {
             case sizeof(short):

               ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,*,float,short,float)
               break;
             case sizeof(float):
               ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,*,float,float,float)
               break;
             case sizeof(double):
               ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,*,float,double,double)
            }
            break;
           case sizeof(double):
             switch(BB->tamanho) {
             case sizeof(short):
               ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,*,double,short,double)
               break;
             case sizeof(float):
               ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,*,double,float,double)
               break;
             case sizeof(double):
               ELEMENTO_MAT(AA,BB,CC,lin_a,col_a,*,double,double,double)
             }
             break;
         }
       return(CC);

    }
/*********************************************************************/
/*********************************************************************
multiplicar() função para realizar o produto de duas matrizes, "A" e "B"
            do tipo C = A*B usando a macro PRODUTO_MAT. O número de
linhas
            de "B" deve ser igual ao número de colunas de "A". Retorna um
            ponteiro para a nova estrutura de matriz "C". O tipo
            retornado depende dos tipos recebidos pela macro. Emite erro
            e exit(1) se "A" e "B" não estao corretas
MATRIX *multiplicar(MATRIX *AA,MATRIX *BB)
*********************************************************************/
MATRIX *multiplicar(MATRIX *AA,MATRIX *BB)
 {
  MATRIX *aloca_mat();
  MATRIX *CC;
  int lin_a, col_a, col_b;


  if(BB->linha !=  AA->coluna) {
    printf("\nErro, as linhas de BB deve ser iguas as colunas de AA \n");
```

```
        printf("\nMatriz AA eh %dx%d e BB eh %dx%d \n"
            ,AA->linha,AA->coluna,BB->linha,BB->coluna);
        exit(1);
       }

      if(!AA->ptr_mat) {
         printf("\nErro, nenhuma matriz AA esta alocada para produto\n");
         exit(1);
        }


      if(!BB->ptr_mat) {
         printf("\nErro, nenhuma matriz BB esta alocada para produto\n");
         exit(1);
        }

      /* Atribui os valores de linha e coluna da matrix AA */
       lin_a = AA->linha;
       col_a = AA->coluna;
       col_b = BB->coluna;

       if(AA->tamanho > BB->tamanho)
            CC = aloca_mat(lin_a,col_b,AA->tamanho);
       else
            CC = aloca_mat(lin_a,col_b,BB->tamanho);
        switch(AA->tamanho) {
          case sizeof(short):
            switch(BB->tamanho) {
            case sizeof(short):
              PRODUTO_MAT(AA,BB,CC,lin_a,col_a,col_b,short,short,short)
              break;
            case sizeof(float):
              PRODUTO_MAT(AA,BB,CC,lin_a,col_a,col_b,short,float,float)
              break;
            case sizeof(double):
              PRODUTO_MAT(AA,BB,CC,lin_a,col_a,col_b,short,double,double)
             }
             break;
          case sizeof(float):
            switch(BB->tamanho) {
            case sizeof(short):
              PRODUTO_MAT(AA,BB,CC,lin_a,col_a,col_b,float,short,float)
              break;
            case sizeof(float):
              PRODUTO_MAT(AA,BB,CC,lin_a,col_a,col_b,float,float,float)
              break;
            case sizeof(double):
              PRODUTO_MAT(AA,BB,CC,lin_a,col_a,col_b,float,double,double)
             }
             break;
          case sizeof(double):
            switch(BB->tamanho) {
            case sizeof(short):  /
              PRODUTO_MAT(AA,BB,CC,lin_a,col_a,col_b,double,short,double)
              break;
            case sizeof(float):
              PRODUTO_MAT(AA,BB,CC,lin_a,col_a,col_b,double,float,double)
```

```
             break;
           case sizeof(double):
        PRODUTO_MAT(AA,BB,CC,lin_a,col_a,col_b,double,double,double)
             }
             break;
         }
        return(CC);
     }

/*********************************************************************
transposta() função para realizar a transposição de uma matriz "A".
          Copia o resultado em uma nova matria At e retorna um ponteiro
          para a nova estrutura da matriz.
MATRIX *transposta(MATRIX *AA)
*********************************************************************/
MATRIX *transposta(MATRIX *AA)
      {
       MATRIX *aloca_mat();
       MATRIX *AAt;
       short **ai, **ait;
       float **af, **aft;
       double **ad, **adt;
       int i, j;

        AAt = aloca_mat(AA->coluna,AA->linha,AA->tamanho);

       switch(AA->tamanho) {
            case sizeof(short):
                  ai = (short **)AA->ptr_mat;
                  ait = (short **)AAt->ptr_mat;
                  for(i = 0; i < AA->linha; i++)
                      for(j = 0; j < AA->coluna; j++)
                        ait[j][i] = ai[i][j];
                  break;
            case sizeof(float):
                  af = (float **)AA->ptr_mat;
                  aft = (float **)AAt->ptr_mat;
                  for(i = 0; i < AA->linha; i++)
                      for(j = 0; j < AA->coluna; j++)
                        aft[j][i] = af[i][j];
                  break;
            case sizeof(double):
                  ad = (double **)AA->ptr_mat;
                  adt = (double **)AAt->ptr_mat;
                  for(i = 0; i < AA->linha; i++)
                      for(j = 0; j < AA->coluna; j++)
                        adt[j][i] = ad[i][j];
                  break;
         }
       return(AAt);

     }
```

```
/*********************************************************************
*********************************************************************/
/*QUANTMAX.C  Quantizadores Gauss e Laplace baseado no processo de MAX */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <io.h>
#include <string.h>
#include <process.h>
#include <math.h>
#include <alloc.h>

#include "maniparq.h"
/*-------------------------------------------------------------------*/
float gaus_d_1bit[2] = { 0.000000 };
float gaus_r_1bit[2] = { 0.797885 };
/*-------------------------------------------------------------------*/
float gaus_d_2bit[4]={ 0.000000, 0.981830 };
float gaus_r_2bit[4]={ 0.452931, 1.510729 };
/*-------------------------------------------------------------------*/
float gaus_d_3bit[4] = { 0.000000,  0.501148,  1.050993,  1.749115 };
float gaus_r_3bit[4] = { 0.245415,  0.756881,  1.345105,  2.153125 };
/*-------------------------------------------------------------------*/
float gaus_d_4bit[8] = { 0.000000,  0.258950,  0.523808,  0.801527,
                  1.101693,  1.439799,  1.846245,  2.403336 };

float gaus_r_4bit[8] = { 0.128766,  0.389134, 0.658481,  0.944573,
                  1.258813,  1.620786,  2.071703, 2.734968 };
/*-------------------------------------------------------------------*/
float gaus_d_5bit[16] = { 0.000000,  0.132752,  0.266263,  0.401324,
                   0.538796,  0.679651,  0.825036,  0.976349,
                   1.135370,  1.304453,  1.486866,  1.687404,
                   1.913631,  2.178737,  2.509647,  2.980666 };
float gaus_r_5bit[16] = { 0.066282,  0.199222,  0.333304,  0.469344,
                   0.608247,  0.751055,  0.899016,  1.053682,
                   1.217058,  1.391848,  1.581883,  1.792924,
                   2.034337,  2.323138,  2.696156,  3.265176 };
/*-------------------------------------------------------------------*/
float gaus_d_6bit[32] = { 0.000000,  0.079288,  0.158626,  0.238064,
                   0.317656,  0.397457,  0.477529,  0.557939,
                   0.638762,  0.720082,  0.801997,  0.884618,
                   0.968072,  1.052509,  1.138106,  1.225069,
                   1.313645,  1.404130,  1.496880,  1.592332,
                   1.691021,  1.793618,  1.900975,  2.014193,
                   2.134731,  2.264580,  2.406559,  2.564869,
                   2.746196,  2.962279,  3.237185,  3.637592 };
float gaus_r_6bit[32] = { 0.039638,  0.118939,  0.198314,  0.277815,
                   0.357497,  0.437417,  0.517640,  0.598237,
                   0.679286,  0.760878,  0.843117,  0.926119,
                   1.010025,  1.094994,  1.181217,  1.268920,
                   1.358370,  1.449890,  1.543871,  1.640793,
                   1.741249,  1.845988,  1.955963,  2.072424,
                   2.197039,  2.332120,  2.480998,  2.648740,
                   2.843652,  3.080905,  3.393465,  3.881718 };
```

```
/*-----------------------------------------------------------------*/
float gaus_d_7bit[64] = { 0.000000,  0.075190,  0.150380,  0.225570,
                    0.300761,  0.375952,  0.451144,  0.526336,
                    0.601530,  0.676724,  0.751919,  0.827115,
                    0.902313,  0.977512,  1.052713,  1.127915,
                    1.203119,  1.278325,  1.353534,  1.428744,
                    1.503957,  1.579173,  1.654392,  1.729613,
                    1.804839,  1.880068,  1.955302,  2.030540,
                    2.105785,  2.181035,  2.256293,  2.331560,
                    2.406838,  2.482128,  2.557433,  2.632758,
                    2.708105,  2.783482,  2.858895,  2.934354,
                    3.009872,  3.085463,  3.161148,  3.236954,
                    3.312915,  3.389074,  3.465489,  3.542234,
                    3.619406,  3.697131,  3.775573,  3.854950,
                    3.935548,  4.017749,  4.102065,  4.189194,
                    4.280103,  4.376160,  4.479365,  4.592769,
                    4.721321,  4.873842,  5.068614,  5.356783 };
float gaus_r_7bit[64] = { 0.037595,  0.112785,  0.187975,  0.263166,
                    0.338357,  0.413548,  0.488740,  0.563933,
                    0.639126,  0.714321,  0.789517,  0.864714,
                    0.939912,  1.015112,  1.090313,  1.165517,
                    1.240722,  1.315929,  1.391138,  1.466350,
                    1.541564,  1.616781,  1.692002,  1.767225,
                    1.842453,  1.917684,  1.992920,  2.068161,
                    2.143408,  2.218662,  2.293925,  2.369196,
                    2.444479,  2.519776,  2.595090,  2.670425,
                    2.745786,  2.821179,  2.896612,  2.972097,
                    3.047646,  3.123279,  3.199017,  3.274891,
                    3.350938,  3.427209,  3.503769,  3.580700,
                    3.658113,  3.736149,  3.814997,  3.894903,
                    3.976193,  4.059304,  4.144825,  4.233563,
                    4.326643,  4.425677,  4.533053,  4.652484,
                    4.790159,  4.957525,  5.179703,  5.533863 };
/*-----------------------------------------------------------------*/
float gaus_d_8bit[128] = {0.000000,  0.069396,  0.138791,  0.208187,
                    0.277583,  0.346979,  0.416376,  0.485773,
                    0.555170,  0.624568,  0.693966,  0.763365,
                    0.832765,  0.902166,  0.971567,  1.040970,
                    1.110373,  1.179777,  1.249183,  1.318590,
                    1.387997,  1.457407,  1.526817,  1.596229,
                    1.665642,  1.735057,  1.804474,  1.873892,
                    1.943312,  2.012734,  2.082158,  2.151584,
                    2.221011,  2.290441,  2.359873,  2.429307,
                    2.498743,  2.568182,  2.637623,  2.707067,
                    2.776513,  2.845962,  2.915413,  2.984867,
                    3.054324,  3.123784,  3.193246,  3.262712,
                    3.332181,  3.401652,  3.471127,  3.540606,
                    3.610087,  3.679572,  3.749061,  3.818552,
                    3.888048,  3.957547,  4.027050,  4.096557,
                    4.166067,  4.235581,  4.305100,  4.374622,
                    4.444149,  4.513679,  4.583214,  4.652753,
                    4.722297,  4.791845,  4.861398,  4.930955,
                    5.000517,  5.070083,  5.139654,  5.209230,
                    5.278811,  5.348397,  5.417988,  5.487584,
                    5.557186,  5.626792,  5.696404,  5.766021,
                    5.835644,  5.905272,  5.974906,  6.044545,
                    6.114190,  6.183841,  6.253498,  6.323161,
```

