UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

DEPARTAMENTO DE COMUNICAÇÕES

IMPLEMENTAÇÃO DE UM COMPRESSOR DE IMAGENS NA LINGUAGEM PYTHON USANDO A TRANSFORMADA WAVELET

Dissertação apresentada à Faculdade de Engenharia Elétrica e Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica.

AUTOR: AURIO GONÇALEZ TENÓRIO
ORIENTADOR: LUIZ CÉSAR MARTINI

BANCA EXAMINADORA:

Prof. Dr. Luiz César Martini, FEEC/UNICAMP
Prof. Dr. Roberto de Alencar Lotufo, FEEC/UNICAMP
Prof. Dr. João Batista Tadanobu Yabuuti, FEEC/UNICAMP
Prof. Dr. José Eduardo Cogo Castanho, UNESP/BAURU

Campinas, 13 de outubro de 2003.

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

G586i

Gonçalez, Aurio Tenório
Implementação de um compressor de imagens na linguagem python usando a transformada wavelet. / Aurio Gonçalez Tenório.--Campinas, SP: [s.n.], 2003.

Orientador: Luiz César Martini
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Compressão de imagens. 2. Programação (Computadores). 3. Algoritmos. 4. Wavelet. I. Martini, Luiz César. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

RESUMO

O propósito deste trabalho é estudar e implementar um esquema compressor com perdas de imagens bidimensionais com 256 níveis de cinza utilizando a linguagem de programação Python. O software foi desenvolvido em Python, primeiro, porque é uma linguagem de programação de código aberto, e segundo, porque apresenta ferramentas adequadas ao processamento de sinais bidimensionais, dentro do pacote Numerical Python. Na simulação do modelo compressor, estaremos verificando se as técnicas estudadas e implementadas apresentam resultados satisfatórios na compressão do sinal. O objetivo é conseguir alta compressão com baixa distorção.

O esquema de compressão escolhido foi um modelo de *codificação por transformada* onde três etapas são necessárias para comprimir uma imagem: transformada linear, quantização, e codificação por entropia. Neste contexto, a transformada wavelet é aplicada para descorrelacionar os componentes de alta freqüência dos componentes de baixa freqüência. A seguir a imagem é quantizada e codificada por entropia. Na quantização dos coeficientes, modelos de quantização escalar são simulados, incluindo quantização escalar uniforme e uniforme com zona morta. Nesta etapa, é simulado ainda, o desempenho do algoritmo *EZW – Embedded Zerotree Wavelet*. Finalmente, a codificação por entropia é usada com o objetivo de comprimir a imagem de modo eficiente. Os codificadores implementados incluem o código de corrida, o código de Huffman e o código aritmético. Neste trabalho, também será simulado o desempenho da codificação recursiva do sinal, técnica mais eficaz de codificação que o sinal codificado diretamente, como será demonstrado nas simulações. Será simulado também, o desempenho da transformada do sinal de média nula em contraste com a transformada do sinal original.

ABSTRACT

The purpose of this work is to study and implement a lossy compression scheme to compress two-dimensional 256 gray-scale images under Python programming language. The software was developed using Python, first, because its an open source programming language, and second, because of the tools available to process two-dimensional signals inside the Numerical package. During the compressor model simulation, we will verify if the techniques studied and implemented present satisfactory performance when compressing the signal.

The compression scheme chosen was an image-transformed model, where three steps are necessary to compress the signal, which are linear transform, quantization and entropy encoding operations. In this context, wavelet transform is applied to decorrelate the high frequencies components from the low frequencies. Next, the image is quantized and entropy encoded. During the coefficients quantization step, scalar quantization models are simulated, including uniform scalar quantization and uniform dead zone scalar quantization. In addition, we simulate the algorithm *EZW – Embedded Zerotree Wavelet*. Finally, entropy encoding is used to compress the image in an efficient way. The encoders implemented include run-length encoding, Huffman encoding and arithmetic encoding techniques. In this work, we also simulate the signal recursive encoding, technique more efficient than direct codification, as shown in the simulations. We simulated also, the performance of the zero mean signal transform, in contrast with the original signal transform.

Mas a vida é curta e a informação infinita...
Abreviações são um mal necessário e a tarefa do abreviador
é fazer o melhor trabalho que, embora intrinsecamente ruim,
seja ainda melhor que nada.
Aldous Huxley

AGRADECIMENTOS

Gostaria de agradecer as seguintes pessoas e instituições:

- *Ao Prof. Dr. Luiz César Martini, pela oportunidade de desenvolver este projeto nesta conceituada instituição e pela ótima convivência durante este período.*
- *Ao CNPQ – Conselho Nacional de Desenvolvimento Científico – pelo suporte financeiro fornecido durante o desenvolvimento do projeto.*
- *A minha família, pelo apoio sempre presente.*
- *A minha noiva e aos seus pais, pela paciência e apoio nos momentos de dificuldade.*
- *Aos colegas do Departamento de Comunicações, da Faculdade de Engenharia Elétrica e Computação da Unicamp, pelas novas amizades e momentos de descontração*.

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO 1 INTRODUÇÃO

1.1 INTRODUÇÃO À COMPRESSÃO DE IMAGENS

Uma enorme quantidade de dados é produzida quando uma função intensidade de luz bidimensional é amostrada e quantizada para criar uma imagem digital. Uma imagem é uma função positiva num plano. O valor desta função em cada ponto especifica a luminância ou brilho da figura naquele ponto. Cada valor da função é especificado apenas em posições discretas no plano da imagem, denominadas *pixels*. O valor da luminância de cada pixel é representado por uma precisão pré-definida *M*. Usar oito bits de precisão para a luminância é uma representação comum. A precisão em oito bits é motivada pela estrutura de memória dos computadores (1 byte = 8 bits). É comum que as amostras (pixels) residam num grid retangular $N \times N$. O valor do brilho em cada pixel é um número entre 0 e $2^M - 1$. A representação binária mais simples de tal imagem é uma lista de valores de brilho para cada pixel, contendo N^2M bits. As imagens utilizadas no desenvolvimento desta tese são imagens com 256 pixels em cada lado, com cada pixel assumindo um valor entre 0 e 255. Então, sua representação canônica necessita de $256^2 \times 8 = 524.288$ bits [1].

A quantidade de dados gerada pode ser tão grande que inviabiliza o armazenamento e o processamento, demandando grande largura de banda para transmissão que a tecnologia atual não é capaz de suportar tecnicamente e economicamente. A compressão de imagens trata do problema de reduzir a quantidade de dados necessária para representar uma imagem digital. A base do processo de redução é a eliminação das redundâncias [2]. As informações redundantes são classificadas em dois tipos [3]:

- Redundância espacial: correlação entre pixels adjacentes.
- Redundância espectral: em diferentes bandas espectrais as magnitudes dos pixels nas mesmas posições espaciais não são estatisticamente independentes.

A codificação de imagens consiste em mapear imagens para seqüências de dígitos binários. Um bom codificador é aquele que produz seqüências binárias cujos comprimentos são, na média, muito menores que a representação canônica original da imagem. Em muitas aplicações, a exata reprodução dos bits da imagem não é necessária. Neste caso pode-se inserir uma ligeira perturbação na imagem para obter uma representação mais compacta. Este procedimento de codificação, onde as perturbações reduzem as exigências de armazenamento, é conhecido por codificação ou *compressão com perdas*, em contraste com a *compressão sem perdas*.

Na compressão sem perdas, a taxa de compressão atingida é baixa, porém, a imagem pode ser reconstruída perfeitamente a partir da versão comprimida (exceto pelo erro gerado pela precisão numérica computacional). Na compressão com perdas, a perfeita reconstrução da imagem é comprometida para alcançar melhores taxas de compressão. O objetivo da compressão com perdas é a máxima compressão para uma determinada distorção. O modelo seguido nesta tese é a compressão com perdas.

Um modelo para um codificador de imagens [4] consiste de três operações mostradas na figura 1.1.

Fig. 1.1: *Codificador de imagens.*

Usualmente, algum tipo de transformada linear é utilizado para eliminar as redundâncias espaciais. Uma transformada linear elimina as estatísticas de segunda ordem da imagem. Portanto, a entropia da imagem é reduzida. Exemplos de transformadas lineares incluem a Transformada Discreta de Cosseno (*DCT – Discrete Cosine Transform*) e a Transformada Wavelet baseada em subbandas. A seguir a imagem é quantizada e codificada por entropia, operações que tentam eliminar as redundâncias espectrais [4].

A quantização é uma operação irreversível e introduz distorções. Consiste em mapear uma determinada faixa de valores para um mesmo código, reduzindo o número de símbolos necessários para representar a imagem transformada. É possível quantizar cada pixel separadamente, um processo denominado *quantização escalar*. Quantizar um grupo de pixels conjuntamente é conhecido como *quantização vetorial*. A quantização vetorial pode, a princípio, capturar a máxima taxa de compressão que é teoricamente possível. Embora a quantização vetorial seja um paradigma teoricamente poderoso, ele pode ser otimizado apenas assintoticamente, conforme suas dimensões aumentam, limitando sua utilização prática. A *codificação por transformada* consiste em substituir a quantização vetorial por uma transformada linear em conjunção com uma quantização escalar. Este método captura muito dos ganhos da quantização vetorial, com um nível de complexidade muito menor [1].

A codificação por entropia produz representações menores das imagens pela utilização de palavras-código curtas para símbolos mais prováveis e palavras-código longas para símbolos menos prováveis. Exemplos da codificação por entropia incluem o código de Huffman, o código aritmético e o código de corrida.

Para reconstruir a imagem, primeiro a imagem é decodificada por entropia, depois é feita a dequantização e então a transformada inversa é aplicada. A figura 1.2 mostra o processo de reconstrução da imagem.

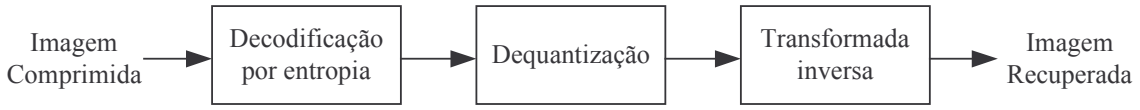


Fig. 1.2: *Decodificador de imagens.*

A técnica de codificação JPEG [34] utiliza um modelo semelhante para compressão de imagens. Na compressão JPEG, a imagem é dividida em blocos de tamanho 8x8, e em cada subimagem é aplicada a Transformada Discreta de Cosseno. Os coeficientes transformados são quantizados, sofrendo um processo de arredondamento. Matrizes especiais de quantização foram elaboradas para diferentes características das imagens sendo codificadas. Os coeficientes quantizados são então codificados por entropia. Seqüências consecutivas de símbolos são codificadas pelo código de corrida, usando uma varredura em zig-zag, e na seqüência o código de Huffman executa a compressão final. O processo se repete até que todos os blocos sejam codificados. A recuperação do sinal executa os passos na direção contrária, sendo a transformada inversa de cosseno a etapa final.