```
                    6.392829,  6.462504,  6.532185,  6.601872,
                    6.671566,  6.741265,  6.810972,  6.880684,
                    6.950404,  7.020131,  7.089864,  7.159606,
                    7.229355,  7.299113,  7.368880,  7.438659,
                    7.508452,  7.578261,  7.648093,  7.717955,
                    7.787859,  7.857825,  7.927881,  7.998070,
                    8.068460,  8.139155,  8.210313,  8.282181,
                    8.355146,  8.429815,  8.507171,  8.588835,
                    8.677621,  8.778801,  8.903703,  9.084786 };
float gaus_r_8bit[128] = {0.034698,  0.104093,  0.173489,  0.242885,
                    0.312281,  0.381677,  0.451074,  0.520471,
                    0.589869,  0.659267,  0.728666,  0.798065,
                    0.867465,  0.936866,  1.006268,  1.075671,
                    1.145075,  1.214480,  1.283886,  1.353293,
                    1.422702,  1.492111,  1.561523,  1.630935,
                    1.700350,  1.769765,  1.839183,  1.908602,
                    1.978023,  2.047446,  2.116870,  2.186297,
                    2.255726,  2.325156,  2.394589,  2.464025,
                    2.533462,  2.602902,  2.672344,  2.741789,
                    2.811237,  2.880687,  2.950139,  3.019595,
                    3.089053,  3.158514,  3.227978,  3.297445,
                    3.366916,  3.436389,  3.505866,  3.575346,
                    3.644829,  3.714315,  3.783806,  3.853299,
                    3.922797,  3.992298,  4.061802,  4.131311,
                    4.200823,  4.270340,  4.339860,  4.409384,
                    4.478913,  4.548446,  4.617983,  4.687524,
                    4.757070,  4.826620,  4.896175,  4.965734,
                    5.035299,  5.104867,  5.174441,  5.244020,
                    5.313603,  5.383192,  5.452785,  5.522384,
                    5.591988,  5.661597,  5.731211,  5.800831,
                    5.870457,  5.940088,  6.009724,  6.079366,
                    6.149014,  6.218668,  6.288328,  6.357994,
                    6.427665,  6.497343,  6.567027,  6.636717,
                    6.706414,  6.776117,  6.845826,  6.915543,
                    6.985266,  7.054996,  7.124733,  7.194478,
                    7.264232,  7.333994,  7.403767,  7.473552,
                    7.543352,  7.613171,  7.683015,  7.752894,
                    7.822824,  7.892826,  7.962935,  8.033205,
                    8.103716,  8.174593,  8.246032,  8.318330,
                    8.391961,  8.467670,  8.546672,  8.630998,
                    8.724244,  8.833358,  8.974048,  9.195524 };
/*-------------------------------------------------------------------*/
int QGAUS(float xaq, char nbit)
{
  int xq,i;
  float x,xabs;
  /*-------------------------*/
  if(nbit == 0) xq = 0;
  /*-------------------------*/
  if(nbit == 1) {
      if(xaq < 0) xq = 0;
      else xq = 1;
   }
  /*-------------------------*/

  if(nbit == 2) {
     x = xaq;
```

```
    if(x < -0.98183)                    xq = -2;
    else if(x >= -0.98183 && x < 0.0)   xq = -1;
    else if(x >=  0.0 && x < 0.98183)   xq =  0;
    if(x > 0.98183)                     xq =  1;
  }
 /*-------------------------*/
 if(nbit == 3) {
   x = xaq;
   xabs = fabs(x);
   if(xabs >= 1.749115)  xq = 3;
   else {
     for(i = 0; i < 3; i++) {
      if(xabs >= gaus_d_3bit[i] && xabs < gaus_d_3bit[i+1])
        {  xq = i;  break; }
      }
    }
   if(x < 0)     xq = -xq - 1;
  }
 /*-------------------------*/
 if(nbit == 4) {
   x = xaq;
   xabs = fabs(x);
   if(xabs >= 2.403336)  xq = 7;
   else  {
     for(i = 0; i < 7; i++) {
      if(xabs >= gaus_d_4bit[i] && xabs < gaus_d_4bit[i+1])
        {  xq = i;  break; }
      }
    }
   if(x < 0)     xq = -xq - 1;
  }
 /*-------------------------*/
 if(nbit == 5) {
   x = xaq;
   xabs = fabs(x);
   if(xabs >= 2.98066)  xq = 15;
   else {
     for(i = 0; i < 15; i++) {
      if(xabs >= gaus_d_5bit[i] && xabs < gaus_d_5bit[i+1])
        {  xq = i;  break; }
      }
    }
   if(x < 0)     xq = -xq - 1;
  }
 /*-------------------------*/
 if(nbit == 6) {
   x = xaq;
   xabs = fabs(x);
   if(xabs >= 3.637592)  xq = 31;
   else {
     for(i = 0; i < 31; i++) {
      if(xabs >= gaus_d_6bit[i] && xabs < gaus_d_6bit[i+1])

        {  xq = i;  break; }
      }
    }
   if(x < 0)     xq = -xq - 1;
```

```
   }
  /*-------------------------*/
  if(nbit == 7) {
    x = xaq;
    xabs = fabs(x);
    if(xabs >= 5.356783)  xq = 63;
    else {
      for(i = 0; i < 63; i++) {
       if(xabs >= gaus_d_7bit[i] && xabs < gaus_d_7bit[i+1])
         {  xq = i;  break; }
       }
     }
    if(x < 0)     xq = -xq - 1;
   }
  /*-------------------------*/
  if(nbit == 8) {
    x = xaq;
    xabs = fabs(x);
    if(xabs >= 9.084786)  xq = 127;
    else  {
      for(i = 0; i < 127; i++) {
       if(xabs >= gaus_d_8bit[i] && xabs < gaus_d_8bit[i+1])
         {  xq = i;  break; }
       }
     }
    if(x < 0)     xq = -xq - 1;
   }
  /*-------------------------*/
  return(xq);
}
/*-----------------------------------------------------------------*/
/*-----------------------------------------------------------------*/
float DQGAUS(int x, char nbit)
{
  int i,xabs;
  float xq;
  /*-------------------------*/
  if(nbit == 0) xq = 0.0;
  /*-------------------------*/
  if(nbit == 1) {
      if(x==0)  xq = -0.797885;
      else      xq =  0.797885;
   }
  /*-------------------------*/
  if(nbit == 2) {
     if(x == -2)       xq = -1.510729;
     else if(x == -1)  xq = -0.452931;
     else if(x ==  0)  xq =  0.452931;
     else if(x ==  1)  xq =  1.510729;
     /*xq = xq/1.510729;*/
   }

  /*-------------------------*/
  if(nbit == 3) {
    xabs = abs(x);
    if(x < 0)  xabs = xabs - 1;
    xq = gaus_r_3bit[xabs];
```

```
    /*for(i = 0; i < 4; i++) {
      if(xabs == i)  {  xq = gaus_r_3bit[i];   break; }
       }*/
    if(x < 0)     xq = -xq;
    /*xq = xq/gaus_r_3bit[3];*/
   }
  /*-------------------------*/
  if(nbit == 4) {
    xabs = abs(x);
    if(x < 0)  xabs = xabs - 1;
    xq = gaus_r_4bit[xabs];
    /*for(i = 0; i < 8; i++) {
       if(xabs == i)  {  xq = gaus_r_4bit[i];  break; }
       }*/
    if(x < 0)     xq = -xq;
    /*xq = xq/gaus_r_4bit[7];*/
   }
  /*-------------------------*/
  if(nbit == 5) {
    xabs = abs(x);
    if(x < 0)  xabs = xabs - 1;
    xq = gaus_r_5bit[xabs];
    /*for(i = 0; i < 16; i++) {
      if(xabs == i)  {  xq = gaus_r_5bit[i];  break; }
      }*/
    if(x < 0)     xq = -xq;
    /*xq = xq/gaus_r_5bit[15];*/
   }
  /*-------------------------*/
  if(nbit == 6) {
    xabs = abs(x);
    if(x < 0)  xabs = xabs - 1;
    xq = gaus_r_6bit[xabs];
    /*for(i = 0; i < 32; i++) {
      if(xabs == i)  {  xq = gaus_r_6bit[i];  break; }
     }*/
    if(x < 0)     xq = -xq;
    /*xq = xq/gaus_r_6bit[31];*/
   }
  /*-------------------------*/
  if(nbit == 7) {
    xabs = abs(x);
    if(x < 0)  xabs = xabs - 1;
    xq = gaus_r_7bit[xabs];
    /*for(i = 0; i < 64; i++) {
      if(xabs == i)  {  xq = gaus_r_7bit[i];  break; }
      }*/
    if(x < 0)     xq = -xq;
    /*xq = xq/gaus_r_7bit[63];*/
   }