1.2 INTRODUÇÃO AO PYTHON

O Python é uma linguagem de alto nível, interpretada, orientada a objetos com uma semântica dinâmica. Suas estruturas de alto nível, combinadas com sua amarração dinâmica a tornar atrativa para desenvolvimento de largos aplicativos, assim como para uso como uma linguagem de script. Sua estrutura permite a aglutinação de módulos computacionais criando sistemas muito complexos em um nível de abstração muito mais alto do que as linguagens convencionais [5].

Uma das estruturas mais marcantes nesta linguagem é a ausência da necessidade de definição de tipos. Ou seja, uma variável não precisa ser declarada e atribuída um tipo que represente a função desta variável no programa. Essa variável pode ser uma string num instante e em outro um inteiro ou ainda uma função. A validação de tipos é feita em tempo de execução. Isso otimiza a troca de dados entre os componentes, facilitando a filosofia do aglutinamento de componentes, já que abstrai boa parte do processo de normalização de dados. Outra característica é o fato do Python ser uma linguagem interpretada, em contraste com as linguagens tradicionais, que são normalmente compiladas [6].

O objetivo dessa tese é estudar e implementar em Python, as operações usadas na compressão com perdas de imagens bidimensionais com 256 níveis de cinza. A aplicação desenvolvida implementa as três etapas discutidas acima, necessárias para comprimir uma imagem: a transformada linear, a quantização e a codificação por entropia. Também implementa o processo inverso, necessário para recuperar a imagem.

1.3 VISÃO GERAL

Esta tese segue a seguinte organização:

Capítulo 1 – Introdução à compressão de imagens e à linguagem de programação Python.

Capítulo 2 – Transformada Linear. O capítulo discutirá a Transformada de Fourier, Transformada curta de Fourier, Transformada Wavelet Contínua e Transformada Wavelet Discreta. Também apresentará os algoritmos para implementação computacional da Transforma Wavelet Discreta unidimensional e bidimensional.

Capítulo 3 – Quantização e codificação por entropia. Os métodos de quantização discutidos neste capítulo incluem quantização escalar uniforme, quantização escalar uniforme com zona morta. Também estaremos apresentando o algoritmo *EZW – Embedded Zerotree Wavelet*, como técnica alternativa de quantização. Na codificação por entropia, os tópicos discutidos incluem o código de corrida, o código de Huffman e o código aritmético, sendo também discutida a codificação recursiva da seqüência, como uma técnica para aumentar o nível de compressão. Também serão apresentados os fluxogramas dos algoritmos de codificação.

Capítulo 4 – Discussão e análise dos resultados obtidos. Este capítulo apresenta e discute o resultado das simulações dos algoritmos propostos sobre diferentes imagens teste.

Capítulo 5 – Resumo e conclusão.

CAPÍTULO 2 TRANSFORMADA LINEAR

2.1 INTRODUÇÃO

A *transformada wavelet* é uma solução recente para superar as limitações da transformada de Fourier. Enquanto a transformada de Fourier está limitada apenas a resolução em freqüência, a transformada wavelet oferece tanto boa resolução em freqüência quanto temporal. Wavelets são funções matemáticas que podem ser usadas para analisar componentes de diferentes freqüências de acordo com sua escala. Em oposição à transformada de Fourier, que usa como funções base senos e cossenos, a transformada wavelet usa funções base mais complexas, chamadas *wavelets*. Não existe uma função wavelet fixa, ao contrário, existe uma grande variedade. Essas funções se relacionam umas com as outras por dilatações (espalhamento ou contração) e translações. A versão espalhada perfaz a análise em freqüência enquanto a versão contraída perfaz a análise temporal ou espacial. Em imagens, o conteúdo de escala fina é geralmente encontrado em altas freqüências, enquanto que a informação grosseira é encontrada em baixas freqüências. A transformada wavelet usa sua capacidade de multiresolução para decompor a imagem em múltiplas bandas de freqüência. Sugerimos ao leitor consultar as referências [14], [15], [16], [17], [18], [19], [20], [21] e [22], que serviram de base para o capítulo, para conseguir informações complementares.

Esse capítulo começa fazendo uma breve revisão da transformada de Fourier e seus pares. A seguir, são apresentadas a transformada wavelet contínua e a transformada wavelet discreta.

2.2 TRANSFORMADA DE FOURIER

A transformada de Fourier contínua é uma integral de duas funções contínuas (um sinal e seu espectro). Essa transformada e sua transformada inversa são dadas em uma dimensão por:

$$F(s) = \int_{-\infty}^{\infty} f(x).e^{-j2\pi(xs)}dx \quad e \quad f(x) = \int_{-\infty}^{\infty} F(s).e^{j2\pi(xs)}ds\ , \tag{2.1}$$

A expansão em série de Fourier representa uma função periódica (ou uma função transiente que pode ser considerada um ciclo de uma função periódica) como uma seqüência (finita ou infinita) de coeficientes Fourier. Essa série e sua inversa são obtidas fazendo $s = n\Delta s$ uma variável discreta, tal que:

$$F_n = F(n\Delta s) = \int_0^L f(x).e^{-j2\pi(n\Delta s x)}dx \quad e \quad f(x) = \Delta s \sum_{n=0}^{\infty} F_n.e^{j2\pi(n\Delta s x)} \tag{2.2}$$

onde L é o período e $\Delta s = 1/L$.

A transformada discreta de Fourier, *DFT – Discrete Fourier Transform*, representa uma função amostrada de um espectro amostrado, e o número de amostras independentes é o mesmo em ambos os domínios. Ela é obtida fazendo $x = i\Delta x$ uma variável discreta. Se $g(x)$ é limitado em faixa e amostrado conforme exigido pelo teorema da amostragem, então $g_i = g(i\Delta x)$, e:

$$G_k = \frac{1}{\sqrt{N}}.\sum_{i=0}^{N-1} g_i.e^{-j2\pi k\frac{i}{N}} \qquad e \qquad g_i = \frac{1}{\sqrt{N}}.\sum_{k=0}^{N-1} G_k.e^{j2\pi i\frac{k}{N}}. \tag{2.3}$$

Em todas as três técnicas de transformação, senos e cossenos de diferentes freqüências formam um conjunto de funções base ortonormais. Também, cada coeficiente transformado é determinado pelo produto interno de uma função sendo transformada e uma das funções base. Em cada caso, a transformada inversa consiste em somar funções base cujas amplitudes são balanceadas pelos coeficientes transformados. Esta soma se torna uma integral para a transformada de Fourier contínua [16].

2.2.1 Transformada curta de Fourier

Para superar as limitações da transformada de Fourier, uma versão janela da transformada de Fourier, conhecida como transformada curta de Fourier, *STFT – Short Time Fourier Transform*, foi desenvolvida. Nesta transformada, pode-se dividir um sinal não estacionário em pequenos segmentos onde cada segmento é assumido como estacionário. Então, aplica-se a STFT nestes segmentos. A STFT tem vários problemas. Ao usar uma janela de comprimento infinito, voltamos para a transformada de Fourier, que oferece ótima resolução em freqüência, mas sem informação temporal. Por outro lado, objetivando obter uma amostra estacionária, deve-se ter uma janela pequena o suficiente para o sinal ser considerado estacionário. Quanto mais curta a janela, melhor a resolução no tempo, e melhor a suposição de sinal estacionário, porém, pior é a resolução em freqüência. Portanto, temos um dilema que a transformada wavelet tenta resolver.

2.3 TRANSFORMADA WAVELET

A idéia fundamental das wavelets é analisar um sinal em diferentes escalas ou resoluções, o que é chamado de multiresolução. Wavelets são uma classe de funções usadas para localizar um dado sinal tanto no espaço (tempo) quanto em freqüência. Uma família de wavelets pode ser construída a partir de uma wavelet mãe. Comparada com a análise da STFT, uma wavelet mãe é espalhada e comprimida para mudar o tamanho

da janela. Desta maneira, uma wavelet espalhada dá uma visão aproximada do sinal, enquanto que uma wavelet mais e mais contraída fornece detalhes finos do sinal. Portanto, as wavelets adaptam-se tanto aos componentes de baixa freqüência quanto aos componentes de alta freqüência conforme se varia o tamanho da janela. Qualquer pequena mudança na representação da wavelet produz uma mudança correspondente no sinal original [22].

Estaremos discutindo dois tipos de transformada wavelet: a transformada wavelet contínua, *CWT – Continuous Wavelet Transform*, e a transformada wavelet discreta, *DWT – Discrete Wavelet Transform*. A idéia principal é a mesma, entretanto, elas diferem na maneira como a transformada é realizada. Na CWT, uma janela de análise é deslocada ao longo do tempo para capturar a informação sobre o sinal. Na DWT, o sinal é analisado em passos discretos através de uma série de filtros, o que torna o processo computacionalmente realizável.

2.3.1 Transformada Wavelet Contínua e Série Wavelet

A *transformada wavelet contínua* de uma função *f(t)*, é definida como [21]:

$$W(s,\tau) = \left\langle f,\, \psi_{s,\tau} \right\rangle = \int_{-\infty}^{\infty} f(t)\ .\ \psi^{*}_{s,\tau}(t)dt\ . \tag{2.4}$$

Esta equação mostra como uma função *f(t)* é decomposta num conjunto de funções base $\{\psi_{s,\tau}(t)\}$, denominado *wavelets*. As variáveis *s* e τ são as novas dimensões, escala e translação. Os coeficientes da transformada wavelet são dados como o produto interno da função sendo transformada por cada uma das funções base.

Um conjunto de funções base wavelet, $\{\psi_{s,\tau}(t)\}$, é formado pela translação e dilatação da wavelet básica ou *wavelet mãe*, $\psi(t)$ [16]:

$$\psi_{s,\tau}(t) = \frac{1}{\sqrt{s}}.\psi\left(\frac{t-\tau}{s}\right). \tag{2.5}$$

A variável *s* reflete a escala (largura) de uma função base particular, enquanto τ especifica a posição de translação ao longo do eixo *t*. O fator $s^{-1/2}$ é para normalização de energia através de diferentes escalas.