  /*-------------------------*/
  if(nbit == 8) {
    xabs = abs(x);
    if(x < 0)  xabs = xabs - 1;
    xq = gaus_r_8bit[xabs];
    /*for(i = 0; i < 128; i++) {
```

```
      if(xabs == i)  {  xq = gaus_r_8bit[i];  break; }
      }*/
    if(x < 0)     xq = -xq;
    /*xq = xq/gaus_r_8bit[127];*/
   }
  /*-------------------------*/
  return(xq);
}
/*QUANTMAX.C  Quantizadores Gauss e Laplace baseado no processo de MAX */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <io.h>
#include <string.h>
#include <process.h>
#include <math.h>
#include <alloc.h>

#include "maniparq.h"
/*-----------------------------------------------------------------------*/
float lapl_d_1bit[1] = { 0.000000 };
float lapl_r_1bit[1] = { 0.701778 };
/*-----------------------------------------------------------------------*/
float lapl_d_2bit[2]={ 0.000000, 1.120161 };
float lapl_r_2bit[2]={ 0.418046, 1.822276 };
/*-------------------------------------------------------------*/
float lapl_d_3bit[4] = { 0.000000,  0.531984,  1.249229,  2.369937 };
float lapl_r_3bit[4] = { 0.232955,  0.831013,  1.667445,  3.072430 };
/*-------------------------------------------------------------*/
float lapl_d_4bit[8] = { 0.000000,  0.266270,  0.570701,  0.925900,
                  1.352340,  1.886731,  2.606046, 3.728556 };

float lapl_r_4bit[8] = { 0.124832,  0.407708,  0.733693,  1.118107,
                  1.586573,  2.186890,  3.025201, 4.431911 };
/*-------------------------------------------------------------*/
float lapl_d_5bit[16] = {0.000000,  0.133219,  0.275338,  0.427593,
                  0.591508,  0.768994,  0.962486,  1.175163,
                  1.411277,  1.676701,  1.979871,  2.333536,
                  2.758341,  3.291191,  4.009276,  5.131228 };

float lapl_r_5bit[16] = {0.064524,  0.201914,  0.348761,  0.506424,
                  0.676592,  0.861396,  1.063577,  1.286749,
                  1.535805,  1.817597,  2.142144,  2.524927,
                  2.991754,  3.590627,  4.427926,  5.834531 };
/*-------------------------------------------------------------*/
float lapl_d_6bit[32] = {0.000000,  0.079320,  0.161594,  0.246925,
                  0.335422,  0.427207,  0.522415,  0.621197,
                  0.723725,  0.830194,  0.940829,  1.055894,
                  1.175694,  1.300592,  1.431018,  1.567486,
                  1.710616,  1.861163,  2.020051,  2.188428,
                  2.367730,  2.559782,  2.766943,  2.992314,
                  3.240080,  3.516050,  3.828602,  4.190438,
                  4.622165,  5.160668,  5.883158,  7.008477 };

float lapl_r_6bit[32] = {0.038933,  0.119706,  0.203482,  0.290367,
                  0.380477,  0.473938,  0.570893,  0.671502,
```

```
                  0.775948,  0.884439,  0.997219,  1.114569,
                  1.236820,  1.364365,  1.497671,  1.637300,
                  1.783932,  1.938394,  2.101709,  2.275148,
                  2.460312,  2.659252,  2.874633,  3.109995,
                  3.370165,  3.661935,  3.995269,  4.385607,
                  4.858724,  5.462612,  6.303703,  7.713250 };
/*-------------------------------------------------------------*/
float lapl_d_7bit[64] = {0.000000,  0.076822,  0.156370,  0.238686,
                  0.323806,  0.411770,  0.502614,  0.596374,
                  0.693086,  0.792784,  0.895500,  1.001269,
                  1.110120,  1.222086,  1.337196,  1.455479,
                  1.576964,  1.701678,  1.829648,  1.960901,
                  2.095461,  2.233353,  2.374603,  2.519233,
                  2.667266,  2.818724,  2.973630,  3.132006,
                  3.293872,  3.459249,  3.628159,  3.800622,
                  3.976659,  4.156293,  4.339545,  4.526440,
                  4.717004,  4.911264,  5.109255,  5.311012,
                  5.516581,  5.726016,  5.939384,  6.156770,
                  6.378280,  6.604054,  6.834273,  7.069171,
                  7.309061,  7.554356,  7.805606,  8.063549,
                  8.329182,  8.603860,  8.889441,  9.188509,
                  9.504715,  9.843333, 10.212215, 10.623559,
                 11.097513, 11.670579, 12.419733, 13.564157 };

float lapl_r_7bit[64] = {0.037734,  0.115909,  0.196831,  0.280540,
                  0.367072,  0.456467,  0.548760,  0.643988,
                  0.742184,  0.843383,  0.947618,  1.054920,
                  1.165321,  1.278851,  1.395541,  1.515418,
                  1.638510,  1.764846,  1.894451,  2.027351,
                  2.163571,  2.303136,  2.446070,  2.592395,
                  2.742136,  2.895313,  3.051948,  3.212064,
                  3.375680,  3.542818,  3.713499,  3.887744,
                  4.065574,  4.247011,  4.432079,  4.620801,
                  4.813206,  5.009323,  5.209186,  5.412837,
                  5.620324,  5.831707,  6.047061,  6.266479,
                  6.490082,  6.718027,  6.950518,  7.187824,
                  7.430298,  7.678413,  7.932798,  8.194300,
                  8.464065,  8.743655,  9.035226,  9.341792,
                  9.667639, 10.019027, 10.405402, 10.841716,
                 11.353311, 11.987846, 12.851621, 14.276693 };
/*-------------------------------------------------------------*/
float lapl_d_8bit[128] ={0.000000,   0.075969,   0.154606,   0.235948,
                  0.320032,   0.406897,   0.496576,   0.589105,
                  0.684519,   0.782849,   0.884129,   0.988388,
                  1.095659,   1.205970,   1.319350,   1.435828,
                  1.555431,   1.678184,   1.804114,   1.933246,
                  2.065604,   2.201212,   2.340093,   2.482268,
                  2.627760,   2.776589,   2.928777,   3.084341,
                  3.243303,   3.405680,   3.571491,   3.740753,
                  3.913484,   4.089699,   4.269415,   4.452647,
                  4.639412,   4.829722,   5.023593,   5.221039,
                  5.422073,   5.626708,   5.834956,   6.046831,
                  6.262344,   6.481506,   6.704330,   6.930825,
                  7.161003,   7.394874,   7.632448,   7.873735,
                  8.118744,   8.367484,   8.619965,   8.876194,
                  9.136182,   9.399934,   9.667461,   9.938768,
                 10.213865,  10.492757,  10.775453,  11.061958,
```

```
                  11.352280,  11.646426,  11.944400,  12.246210,
                  12.551860,  12.861358,  13.174708,  13.491916,
                  13.812986,  14.137924,  14.466735,  14.799423,
                  15.135993,  15.476448,  15.820794,  16.169034,
                  16.521172,  16.877212,  17.237157,  17.601010,
                  17.968776,  18.340456,  18.716054,  19.095572,
                  19.479013,  19.866380,  20.257675,  20.652900,
                  21.052057,  21.455147,  21.862173,  22.273136,
                  22.688038,  23.106878,  23.529660,  23.956383,
                  24.387048,  24.821655,  25.260207,  25.702701,
                  26.149140,  26.599524,  27.053853,  27.512130,
                  27.974356,  28.440538,  28.910683,  29.384808,
                  29.862939,  30.345122,  30.831434,  31.322007,
                  31.817058,  32.316961,  32.822337,  33.334241,
                  33.854453,  34.386015,  34.934176,  35.508204,
                  36.125082,  36.818089,  37.661231,  38.871844 };

float lapl_r_8bit[128] ={0.037323,   0.114616,   0.194596,   0.277300,
                   0.362765,   0.451028,   0.542124,   0.636087,
                   0.732951,   0.832748,   0.935509,   1.041267,
                   1.150051,   1.261889,   1.376812,   1.494845,
                   1.616016,   1.740352,   1.867877,   1.998616,
                   2.132593,   2.269832,   2.410354,   2.554182,
                   2.701338,   2.851841,   3.005712,   3.162971,
                   3.323636,   3.487725,   3.655257,   3.826249,
                   4.000718,   4.178680,   4.360150,   4.545145,
                   4.733678,   4.925766,   5.121421,   5.320657,
                   5.523488,   5.729927,   5.939986,   6.153676,
                   6.371011,   6.592001,   6.816658,   7.044992,
                   7.277014,   7.512734,   7.752162,   7.995308,
                   8.242180,   8.492788,   8.747141,   9.005248,
                   9.267116,   9.532753,   9.802168,  10.075369,
                  10.352361,  10.633153,  10.917752,  11.206164,
                  11.498396,  11.794455,  12.094345,  12.398074,
                  12.705647,  13.017069,  13.332347,  13.651484,
                  13.974488,  14.301361,  14.632109,  14.966737,
                  15.305249,  15.647648,  15.993940,  16.344128,
                  16.698216,  17.056208,  17.418106,  17.783915,
                  18.153637,  18.527275,  18.904832,  19.286312,
                  19.671715,  20.061045,  20.454305,  20.851495,
                  21.252618,  21.657676,  22.066670,  22.479602,
                  22.896473,  23.317284,  23.742036,  24.170730,
                  24.603366,  25.039945,  25.480468,  25.924935,
                  26.373346,  26.825702,  27.282004,  27.742255,
                  28.206457,  28.674618,  29.146748,  29.622868,
                  30.103011,  30.587234,  31.075635,  31.568378,
                  32.065739,  32.568183,  33.076492,  33.591989,
                  34.116917,  34.655112,  35.213239,  35.803169,
                  36.446996,  37.189181,  38.133280,  39.610408 };
/*-------------------------------------------------------------------*/
int QLAPLAC(float xaq, char nbit)
{
  int xq,i;
  float x,xfabs;
  /*-------------------------*/
  if(nbit == 0) xq = 0;
  /*-------------------------*/
```