As versões discretas da série e da transformada wavelet podem ser obtidas discretizando as dilatações e as translações. Assumindo dilatações binárias e translações unárias, as funções de base wavelet tornam-se:

$$\psi_{m,n}(t) = \psi_{s,\tau}(t) = 2^{m/2}.\psi(2^m t - n) \tag{2.6}$$

onde fizemos $(s,\tau) = (2^{-m},\ n.2^{-m})$, com m e n inteiros. A *série* e a *transformada wavelet* discretas serão então definidas como:

$$f(t) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} c_{m,n} . \psi_{m,n}(t) \tag{2.7}$$

onde, $c_{m,n} = \left\langle f(t),\ \psi_{m,n}(t) \right\rangle = 2^{m/2} \int_{-\infty}^{\infty} f(t).\psi^*(2^m t - n)dt$.

2.3.2 Transformada Wavelet Discreta

Através do conceito de análise em multiresolução, criado por [19], e difundido em [22], [23] e [24], podemos construir famílias de wavelets discretas e desenvolver algoritmos rápidos para o cálculo da *transformada wavelet discreta*, *DWT*. A análise em multiresolução parte da existência de duas funções básicas, um função wavelet mãe $\psi(t)$ e uma função escala mãe $\phi(t)$, ortogonais entre si e tais que, ao longo dos diversos níveis de resolução, as diversas funções de escala $\phi_{m,n}(t)$ e wavelet $\psi_{m,n}(t)$ estão relacionadas às respectivas funções mãe por meio das seguintes equações [20]:

$$\phi_{m,n}(t) = 2^{m/2}.\phi(2^m t - n)$$

$$\psi_{m,n}(t) = 2^{m/2}.\psi(2^m t - n)\ .$$

A relação entre os níveis de resolução pode ser descrita sob a forma de um aninhamento dos espaços de função, onde cada espaço de maior resolução contém os espaços de menor resolução, como expresso abaixo:

$$\cdots \subset V_{-2} \subset V_{-1} \subset V_0 \subset V_1 \subset V_2 \subset \cdots\ .$$

Cada espaço V_m é coberto pela família de funções base $\phi_{m,n}(t)$ correspondente. As funções wavelet $\psi_{m,n}(t)$ estão contidas nos espaços W_m, sendo cada espaço W_m o complemento ortogonal de V_m em relação ao espaço V_{m+1}. Desta forma, escrevemos a relação entre dois níveis adjacentes de resolução como:

$$V_{m+1} = V_m \oplus W_m$$

onde o símbolo $\oplus$ representa a operação de soma direta. Uma vez que V_m e W_m estão contidos em V_{m+1}, então ambos os conjuntos de funções base podem ser expressos como combinação linear das funções base de V_{m+1}, sendo cada uma delas ponderada pelos coeficientes $h_0[n]$ e $h_1[n]$. Portanto,

$$\phi_{m,n}(t) = \sum_k h_0[k-2n].\phi_{m+1,k}(t) \tag{2.8}$$

$$\psi_{m,n}(t) = \sum_k h_1[k-2n].\psi_{m+1,k}(t). \tag{2.9}$$

Estando a função f contida no espaço V_{m+1}, então f pode ser expressa como combinação linear das funções de base $\phi_{m+1,n}(t)$. De forma similar, projetando f em V_m e W_m e escrevendo estas projeções em função das bases de V_m e W_m, é possível relacionar as projeções de f e os diversos níveis de refinamento da seguinte forma:

$$a_{m+1,k} = \langle f, \phi_{m+1,k} \rangle . a_{m,n} = \langle f, \phi_{m,n} \rangle . c_{m+1,k} = \langle f, \psi_{m,n} \rangle .$$

Aplicando (2.8) e (2.9) nas equações acima, obtemos:

$$a_{m,n} = \langle f, \phi_{m,n} \rangle = \sum_k h_0[k-2n] . \langle f, \phi_{m+1,k} \rangle = \sum_k h_0[k-2n] . a_{m+1,k} \tag{2.10}$$

$$c_{m,n} = \langle f, \psi_{m,n} \rangle = \sum_k h_1[k-2n] . \langle f, \psi_{m+1,k} \rangle = \sum_k h_1[k-2n] . a_{m+1,k} \tag{2.11}$$

e dado que V_{m+1} é a soma direta dos espaços V_m e W_m, então, podemos expressar a base V_{m+1} como uma combinação linear das bases V_m e W_m, onde cada uma delas está ponderada pelos coeficientes $g_0[n]$ e $g_1[n]$:

$$\phi_{m+1,k}(t) = \sum_n g_0[k-2n] . \phi_{m,n}(t) + \sum_n g_1[k-2n] . \psi_{m,n}(t) . \tag{2.12}$$

Usando (2.12), podemos expressar $a_{m+1,k}$ como função das projeções $a_{m,n}$ e $c_{m,n}$:

$$a_{m+1,k} = \sum_n g_0[k-2n] . a_{m,n} + \sum_n g_1[k-2n] . c_{m,n} . \tag{2.13}$$

Os processos de síntese (composição) e análise (decomposição) descritos acima podem ser entendidos como a aplicação dos filtros passa-baixas e passa-altas seguidos de uma subamostragem (na análise) ou precedidos por uma superamostragem (na síntese) – como nos métodos de codificação de bandas (*Subband Coding*), onde os coeficientes $\{h_0[n], h_1[n]\}$ e $\{g_0[n], g_1[n]\}$ formam os bancos de filtros utilizados por estes métodos. Eles correspondem aos bancos de filtros de análise e de síntese, respectivamente. Todas as expressões apresentadas até aqui sugerem uma recursão, onde a projeção de f sobre V_m é decomposta em V_{m-1} e W_{m-1}; a projeção em V_{m-1} é decomposta sobre V_{m-2} e W_{m-2}, e assim por diante. Cada projeção equivale a uma versão menos refinada da projeção anterior. A figura 2.1 mostra exatamente esta relação recursiva entre os coeficientes da transformada contidos em cada subespaço. No detalhe, vemos também a estrutura do banco de filtros de análise.

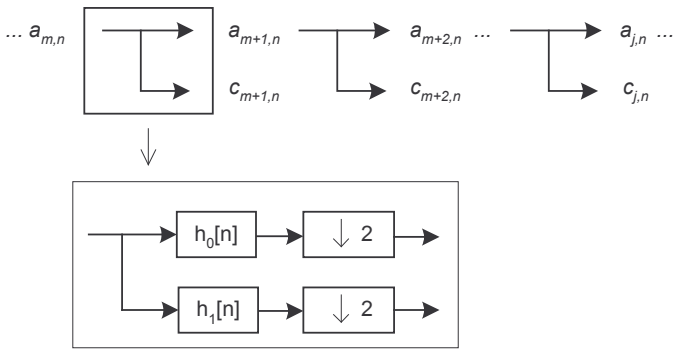


Fig. 2.1: *Processo de análise composto pela repetição do banco de filtros $\{h_0[n], h_1[n]\}$.*

Uma vez que as projeções nos espaços V_m e W_m são ortogonais entre si e complementares em relação ao espaço V_{m+1}, a reconstrução da função original do processo de síntese é perfeita – isto se não houver perdas entre a decomposição e a reconstrução. A figura 2.2 mostra o processo de recuperação do sinal decomposto e a estrutura do banco de filtros de síntese.

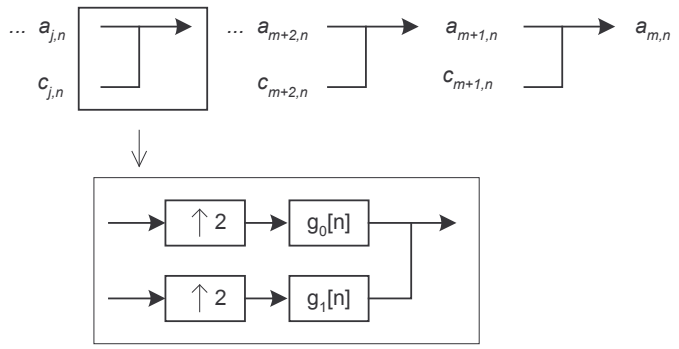


Fig. 2.2: *Processo de síntese composto pela repetição do banco de filtros $\{g_0[n], g_1[n]\}$.*

2.3.3 Transformada Wavelet Discreta Bidimensional

Felizmente, a transformada wavelet é separável. Então, o algoritmo descrito na seção anterior pode facilmente ser estendido para o caso bidimensional [4]. A DWT 2D de uma imagem, pode ser facilmente computada executando a DWT em cada dimensão. A figura 2.3 ilustra como a DWT 2D funciona. Primeiro, a imagem é filtrada ao longo do eixo x e dizimada por 2. Em seguida, essa nova imagem é filtrada ao longo do eixo y e dizimada por 2. Temos como resultado dessa operação a divisão da imagem em quatro subbandas, denominadas *LL*, *LH*, *HL*, *HH*, após um nível de decomposição.

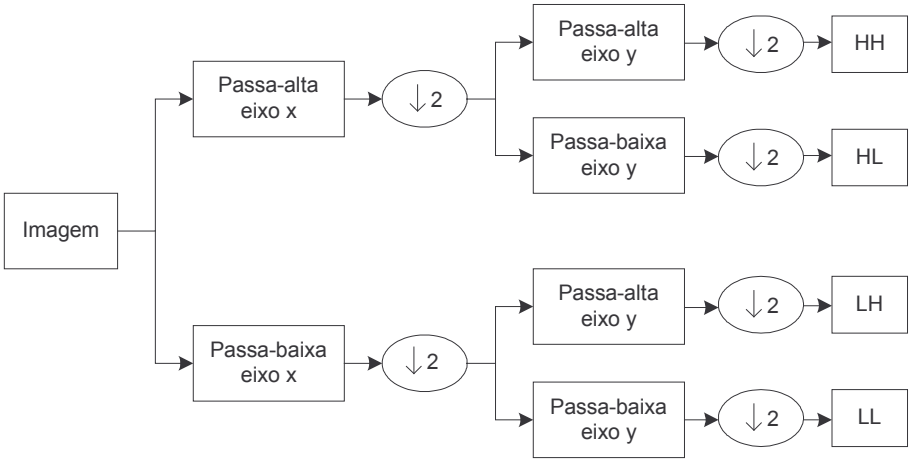


Fig. 2.3: *DWT 2D.*

Níveis mais altos de decomposição podem ser alcançados agindo sobre a subbanda *LL* sucessivamente, resultando numa divisão da imagem em múltiplas bandas. A figura 2.4 mostra a aplicação da DWT 2D a uma imagem até o segundo nível de decomposição.