```
  if(nbit == 1) {
      if(xaq < 0) xq = 0;
      else xq = 1;
   }   /*-------------------------*/
  if(nbit == 2) {
     x = xaq;
     if(x < -1.120161)                    xq = -2;
     else if(x >= -1.120161 && x < 0.0)   xq = -1;
     else if(x >=  0.0 && x < 1.120161)   xq =  0;
     if(x > 1.120161)                     xq =  1;
   }  /*-------------------------*/
  if(nbit == 3) {
    x = xaq;
    xfabs = fabs(x);
    if(xfabs >= 2.369937)  xq = 3;
    else {
      for(i = 0; i < 3; i++) {
       if(xfabs >= lapl_d_3bit[i] && xfabs < lapl_d_3bit[i+1])
         {  xq = i;  break; }
       }
     }
    if(x < 0)     xq = -xq - 1;
   }  /*-------------------------*/
  if(nbit == 4) {
    x = xaq;
    xfabs = fabs(x);
    if(xfabs >= 3.728556)  xq = 7;
    else  {
      for(i = 0; i < 7; i++) {
       if(xfabs >= lapl_d_4bit[i] && xfabs < lapl_d_4bit[i+1])
         {  xq = i;  break; }
       }
     }
    if(x < 0)     xq = -xq - 1;
   }
  /*-------------------------*/
  if(nbit == 5) {
    /*x = xaq * lapl_d_5bit[15];*/
    x = xaq;
    xfabs = fabs(x);
    if(xfabs >= 5.131228)  xq = 15;
    else {

      for(i = 0; i < 15; i++) {
       if(xfabs >= lapl_d_5bit[i] && xfabs < lapl_d_5bit[i+1])

         {  xq = i;  break; }
       }
     }
    if(x < 0)     xq = -xq - 1;
   }  /*-------------------------*/
  if(nbit == 6) {
    x = xaq;
    xfabs = fabs(x);
    if(xfabs >= 7.008477)  xq = 31;
    else {
      for(i = 0; i < 31; i++) {
```

```
       if(xfabs >= lapl_d_6bit[i] && xfabs < lapl_d_6bit[i+1])
         {  xq = i;  break; }
       }
     }
    if(x < 0)     xq = -xq - 1;
   }  /*-------------------------*/
  if(nbit == 7) {
    x = xaq;
    xfabs = fabs(x);
    if(xfabs >= 13.564157)  xq = 63;
    else {
      for(i = 0; i < 63; i++) {
       if(xfabs >= lapl_d_7bit[i] && xfabs < lapl_d_7bit[i+1])
         {  xq = i;  break; }
       }
     }
    if(x < 0)     xq = -xq - 1;
   }  /*-------------------------*/
  if(nbit == 8) {
    x = xaq;
    xfabs = fabs(x);
    if(xfabs >= 38.871844)  xq = 127;
    else  {
      for(i = 0; i < 127; i++) {
       if(xfabs >= lapl_d_8bit[i] && xfabs < lapl_d_8bit[i+1])
         {  xq = i;  break; }
       }
     }
    if(x < 0)     xq = -xq - 1;
   }
  return(xq);
}
/*---------------------------------------------------------------------*/
float DQLAPLAC(int x, char nbit)
{
  int i,xabs;
  float xq;
  /*-------------------------*/
  if(nbit == 0) xq = 0.0;
  /*-------------------------*/
  if(nbit == 1) {
      if(x==0)  xq = -0.5;
      else      xq =  0.5;
   }
  /*-------------------------*/
  if(nbit == 2) {
     if(x == -2)       xq = -1.822276;
     else if(x == -1)  xq = -0.418046;
     else if(x ==  0)  xq =  0.418046;
     else if(x ==  1)  xq =  1.822276;
   }
  /*-------------------------*/
  if(nbit == 3) {
    xabs = abs(x);
    if(x < 0)  xabs = xabs - 1;
    xq = lapl_r_3bit[xabs];
    if(x < 0)     xq = -xq;
```

```
   }
  /*--------------------------*/
  if(nbit == 4) {
    xabs = abs(x);
    if(x < 0)  xabs = xabs - 1;
    xq = lapl_r_4bit[xabs];
    if(x < 0)     xq = -xq;
   }
  /*--------------------------*/
  if(nbit == 5) {
    xabs = abs(x);
    if(x < 0)  xabs = xabs - 1;
    xq = lapl_r_5bit[xabs];
    if(x < 0)     xq = -xq;
   }
  /*--------------------------*/
  if(nbit == 6) {
    xabs = abs(x);
    if(x < 0)  xabs = xabs - 1;
    xq = lapl_r_6bit[xabs];
    if(x < 0)     xq = -xq;
   }
  /*--------------------------*/
  if(nbit == 7) {
    xabs = abs(x);
    if(x < 0)  xabs = xabs - 1;
    xq = lapl_r_7bit[xabs];
    if(x < 0)     xq = -xq;
   }
  /*--------------------------*/
  if(nbit == 8) {
    xabs = abs(x);
    if(x < 0)  xabs = xabs - 1;
    xq = lapl_r_8bit[xabs];
    if(x < 0)     xq = -xq;
   }
  /*--------------------------*/
  return(xq);
}
```

```
/*-------------------------------------------------------------------
QUANTIZA.C Rotinas para quantização dos coeficientes das TRANSFORMAS
    COSSENO OU WALSH. Como dado de entrada deve ser fornecido a imagem
    com os coeficientes transformados e gravados por bloco, como os
    mesmos foram processados.
       Esse programa eh formado de funções que calculam a media "mi" e
    a media quadrada "mi2" e o desvio padrao "sigma" de cada coeficiente.
       As medias de cada coeficiente eh guardada em um vetor de media
    mi[] e os desvios padrões em um vetor sigma[]. Cada elemento dos
    vetores mi[] e sigma[] correspondem a um coeficiente das
    transformadas.
       Eh determinado o histograma da imagem, assumindo que todos os
    coeficientes podem ser modelados pela mesma Função Densidade de Pro-
    babilidade. Através da formula [(y-mi)/sigma] eh forcado que todos os
    coeficientes tenham distribuição identica com media e desvio padrao
    iguais a zero, resultando em identicos niveis de decisao e de recons-
    trução para todos os coeficientes escalados. Isso forca que todos os
    coef. sejam representados pelo mesmo número de bits. Portanto, um
    unico vetor d[] e r[] pode ser usado para a quantizados dos coef. O
    histograma eh construido baseado na distribuição de cada amostra.
       Eh procurado o valor da minima e da maxima distribuição presente
    na imagem. Esses valores Min e Max sao usados para a determinação
    dos niveis de decisao d[] e de reconstrução r[] através da altura h
    de cada passo (como: h=Max-Min, d[0]=Min, d[N]=Max, onde N eh o
    número de niveis de quantização adotado) Os niveis d[] sao usados
    como base mara a construção do histograma dos coeficientes.
       Como eh forcado que todos os coeficientes tenham a mesma distri-
    buição, entao, eh calculado a media total miT e desvio padrao total
    sigmaT de todos os coeficientes juntos, para a montagem do quanti-
    zador que sera usado. As amostras d[] e r[] do quantizador, terao
    seus  niveis ajustados pelo desvio padrao sigmaT. Normalizando assim
    os niveis de todos os coeficientes para o quant. de Lloyd_Max.
-------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <io.h>
#include <string.h>
#include <process.h>
#include <math.h>
#include <alloc.h>
#include "maniparq.h"
#define pi 3.141592654
/*-------------------------------------------------------------------
media_sigma() Calcula a media[] e o desvio padrao sigma[] de cada coefici
          ente da transformada armazenando oresultado em cada indice dos
          vetores acima. Essa função recebe o arquivo completo contendo
            os coeficientes da transformada como foram calculados e a
            partir    deles calcula suas medias e sigmas. Primeiramente
            eh calculado a soma e a soma quadrada de cada coeficiente.
            soma[i] = {somatorio de x[i]}
            soma_2[i] = {somatorio de x[i]^2}
            mi[i] = {soma[i]/NtBl}

      sigma[i] = sqrt({NtBl*soma_2[i] -(soma[i]^2)}/{NtBl*(NtBl-1)})
```

```
void media_sigma(ARQUIVO *arq_in,int linha,int coluna,int NB, \\
                 float*media,float *sigma)
-----------------------------------------------------------------------*/
/*-----------------------------------------------------------------------*/
void media_sigma(ARQUIVO *arq_in,int linha,int coluna,int NB, \\
       float *media,float *sigma)
  {
   int i,j,k,NBh,NBv,NaB;
   int lin,col;
   float **x,*soma,*soma_2,NtBl,denom;
   MATRIX *XX;

     NaB = NB*NB;
     NBv = linha/NB;
     NBh = coluna/NB,
     NtBl = (float)(NBh*NBv);
     denom = NtBl*(NtBl-1.0);
     if(denom==0.0) denom = 1.0;

     soma = (float *)calloc(NaB,sizeof(float));
     soma_2 = (float *)calloc(NaB,sizeof(float));
     XX = aloca_mat(NB,coluna,sizeof(float));
     x = (float **)XX->ptr_mat;

     for(lin = 0; lin < linha; lin+=NB) {
      for(i = 0; i < NB; i++)
         ler_registro((char *)XX-
>ptr_mat[i],arq_in,coluna,sizeof(float));
      for(col = 0; col < coluna; col+=NB) {
          k = 0;
          for(i = 0; i < NB; i++) {
            for(j = 0; j < NB; j++) {
                soma[k]+=x[i][col+j];
                soma_2[k]+=x[i][col+j]*x[i][col+j];
                k++;
              }
            }
        }
       }
     rewind(arq_in->ptr_arq);
     for(k = 0; k < NaB; k++) {
       media[k] = soma[k]/NtBl;
       sigma[k] = fabs(((NtBl*soma_2[k])-(soma[k]*soma[k]))/(denom));
       sigma[k] = (float)sqrt((double)sigma[k]);
       if(sigma[k]==0.0)  sigma[k] = 1.0e-10;
       }
     free(soma);
     free(soma_2);
     libera_mat(XX);
  }
/*-----------------------------------------------------------------------*/
```

```
/*-------------------------------------------------------------------
aloca_bit() Função para a alocação do número medio de bits para o bloco
de coeficientes da transformada.

void aloca_bit(unsigned int NT,float B,float *sigma2,int *b)
-------------------------------------------------------------------*/
/*-------------------------------------------------------------------*/
void aloca_bit(unsigned int NT,float B,float *sigma2,int *b)
 {
   int k,Bs,sig1,sig2;
   float teta,erro,xn,xp;
   float prod,sigma_med;

   teta = 1.0e-10;
   Bs = bitaloc(NT,teta,sigma2,b);
   erro = (float)Bs - B;
   sig1 = (erro > 0.0)?1:-1;
   if(erro > 0.0)  xp = teta;
   else xn = teta;
   teta*=10;
   while(1) {
      Bs = bitaloc(NT,teta,sigma2,b);
      erro = (float)Bs - B;
      sig2 = (erro > 0.0)?1:-1;
      if((sig1+sig2)==0)
         break;
      teta*=10;
    }
   if(erro > 0.0)  xp = teta;
   else xn = teta;
   k = 0;
   while(1) {
      k++;
      teta = (xn+xp)/2.0;
      Bs = bitaloc(NT,teta,sigma2,b);
      erro = (float)Bs - B;
      if((fabs(erro)<0.5)||(k>=200)) {
        if(b[0]<8)  b[0]+=1;
        if(b[0]>8)  b[0] = 8;
        break;
      }
      if(erro > 0.0)  xp = teta;
      else xn = teta;
     }

     /*printf("\nBs = %d ",Bs);
     printf(" teta = %f ",teta);
     erro = B - (float)Bs;
     printf("\nErro de Alocação de Bits = %f bits",erro);
     Bs = 0;  */
 }
/*-------------------------------------------------------------------*/
```

```
/*-------------------------------------------------------------------
bitaloc() Função para calcular o número otimo de bits para cada
coeficientedo bloco da transformada.

int bitaloc(unsigned int NT,float teta,float *sigma2,int *b)
-------------------------------------------------------------------*/
/*-------------------------------------------------------------------*/
int bitaloc(unsigned int NT,float teta,float *sigma2,int *b)
{
   int i,Bs;
   float bits;

   b[0] = 7;
   for(i = 1; i < NT; i++) {
       if(sigma2[i]==0.0)  sigma2[i] = 1.0e-10;
       bits = 0.5*log10((double)(sigma2[i]/teta))/log10(2.0);
       if(bits>8.0)  bits = 8.0;
       b[i] = (bits >0)?(int)bits:0;
    }
   Bs = 0;
   for(i = 0; i < NT; i++)
       Bs+=b[i];
   return(Bs);
}
/*-------------------------------------------------------------------*/
/*-------------------------------------------------------------------
Min_MAx() Calcula a Menor e a Maior Distribuição presente em todos os
       coeficientes da transformada armazenando o resultado em Min e Max.
       Esses valores servirao para determinar a altura h do quantizador
       de Lloyd-Max e seus niveis de decisao d[] e de reconstrução r[].
       Servirao tambem como referencia para a determinação do histograma
       dos coeficientes.

            Distrib = (x[i][j]-media[k])/sigma[k]

void Min_Max(ARQUIVO *arq_in,int linha,int coluna,int NB,float *media, \
float *sigma,float *Min,float *Max)
-------------------------------------------------------------------*/
/*-------------------------------------------------------------------*/
void Min_Max(ARQUIVO *arq_in,int linha,int coluna,int NB,float *media, \
float *sigma,float *Min,float *Max)
  {
    int i,j,k,lin,col;
    float **x,Distrib;
    MATRIX *XX;

      XX = aloca_mat(NB,coluna,sizeof(float));
       x = (float **)XX->ptr_mat;

     for(k = 0; k < NB*NB; k++) {
       Min[k] = 1.0e10;
       Max[k] = -1.0e10;
      }
     for(lin = 0; lin < linha; lin+=NB) {
      for(i = 0; i < NB; i++)
          ler_registro((char *)XX-
>ptr_mat[i],arq_in,coluna,sizeof(float));
```

```
      for(col = 0; col < coluna; col+=NB) {
          k = 0;

          for(i = 0; i < NB; i++) {
            for(j = 0; j < NB; j++) {
                Distrib = (x[i][col+j]-media[k])/sigma[k] ;
                if(Distrib<Min[k])   Min[k]=Distrib;
                if(Distrib>Max[k])   Max[k]=Distrib;
                k++;
              }
            }
        }
       }
    rewind(arq_in->ptr_arq);
    libera_mat(XX);
  }
/*-------------------------------------------------------------------*/
/*-------------------------------------------------------------------
Gera_d_r() Gera os niveis de Decisao d[] e de Reconstrução r[] a partir
       das Maxima e Minima Distribuição presentes nos coeficientes.
       Primeiramente eh encontrado a altura h da distribuição, em seguida
       os niveis d[] e r[]. A Distribuição e calculada como a formula
       abaixo e a altura eh dada por h = (Max-Min). Cada passo de decisao
       eh dado por: delta = h/M, onde M eh o número de pontos de dados
       usados paga a geração do histograma.

            Distrib = (x[i][j]-media[k])/sigma[k]

void Gera_d_r(float **d,float **r,float *Min,float *Max,int *b,int NT)
-------------------------------------------------------------------*/
/*-------------------------------------------------------------------*/
void Gera_d_r(float **d,float **r,float *Min,float *Max,int *b,int NT)
   {
     int i,k,Nn;
     float h;

       for(i = 0; i < NT; i++) {
        if(b[i] > 0) {
           Nn = (int)pow(2,b[i]);
           h = (Max[i] - Min[i])/(float)Nn;
           d[i][0] = Min[i];  d[i][Nn] = Max[i];
           r[i][0] = d[i][0] + h/2.0;
           for(k = 1; k < Nn; k++) {
            d[i][k] = d[i][k-1] + h;
            r[i][k] = d[i][k] + h/2.0;
            }
         }
      }
   }
/*-------------------------------------------------------------------*/
/*-------------------------------------------------------------------
Histograma() Gera o histograma apenas dos coeficientes escolhidos e
   marcados pela matiz T[]. Para a geração desse histograma, eh
considerado que todos os coeficientes podem ser modelados pela mesma PDF,
para isso, eh forcado que todos os coeficientes tenham Distribuição
Identicas com media zero e desvio padrao 1. Isso eh conseguido aplicando
a formula {(y-media[i])/sigma[i]} para todos os coeficientes escolhidos.
```

```
      Resultando em identicos niveis de decisao d[] e de reconstrução r[]

 para todos os coeficientes escaladas, forcando assim que todos os
coeficientes sejam representados pela mesmo número de bits. Portanto,
um unico vetor d[] e r[] pode ser usado para quantizar todos os
coeficientes escolhidos.

void Histograma(ARQUIVO *arq_in,int linha,int coluna,int NB,int *b, \
float *media,float *sigma,float **d,float **histo);
-------------------------------------------------------------------------*/
void Histograma(ARQUIVO *arq_in,int linha,int coluna,int NB,int *b, \
float *media,float *sigma,float **d,float **histo)
   {
     int i,j,k,nd,lin,col,Nn;
     float **x,Distrib;
     MATRIX *XX;

      XX = aloca_mat(NB,coluna,sizeof(float));
       x = (float **)XX->ptr_mat;

     for(k = 0; k < NB*NB; k++) {
       Nn = (int)pow(2,b[k]);
       for(i = 0; i < Nn; i++)
           histo[k][i] = 0.0;
      }
     for(lin = 0; lin < linha; lin+=NB) {
      for(i = 0; i < NB; i++)
         ler_registro((char *)XX-
>ptr_mat[i],arq_in,coluna,sizeof(float));
      for(col = 0; col < coluna; col+=NB) {
          k = 0;
          for(i = 0; i < NB; i++) {
            for(j = 0; j < NB; j++) {
                Nn = (int)pow(2,b[k]);
                if(b[k] == (int)0)  {
                   k++;
                   continue;
                 }
                Distrib = (x[i][col+j]-media[k])/sigma[k] ;
                for(nd = 0; nd < Nn; nd++) {
                  if((Distrib>=d[k][nd])&&(Distrib<d[k][nd+1])) {
                      histo[k][nd]+=1.0;
                      break;
                    }
                  if(nd == Nn-1) {
                           histo[k][nd]+=1.0;
                     break;
                    }
                  }
                k++;
              }
            }
        }
       }
    rewind(arq_in->ptr_arq);
    libera_mat(XX);
   }
```

```
/*-------------------------------------------------------------------
Qantizador() Gera o quantizador da imagem a partir do quantizador de
    Lloyd-Max. Primeiramente eh calculado a media e o desvio padrao de
    todas as amostras selecionadas, baseada no histograma das mesmas. O
    desvio padrao das amostras selecionadas servira para NORMALIZAR os
    coeficientes do quantizador de Lloyd-Max escolhido conforme o
    número de bits de quantização. A Normalização eh feita da seguinte
    forma: Cada nivel de decisao e de reconstrução do quantizador eh
    multiplicado pelo sigma total, gerando um novo quantizador NORMA-
    LIZADO.
                di = di*sigma    dN = -d0   i=0,1...N
                ri = ri*sigma

void Qauntizador(int NB,int lin,int col,float **r,int *b,float **histo,\
float **dqLM,float **rqLM)
-------------------------------------------------------------------*/
void Qauntizador(int NB,int lin,int col,float **r,int *b,float **histo,\
float **dqLM,float **rqLM)
{
   int i,k,Nn,NBh,NBv,NaB;
   float NtBl,denom;
   float *med,*med_2,*sigm;