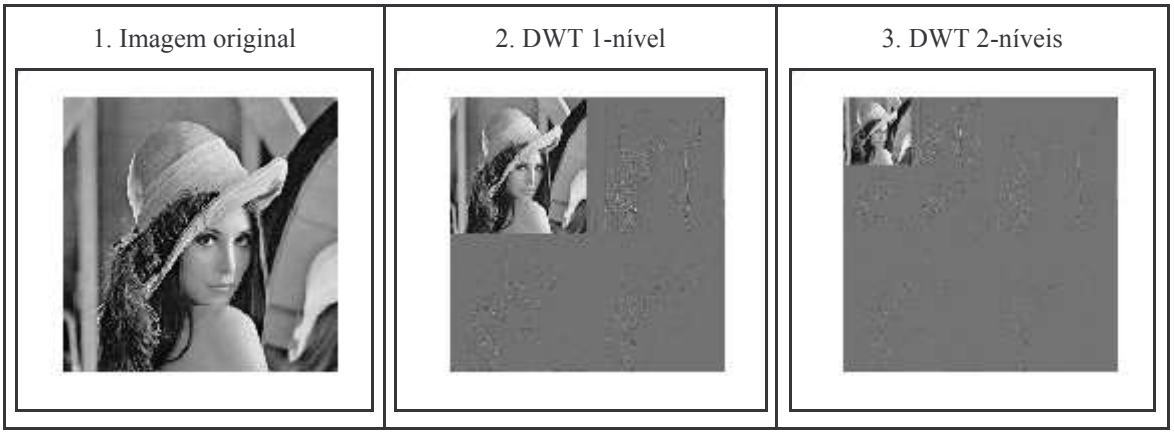


Fig. 2.4: *DWT 2D da imagem Lena usando coeficientes wavelet Daubechies W4.*

Para reconstruir a imagem, interpolar pelo fator 2 nas quatro subbandas do nível de decomposição mais alto. Em seguida, filtrar cada subbanda com o filtro síntese correspondente no eixo y, somar o resultado das filtragens nas bandas HH e LH, e também das bandas HL e LL. Interpolar o resultado de cada soma novamente pelo fator 2, filtrar no eixo x e somar os resultados para recuperar a subbanda LL correspondente ao nível anterior de decomposição. Repetir o processo até a imagem ser completamente reconstruída. Esse processo é ilustrado com a figura 2.5.

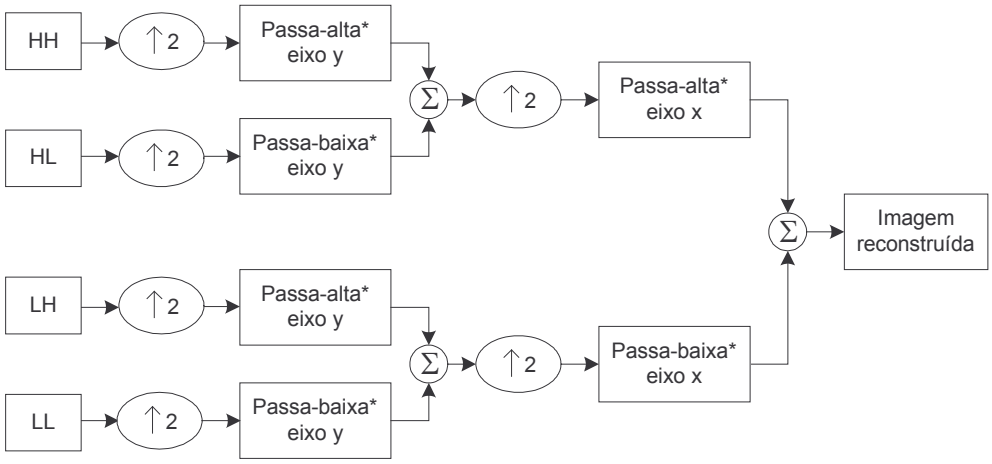


Fig. 2.5: *DWT 2D inversa.*

2.3.4 Coeficientes Wavelet

Nesta seção estaremos derivando os coeficientes dos filtros para o tipo de wavelet utilizado neste trabalho: a wavelet de Daubechies [17].

Caso Ortogonal – Daubechies

Para aplicar a DWT, precisamos encontrar a resposta impulsiva do filtro passa-baixa discreto h_k, denominado *vetor escala*. A partir de h_k pode-se gerar uma função relacionada $\phi(t)$, denominada *função escala*, definida como:

$$\phi(t) = \sum_k h_k \sqrt{2}\phi(2t-k)\,. \tag{2.14}$$

Conhecendo $\phi(t)$ e h_k, podemos usá-los para encontrar a *wavelet mãe*. A resposta impulsiva do filtro passa-alta discreto g_k, denominado *vetor wavelet*, pode ser determinado utilizando a seguinte relação:

$$g_k = (-1)^k h_{k-2N}$$

e a partir de g_k define-se a *wavelet mãe* $\psi(t)$ como:

$$\psi(t) = \sum_k g_k \sqrt{2}\phi(2t-k) \tag{2.15}$$

a partir do qual segue um conjunto de funções base ortonormal $\psi_{j,k}(t) = \sqrt{2^j}\psi(2^j t - k)$.