    NaB = NB*NB;
    NBv = lin/NB;
    NBh = col/NB,
    NtBl = (float)(NBh*NBv);
    denom = NtBl*(NtBl-1.0);
    med = (float *)calloc(NaB,sizeof(float));
    med_2 = (float *)calloc(NaB,sizeof(float));
    sigm = (float *)calloc(NaB,sizeof(float));
    for(k = 0; k < NaB; k++) {
      med[k]=0.0; med_2[k]=0.0; sigm[k]=0.0;
     }
    for(k = 0; k < NaB; k++) {
       if(b[k]>0) {
        Nn = (int)pow(2,b[k]);
        for(i = 0; i < Nn; i++) {
           med[k]+=r[k][i]*histo[k][i];
           med_2[k]+=r[k][i]*r[k][i]*histo[k][i];
         }
      }
     }
     for(k = 0; k < NaB; k++) {
      if(b[k]>0) {
         sigm[k] = (NtBl*med_2[k]-med[k]*med[k])/denom;
         sigm[k] = (float)sqrt((double)sigm[k]);
         if(sigm[k]==0.0)  sigm[k] = 1.0e-10;
       }
      }
    for(k = 0; k < NaB; k++) {
       if(b[k]>0) {
        Nn = (int)pow(2,b[k]);
        for(i = 0; i < Nn; i++) {
           dqLM[k][i]*=sigm[k];

           rqLM[k][i]*=sigm[k];
```

```
         }
        dqLM[k][Nn]*=sigm[k];
      }
     }
  free(med); free(med_2); free(sigm);
}
/*-------------------------------------------------------------------
Qantiza_imagem() Rotina para a quantização da imagem a partir dos dados
    calculados nas rotinas anteriores. Eh usado os niveis de quantização
    dqLM[] e rqLM[] do quantizador NORMALIZADO para quantizar as amostras
    da imagem. Eh quantizado apenas as amostras que tem T[i]=1, as
    amostras que tem T[i]=0 não serao quantizadas.

void Quantiza_imagem(ARQUIVO *arq_in,HEAD_DATA *arq_out,int *b, \
float *media,float *sigma,float **dqLM);
-------------------------------------------------------------------*/
void Quantiza_imagem(ARQUIVO *arq_in,HEAD_DATA *arq_out,int *b, \
float *media,float *sigma,float **dqLM)
{
   int chq,i,j,k,NB,Nn,nd,linha,coluna,lin,col;
   float **x,Distrib;
   MATRIX *XX;

     linha = arq_out->lin_image;
     coluna = arq_out->col_image;
     NB = (int)arq_out->tam_bloco;
     XX = aloca_mat(NB,coluna,sizeof(float));
      x = (float **)XX->ptr_mat;
    for(lin = 0; lin < linha; lin+=NB) {
      for(i = 0; i < NB; i++)
         ler_registro((char *)XX-
>ptr_mat[i],arq_in,coluna,sizeof(float));
      for(col = 0; col < coluna; col+=NB) {
          k = 0;
          for(i = 0; i < NB; i++) {
            for(j = 0; j < NB; j++) {
               Nn = (int)pow(2,b[k]);
               if(b[k] == (int)0)  {
                  k++;
                  continue;
                }
               else
                   Distrib = (x[i][col+j]-media[k])/sigma[k];
               if(Distrib < dqLM[k][1]) chq = (int)0;
               else if(Distrib >= dqLM[k][Nn-1]) chq = (int)(Nn-1);
               else {

                 for(nd = 1; nd < Nn-1; nd++) {
                  if((Distrib>=dqLM[k][nd])&&(Distrib<dqLM[k][nd+1])) {
                      chq = (int)nd;
                      break;
                    }
                  }
                }

        putc(chq,arq_out->fptr);
              k++;
```

```
            }
           }
         }
        }
    rewind(arq_in->ptr_arq);
    libera_mat(XX);
   }
/*-------------------------------------------------------------------------*/
void Desquantiza_imagem(HEAD_DATA *arq_in,ARQUIVO *arq_out,int *b,\
float *media,float *sigma,float **rqLM)
{
    int chq,i,j,k,Nn,NB,nq,linha,coluna,lin,col;
    float xx,**x,Distrib,rq;
    MATRIX *XX;
    char ch;

       linha = arq_in->lin_image;
       coluna = arq_in->col_image;
       NB = (int)arq_in->tam_bloco;
       XX = aloca_mat(NB,coluna,sizeof(float));
       x = (float **)XX->ptr_mat;
       for(lin = 0; lin < linha; lin+=NB) {
         for(col = 0; col < coluna; col+=NB) {
             k = 0;
             for(i = 0; i < NB; i++) {
               for(j = 0; j < NB; j++) {
                   if(b[k] == (int)0)  {
                        /*Nn = (int)pow(2,b[0]);
                        chq = Nn/2;
                        rq = rqLM[0][chq];
                        x[i][col+j] = rq*sigma[k]+media[k];*/
                    x[i][col+j] = media[k];
                    k++;
                    continue;
                  }
                   else  {
                    chq = getc(arq_in->fptr);
                    rq = rqLM[k][chq];
                    xx = rq*sigma[k]+media[k];
                    x[i][col+j] = xx;
                    k++;
                  }
                 }
             }
           }
       for(i = 0; i < NB; i++)
       grava_registro((char *)XX-
>ptr_mat[i],arq_out,coluna,sizeof(float));
     }
    libera_mat(XX);
}
/*-------------------------------------------------------------------------*/
```

```
/*****************************************************************
R_W_MAT.C   ler ou escreve uma matriz em um arquivo aberto no modo
         binario com estrutura ARQUIVO
 *****************************************************************
ler_mat      ler uma matriz de um arquivo aberto no modo binario
grava_mat    grava uma matriz em um arquivo aberto no modo binario
mostra_mat   mostra uma matriz no display
 *****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "maniparq.h"
/*****************************************************************
ler_mat   ler uma matriz de um arquivo com estrutura ARQUIVO. Provoca
        exit(1) se ocorrer um erro de leitura ou se a estrutura for
        invalida. Retorna um ponteiro para estrutura MATRIX.

MATRIX *ler_mat(char *nome_arq,int linha,int coluna,unsigned char tipo)
*****************************************************************/
MATRIX *ler_mat(char *nome_arq,int linha,int coluna,unsigned char tipo)
   {
    MATRIX *aloca_mat();
    MATRIX *AAA;
    ARQUIVO *arquivo;
    int i,tamanho_elem;
    int tamanho;
    double *buf;
    arquivo = abre_ler(nome_arq);
    if(!arquivo) {
       return(NULL);
       }
    switch(tipo) {
           case UNSIGNED_CHAR:
             tamanho = sizeof(unsigned char);
             tamanho_elem = sizeof(unsigned char);
             break;
           case CHAR:
             tamanho = sizeof(char);
             tamanho_elem = sizeof(char);
             break;
           case INT:
             tamanho_elem = sizeof(short int);
             tamanho = tamanho_elem;
             break;
           case UNSIGNED_INT:
             tamanho_elem = sizeof(float);
             tamanho = sizeof(unsigned int);
             break;
           case LONG_INT:
             tamanho_elem = sizeof(float);
             tamanho = sizeof(long int);
             break;
           case UNSIGNED_LONG:
             tamanho_elem = sizeof(float);
             tamanho = sizeof(unsigned long);
             break;
```

```
          case FLOAT:
            tamanho = sizeof(float);
            tamanho_elem = sizeof(float);
            break;
          case DOUBLE:
            tamanho = sizeof(double);
            tamanho_elem = sizeof(double);
            break;
          default:
            printf("\nTipo não compativel com o sistema \n");
            break;
       }
       AAA = aloca_mat(linha,coluna,tamanho_elem);
       buf = (double *)calloc(coluna,tamanho);
       if(!buf) {
       printf("\nErro na alocação de buffer em ler_mat() \n");
       exit(1);
      }
       for(i = 0; i < linha; i++) {
         ler_registro((char *)buf,arquivo,coluna,tamanho);
         switch(tipo) {
            case CHAR:
                 ESCALA_VEC(buf,AAA->ptr_mat[i],1,coluna,char,int)
                 break;
            case UNSIGNED_CHAR:
                  ESCALA_VEC(buf,AAA->ptr_mat[i],1,coluna,unsigned \\
                        char,unsigned char)
                 break;
            case INT:
                 ESCALA_VEC(buf,AAA->ptr_mat[i],1,coluna,int,int)
                 break;
            case UNSIGNED_INT:
                 ESCALA_VEC(buf,AAA->ptr_mat[i],1,coluna,int,float)
                 break;
            case LONG_INT:
                 ESCALA_VEC(buf,AAA->ptr_mat[i],1,coluna,long int,float)
                 break;
            case UNSIGNED_LONG:
            ESCALA_VEC(buf,AAA->ptr_mat[i],1,coluna,unsigned long,float)
            break;
            case FLOAT:
                 ESCALA_VEC(buf,AAA->ptr_mat[i],1,coluna,float,float)
                 break;
            case DOUBLE:
                 ESCALA_VEC(buf,AAA->ptr_mat[i],1,coluna,double,double)
                 break;
            }
       }
     free(arquivo->nome);
     fclose(arquivo->ptr_arq);
     free((char *)arquivo);
     return(AAA);
   }
```

```
/*****************************************************************
grava_mat  grava uma matriz do mesmo tipo dado na estrutura MATRIX (int,
         float ou double) para um arquivo com estrutura ARQUIVO. Provoca
         exit(1) se ocorrer um erro de leitura ou se a estrutura for
         invalida. Retorna um ponteiro para a estrutura ARQUIVO.

ARQUIVO *grava_mat(MATRIX *AAA, char *nome_arq)
*****************************************************************/
ARQUIVO *grava_mat(MATRIX *AAA, char *nome_arq)
   {
    ARQUIVO *arquivo;
    int i;

     arquivo = abre_escreve(nome_arq);
     if(!arquivo) {
      return(NULL);
      }

      for(i = 0; i < AAA->linha; i++)
          grava_registro(AAA->ptr_mat[i],arquivo,AAA->coluna,AAA-
>tamanho);

      return(arquivo);

   }
/*****************************************************************
mostra_mat()  mostra uma matriz no display do monitor usando a função
            printf() para mostrar os elementos nos formatos %g ou %d.

void mostra_mat(MATRIX *AAA)
 *****************************************************************/
void mostra_mat(MATRIX *AAA)
       {
       int i, j, col;
       unsigned char **a;
      if(AAA->tamanho == sizeof(short) || AAA->tamanho == sizeof(unsigned
char))
           col = 8;
       else
           col = 8;

       for(i = 0; i < AAA->linha; i++) {
          if (i > 2) continue;
          for(j = 0; j < AAA->coluna; j++) {
            if(j%col == 0) {
              if(j == 0)
                printf("\nLinha%3d:\n",i);
              else
                printf("\n");
             }
            switch(AAA->tamanho) {
                 case sizeof(unsigned char):
                  a = (unsigned char **)AAA->ptr_mat;
                  printf("AA[%d][%d] = %7u   ",i,j,a[i][j]);
            printf("%6u",((unsigned char **)AAA->ptr_mat)[i][j]);

                  break;
```

```
                case sizeof(short int):
                   printf("%6d",((short **)AAA->ptr_mat)[i][j]);
                   break;
                case sizeof(float):
                   printf("%9.5g",((float **)AAA->ptr_mat)[i][j]);
                   break;
                case sizeof(double):
                   printf("%9.5lg",((double **)AAA->ptr_mat)[i][j]);
                   break;
             }
          }
       }
      }
/*****************************************************************/
```

```
/*****************************************************************
R_W_REC.C   ler ou escreve um registro em um arquivo aberto no modo
         binario com estrutura ARQUIVO
 *****************************************************************
ler_registro    ler um geristro de um arquivo aberto no modo binario
grava_registro  grava um geristro em um arquivo aberto no modo binario
conv_reg_float  ler um registro char, int dou double e converte para
           float.
 *****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "maniparq.h"
/*****************************************************************
ler_registro      ler um geristro de um arquivo com estrutura ARQUIVO.
           Provoca exit(1) se ocorrer um erro de leitura ou se a
           estrutura for invalida. Não retorna nenhum valor.

void ler_registro(char *ptr_buf, ARQUIVO *arquivo,int coluna,int tamanho)
*****************************************************************/
void ler_registro(char *ptr_buf, ARQUIVO *arquivo,int coluna,int tamanho)
   {
    int estado;

    if(!arquivo) {
       printf("\nErro na estrutura passada para ler_rec() \n");
       exit(1);
     }
    estado = fread(ptr_buf,tamanho,coluna,arquivo->ptr_arq);
    if(estado != coluna) {
      printf("\nErro na leitura de registro em %s. ",arquivo->nome);
      printf("\nO tipo gravado eh diferente do tipo pretendido \n");
      exit(1);
     }
  }
/*****************************************************************
grava_registro  grava um geristro de um arquivo com estrutura ARQUIVO.
           Provoca exit(1) se ocorrer um erro de leitura ou se a
           estrutura for invalida. Não retorna nenhum valor.

void grava_registro(char *ptr_reg, ARQUIVO *arquivo,int coluna,int
tamanho)
*****************************************************************/
void grava_registro(char *ptr_reg, ARQUIVO *arquivo,int coluna,int
tamanho)
   {
    int estado;
    if(!arquivo) {
       printf("\nErro na estrutura passada para grava_registro() \n");
       exit(1);
     }
    estado = fwrite(ptr_reg,tamanho,coluna,arquivo->ptr_arq);
    if(estado != coluna) {
      printf("\nErro na gravação de registro em %s \n",arquivo->nome);
      exit(1);

     }
```

```
  }
/*********************************************************************
**********************************************************************
conv_reg_float()  ler um registro do arquivo entrado pelo usuário e
            converte os dedos lidos para um arranjo float. Retorna
            um ponteiro para o inicio do arranjo float alocado,
            representando o registro lido do arquivo.
float *conv_reg_float(ARQUIVO *arquivo,int coluna,unsigned char tipo)
**********************************************************************/
float *conv_reg_float(ARQUIVO *arquivo,int coluna,unsigned char tipo)
   {
    int i, tamanho;
    float *out;
    float *out_ptr;
    static double *buf;
    void ler_registro();
    long int tamanho_byte;
    static long int tamanho_previsto = 0;

    switch(tipo) {
            case CHAR:
            case INT:
                 tamanho = sizeof(short int);
                 break;
            case FLOAT:
                 tamanho = sizeof(float);
                 break;
            case DOUBLE:
                 tamanho = sizeof(double);
                 break;
            default:
                printf("\nTipo não compativel com o sistema \n");
                break;
       }

    tamanho_byte = (long)coluna*tamanho;
    if(tamanho_byte != tamanho_previsto) {
      if(tamanho_previsto != 0)
           free(buf);
       buf = (double *)calloc(coluna,tamanho);
       if(!buf) {
            printf("\nErro na alocação do buffer de entrada \n");
            exit(1);
         }
       tamanho_previsto = tamanho_byte;
      }
     if(tipo != FLOAT) {
         out = (float *)calloc(coluna,sizeof(float));
         if(!out) {
            printf("\nErro de alocação em conv_reg_float() \n");
            exit(1);
          }
       }
      ler_registro((char *)buf,arquivo,coluna,tamanho);

      out_ptr = out;
      switch(tipo) {
```

```
            case CHAR: {
               char *char_ptr;
               char_ptr = (char *)buf;
               for(i = 0; i < coluna; i++)
                  *out_ptr++ = (float)(*char_ptr++);
              }
              break;
            case INT: {
               int *int_ptr;
               int_ptr = (int *)buf;
               for(i = 0; i < coluna; i++)
                  *out_ptr++ = (float)(*int_ptr++);
              }
              break;
            case FLOAT:
               out = (float *)buf;
               tamanho_previsto = 0;
               break;
            case DOUBLE: {
               double *dbl_ptr;
               dbl_ptr = (double *)buf;
               for(i = 0; i < coluna; i++)
                   *out_ptr++ = (float)(*dbl_ptr++);
              }
             break;
          }
         return(out);

}/*********************************************************************/
```

```
/*****************************************************************
RWDATA.C   ler ou escreve um registro em um arquivo aberto no modo
          binario com estrutura HEAD_DATA
 *****************************************************************
ler_data    ler um geristro de um arquivo aberto no modo binario
grava_data  grava um geristro em um arquivo aberto no modo binario
 *****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "maniparq.h"
/*****************************************************************
ler_data   ler um geristro de um arquivo com estrutura HEAD_DATA.
        Provoca exit(1) se ocorrer um erro de leitura ou se a
        estrutura for invalida. Não retorna nenhum valor.

void ler_data(char *ptr_buf, HEAD_DATA *arquivo,int tam,int comp)
*****************************************************************/
void ler_data(char *ptr_buf, HEAD_DATA *arquivo,int tam,int comp)
   {
    int estado;
    if(!arquivo) {
       printf("\nErro na estrutura passada para ler_data() \n");
       exit(1);
     }
    estado = fread(ptr_buf,tam,comp,arquivo->fptr);
    if(estado != comp) {
      printf("\nErro na leitura de registro em %s, ",arquivo->nome_file);
      printf("O tipo gravado eh diferente do tipo pretendido \n");
      exit(1);
     }
  }
/*****************************************************************
grava_data  grava um registro de um arquivo com estrutura HEAD_DATA.
            Provoca exit(1) se ocorrer um erro de leitura ou se a
            estrutura for invalida. Não retorna nenhum valor.

void grava_data(char *ptr_reg, HEAD_DATA *arquivo,int t_byte,int comp)
*****************************************************************/
void grava_data(char *ptr_reg, HEAD_DATA *arquivo,int t_byte,int comp)
   {
    int estado;
    if(!arquivo) {
       printf("\nErro na estrutura passada para grava_data() \n");
       exit(1);
     }
    estado = fwrite(ptr_reg,t_byte,comp,arquivo->fptr);
    if(estado != comp) {
      printf("\nErro na gravação de registro em %s \n",arquivo-
>nome_file);
      exit(1);
     }
  }
```

```
/*********************************************************************
RWHEADER.C   ler ou escreve um cabecalho em um arquivo aberto no modo
          binario com estrutura HEAD_DATA.
*********************************************************************
ler_header      abre e ler o cabecalho de um arquivo aberto em binario
grava_header    abre e grava o cabecalho de um arquivo aberto em binario
*********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "maniparq.h"
/*********************************************************************
*********************************************************************
grava_header  Abre um arquivo no modo "wb" e grava os dados de cabecalho
            em uma estrutura do tipo HEADER. Retorna um ponteiro para
            uma estrutura do tipo HEAD_DATA alocada pela função que
chama.
            Provoca exit(1) se ocorrer um erro ou se a estrutura for
            invalida. Retorna ponteiro NULL se o nome do arquivo for
            invalido.

HEAD_DATA *grava_header(char *file_name,int lin,int col,int NB)
*********************************************************************/
HEAD_DATA *grava_header(char *file_name,int lin,int col,int NB)
   {
    int estado;
    HEAD_DATA *arquivo;
    /*-------------------------------------------------------------*/
    arquivo = (HEAD_DATA *)malloc(sizeof(HEAD_DATA));
    if(!arquivo) {
       printf("\nErro em grava_head(), alocação da estrutura, arquivo:
%s\n",
              file_name);
       exit(1);
     }
    /*-------------------------------------------------------------*/
    /* Atribui os valores iniciais do cabecalho */
       arquivo->lin_image=(unsigned int)lin;
       arquivo->col_image=(unsigned int)col;
       arquivo->tam_bloco=(unsigned char)NB;
    /*-------------------------------------------------------------*/
    arquivo->fptr=fopen(file_name,"wb");
    if(!arquivo->fptr) {
       printf("\nErro em grava_head(), abrindo o arquivo:
%s\n",file_name);
       return(NULL);
     }
    /*-------------------------------------------------------------*/
    arquivo->nome_file=(char *)malloc(strlen(file_name)+1);
    if(!arquivo->nome_file) {
      printf("\nNão foi possivel alocar espaço para %s em rava_head()\n",
             file_name);
       exit(1);
     }

    strcpy(arquivo->nome_file,file_name);
    /*-------------------------------------------------------------*/
```

```
    estado = fwrite((char *)arquivo,sizeof(HEADER),1,arquivo->fptr);
    if(estado != 1) {
      printf("\nErro na gravação do HEADER do arquivo %s \n",file_name);
      exit(1);
     }
    /*-------------------------------------------------------------*/
    /* Retorna o ponteiro para a estrutura HEAD_DATA */
    return(arquivo);
    /*-------------------------------------------------------------*/
  }
/*****************************************************************
ler_header  Abre um arquivo no modo "w+b" e ler o cabecalho desse arquivo
          com estrutura HEAD_DATA. Provoca exit(1) se ocorrer um erro de
          alocação ou estrutura invalida e retorna um ponteiro NULL se
          ocorrer um erro de leitura. A função retorna um ponteiro para
          uma estrutura HEAD_DATA alocada pela função chamadora.

HEAD_DATA *ler_header(char *file_name)
*****************************************************************/
HEAD_DATA *ler_header(char *file_name)
   {
    int estado;
    HEAD_DATA *arquivo;
    /*-------------------------------------------------------------*/
    arquivo = (HEAD_DATA *)malloc(sizeof(HEAD_DATA));
    if(!arquivo) {
      printf("\nErro em ler_header(), alocação da estrutura de %s \n",
            file_name);
      exit(1);
      }
    /*-------------------------------------------------------------*/
    arquivo->fptr=fopen(file_name,"rb");
    if(!arquivo->fptr) {
      printf("\nErro em ler_header(), arbertura do arquivo %s
\n",file_name);
      return(NULL);
      }
    /*-------------------------------------------------------------*/
    /* Aloca e copia o nome do arquivo para a estrutura */
    arquivo->nome_file=(char *)malloc(strlen(file_name)+1);
    if(!arquivo->nome_file) {
      printf("\nNão foi possivel alocar espaço para %s em
ler_header()\n",
            file_name);
      exit(1);
      }
     strcpy(arquivo->nome_file,file_name);
    /*-------------------------------------------------------------*/
    estado = fread((char *)arquivo,sizeof(HEADER),1,arquivo->fptr);
    if(estado != 1) {
      printf("\nErro na leitura do HEADER em %s, ",file_name);
      printf("O tipo gravado eh diferente do tipo pretendido \n");
      exit(1);

     }
    return(arquivo);
  }
```

```
/*********************************************************************/
/* SNRCH.C Calcula a relação sinal/ruido entre duas imagens */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "maniparq.h"
/*********************************************************************/
main()
{
    int i,j,k,lin,col,linha,coluna;
    int Xmin,Ymin,Xmax,Ymax,erromin,erromax;
    int xmil,xmic,ymil,ymic,xmal,xmac,ymal,ymac;
    int erromil,erromic,erromal,erromac;
    int Xerromin,Xerromax,Yerromin,Yerromax;
    char ch,var,*nome_arq;
    unsigned char *X,*Y;
    double Ntam,erro,x2,y2,erro2,soma_x,soma_y,soma_erro;
    double soma_x2,soma_y2,soma_erro2,sigma_x,sigma_y,sigma_erro;
    float media_x,media_y,media_erro,SNRXY,SNRXYpico;
    ARQUIVO *arq_in1,*arq_in2;
/*-------------------------------------------------------------------*/
printf("\nCalcula a relação Sinal/Ruindo entre a imagem Original e a ");
printf("\n imagem processada recuperada.");
/*-------------------------------------------------------------------*/
  do {
      nome_arq=ler_str("nome do arq. da IMAGEM ORIGINAL ");
      linha = ler_int("número de linhas da imagem",1,512);
      coluna = ler_int("número de colunas da imagem",1,1024);
      arq_in1 = abre_ler(nome_arq);
   } while(!arq_in1);
  do {
      nome_arq=ler_str("nome do arquivo da IMAGEM RECUPERADA ");
      arq_in2 = abre_ler(nome_arq);
   } while(!arq_in2);
/*-------------------------------------------------------------------*/
       X = calloc(coluna,sizeof(unsigned char));
       Y = calloc(coluna,sizeof(unsigned char));
       Ntam = (double)linha*coluna;
   Xmin=Ymin=erromin=32766;
   Xmax=Ymax=erromax=-32767;
   Xerromin=Xerromax=Yerromin=Yerromax=0;
   soma_x=soma_x2=soma_erro=soma_erro2=soma_y=soma_y2=0.0;
   media_x=media_y=media_erro=0.0;
   for(lin = 0; lin < linha; lin++) {
       ler_registro((char *)X,arq_in1,coluna,sizeof(char));
       ler_registro((char *)Y,arq_in2,coluna,sizeof(char));
       for(col = 0; col < coluna; col++) {
         erro = (double)(X[col]-Y[col]);
         x2 = (double)(X[col]*X[col]);
         y2 = (double)(Y[col]*Y[col]);

         erro2 = erro*erro;
         soma_x+=(double)X[col]; soma_x2+=x2;
         soma_y+=(double)Y[col]; soma_y2+=y2;
         soma_erro+=fabs(erro);  soma_erro2+=erro2;
         if((int)X[col]<Xmin) { Xmin=(int)X[col]; xmil=lin; xmic=col; }
         if((int)X[col]>Xmax) { Xmax=(int)X[col]; xmal=lin; xmac=col; }
```

```
         if((int)Y[col]<Ymin) { Ymin=(int)Y[col]; ymil=lin; ymic=col; }
         if((int)Y[col]>Ymax) { Ymax=(int)Y[col]; ymal=lin; ymac=col; }
         if((int)erro<erromin) {
            erromin=(int)erro; Xerromin=(int)X[col];
Yerromin=(int)Y[col];
            erromil=lin; erromic=col; }
         if((int)erro>erromax) {
            erromax=(int)erro; Xerromax=(int)X[col];
Yerromax=(int)Y[col];
            erromal=lin; erromac=col; }
       }
     }
     media_x = (float)(soma_x/Ntam);
     media_y = (float)(soma_y/Ntam);
     media_erro = (float)(soma_erro/Ntam);
     sigma_x = fabs(Ntam*soma_x2-(soma_x*soma_x))/(Ntam*(Ntam-1));
     sigma_y = fabs(Ntam*soma_y2-(soma_y*soma_y))/(Ntam*(Ntam-1));
     sigma_erro = fabs(Ntam*soma_erro2-
(soma_erro*soma_erro))/(Ntam*(Ntam-1));
     if(sigma_erro < 0.0)  { SNRXY = -1000.0; SNRXYpico = -1000.0; }
     if(sigma_erro == 0.0)  { SNRXY = 1000.0; SNRXYpico = 1000.0; }
     if(sigma_erro > 0.0)  {
       SNRXY = (float)(10*log10(soma_x2/soma_erro2));
       SNRXYpico = (float)(10*log10((Ntam*255.0*255.0)/soma_erro2));
       }
     printf("\nImagem Original :");
     printf("\nValor Maximo = %d lin=%d col=%d ",Xmax,xmal,xmac);
     printf("\nValor Minimo = %d lin=%d col=%d ",Xmin,xmil,xmic);
     printf("\nValor Medio = %f ",media_x);
     printf("\nDesvio Padrao = %f ",(float)sigma_x);
     printf("\n\nImagem Processada :");
     printf("\nValor Maximo = %d lin=%d col=%d ",Ymax,ymal,ymac);
     printf("\nValor Minimo = %d lin=%d col=%d ",Ymin,ymil,ymic);
     printf("\nValor Medio = %f ",media_y);
     printf("\nDesvio Padrao = %f ",(float)sigma_y);
     printf("\n\nResultados do Processo :");
     printf("\nErro Maximo= %d Sinal= %d ",erromax,Xerromax);
     printf(" Sinal Rec.= %d lin=%d col=%d ",Yerromax,erromal,erromac);
     printf("\nErro Minimo= %d Sinal= %d ",erromin,Xerromin);
     printf(" Sinal Rec.= %d lin=%d col=%d ",Yerromin,erromil,erromic);
     printf("\nErro Medio = %f ",media_erro);
     printf("\nErro Desvio Padrao = %f ",(float)sigma_erro);
     printf("\n\nRELAÇÃO SINAL/RUIDO = %f (dB)",SNRXY);
     printf("\nRELAÇÃO SINAL/RUIDO DE PICO = %f (dB)",SNRXYpico);
/*-----------------------------------------------------------------*/
   free(X); free(Y);
   fcloseall();
}
/*****************************************************************/
```

LISTA DE PUBLICAÇÕES

Publicações realizadas durante o curso de mestrado

A. M. Do Nascimento, Yuzo Iano. "Novas Investigações Comparativas de Arquiteturas para Redução da Taxa de Bits de Sinais de TV PAL-M em 4fsc Usando Modelo de Campo QUINCUNX ". Tese de Mestrado, UNICAMP, 1994.

A. M. Do Nascimento, Yuzo Iano. "Algoritmos para Composição e Decomposição de Sinais Primários de TV com Aplicações em Simulação de PDI". RPIC'93 - ARGENTINA, Universidad Nacional de Tucumán, dez/93.

Publicações realizadas durante o curso de doutorado

A. Cardoso, A. M. Do Nascimento, Yuzo Iano y Guillermo Kemper, "Desarrollo de Programas em Lenguaje C en el Ambiente de Programación Visual Khoros", *revista "CIT - Información Tecnológica"*, ISSN 0716-8756, Vol. 11 N^{o} 2 La Serena, Chile, 2000.

A. M. Do Nascimento, Yuzo Iano y Guillermo Kemper. "Conversión de frecuencias de Imagenes Digitalizadas para Aplicaciones en Simulación de SDVPS", *revista "CIT - Información Tecnológica"*, Vol. 8 N^{o} 4 La Serena, Chile, 1997.

A. M. Do Nascimento, Yuzo Iano. "HDTV: Which Standard Formats", *Intercon'96 Third International Electronic Engineering Conference* , Trujillo – Peru, Aug. 12-17^{tH}, 1996

A. M. Do Nascimento, Yuzo Iano. "Specific Software Development for TV and ATV Signal Formating", *Intercon'96 Third International Electronic Engineering Conference*, Trujillo – Peru, Aug. 12-17^{tH}, 1996

A. M. Do Nascimento, Yuzo Iano, Antonio Maria Fancony. "Technical, Political and Economical Considerations on HDTV Implantation in BRAZIL", *Intercon'96 Third International Electronic Engineering Conference*, Trujillo – Peru, Aug. 12-17^{tH}. 1996

A. M. Do Nascimento, Yuzo Iano, "Processamento Digital de Sinais", *Convênio CPqD-TELEBRAS-UNICAMP,* Publicação FEEC 12/96, Campinas S.P., Brasil, Abril 1996.

A. M. Do Nascimento, Yuzo Iano. "Conversão de Freqüência de Imagens Digitalizadas para Aplicação em Simulação de SDVPI ". *XI Congresso Chileno de Ingenieria Electrica, ELECTRO'95*, Universidad de Magallane, CHILE, Novembro 1995.

Publicações submetidas

A. M. Do Nascimento, Yuzo Iano. "A Scheme for Adaptive Discrete Cosine Transform Coding of Images", IEEE Transactions on Multimedia, 2001.

A. M. Do Nascimento, Yuzo Iano. "Transformed Image Blocks: search for optimum classification" IEEE Transactions on Image Processing, 2001.