Partindo da equação 2.14, e integrando ambos os lados, teremos:

$$\int_{-\infty}^{\infty} \phi(t)dt = \int_{-\infty}^{\infty} \sum_k h_k \sqrt{2}\phi(2t-k)dt$$

substituindo $x = 2t - k$ e $dx = 2dt$ no lado direito:

$$\int_{-\infty}^{\infty} \phi(t)dt = \sum_k h_k \sqrt{2} \int_{-\infty}^{\infty} \phi(x)\frac{dx}{2} = \sum_k \frac{h_k}{\sqrt{2}} \int_{-\infty}^{\infty} \phi(x)dx$$

dividindo ambos os lados pela integral, obtemos:

$$\sum_k h_k = \sqrt{2} \ . \tag{2.16}$$

Usando a condição de ortogonalidade da função escala:

$$\begin{aligned}
\int |\phi(t)|^2 dt &= \int \sum_k h_k \sqrt{2}\phi(2t-k) \sum_m h_m \sqrt{2}\phi(2t-m)dt \\
&= \sum_k \sum_m h_k h_m 2 \int \phi(2t-k)\phi(2t-m)dt \\
&= \sum_k \sum_m h_k h_m \int \phi(x-k)\phi(x-m)dx
\end{aligned}$$

onde na última equação fizemos $x = 2t$ e $dx = 2dt$. A integral à direita é zero exceto quando $k = m$. Neste caso, teremos:

$$\sum_k h_k^2 = 1 . \tag{2.17}$$

Usando a ortogonalidade das translações da função escala, e substituindo o valor $\phi(t)$ pela função escala definida em 2.14, teremos:

$$\int \phi(t)\phi(t-m)dt = \delta_m$$

$$\int \left[\sum_k h_k \sqrt{2}\phi(2t-k) \right] \left[\sum_l h_l \sqrt{2}\phi(2t-2m-l) \right] dt$$

$$= \sum_k \sum_l h_k h_l 2 \int \phi(2t-k)\phi(2t-2m-l)dt$$

substituindo $x = 2t$ e $dx = 2dt$,

$$\int \phi(t)\phi(t-m)dt = \sum_k \sum_l h_k h_l \int \phi(x-k)\phi(x-2m-l)dx$$

$$= \sum_k \sum_l h_k h_l \delta_{k-(2m+l)} = \sum_k h_k h_{k-2m}$$

Portanto, temos a seguinte equação:

$$\sum_k h_k h_{k-2m} = \delta_m . \tag{2.18}$$

Usando as equações 2.16, 2.17 e 2.18, podemos gerar os coeficientes discretos h_k para a função escala. Para $k = 2$, das equações 2.16 e 2.17 temos que:

$$h_0 + h_1 = \sqrt{2}$$

$$h_0^2 + h_1^2 = 1$$

cuja única solução é $h_0 = h_1 = 1/\sqrt{2}$, que são os coeficientes da função escala de Haar, a mais simples das wavelets e sem muitos propósitos práticos.

Para $k = 4$, das equações 2.16, 2.17 e 2.18, temos que:

$$h_0 + h_1 + h_2 = \sqrt{2}$$
$$h_0^2 + h_1^2 + h_2^2 = 1$$
$$h_0 h_2 + h_1 h_3 = 0$$

As soluções para estas equações incluem os quatro coeficientes (comprimento do filtro igual a 4) da função escala de Daubechies, ou seja:

$$h_0 = \frac{1+\sqrt{3}}{4\sqrt{2}}, \quad h_1 = \frac{3+\sqrt{3}}{4\sqrt{2}}, \quad h_2 = \frac{3-\sqrt{3}}{4\sqrt{2}}, \quad h_0 = \frac{1-\sqrt{3}}{4\sqrt{2}}$$

Do mesmo modo para $k = 6 \ e \ k = 8$, obtemos os coeficientes das wavelets Daubechies6 e Daubechies8.

As tabelas 2.1, 2.2 e 2.3, mostram os coeficientes dos filtros da wavelet Daubechies4, Daubechies6 e Daubechies8. As figuras 2.6, 2.7 e 2.8 mostram respectivamente as funções escala e wavelet.

índice	passa-baixa *h*	passa-alta *g*
0	0.48296291	-0.12940952
1	0.83651631	-0.22414386
2	0.22414386	0.83651631
3	-0.12940952	-0.48296291

Tabela 2.1: *coeficientes para wavelet Daubechies W4.*

índice	passa-baixa *h*	passa-alta *g*
0	0.33267055	0.03522629
1	0.80689151	0.08544127
2	0.45987751	-0.13501102
3	-0.13501102	-0.45987751
4	-0.08544127	0.80689151
5	0.03522629	-0.33267055

Tabela 2.2: *coeficientes para wavelet Daubechies W6.*

índice	passa-baixa *h*	passa-alta *g*
0	0.23037781	-0.01059741
1	0.71484657	-0.03288301
2	0.63088077	0.03084138
3	-0.02798377	0.18703481
4	-0.18703481	-0.02798377
5	0.03084138	-0.63088077
6	0.03288301	0.71484657
7	-0.01059741	-0.23037781

Tabela 2.3: *coeficientes para wavelet Daubechies W8.*

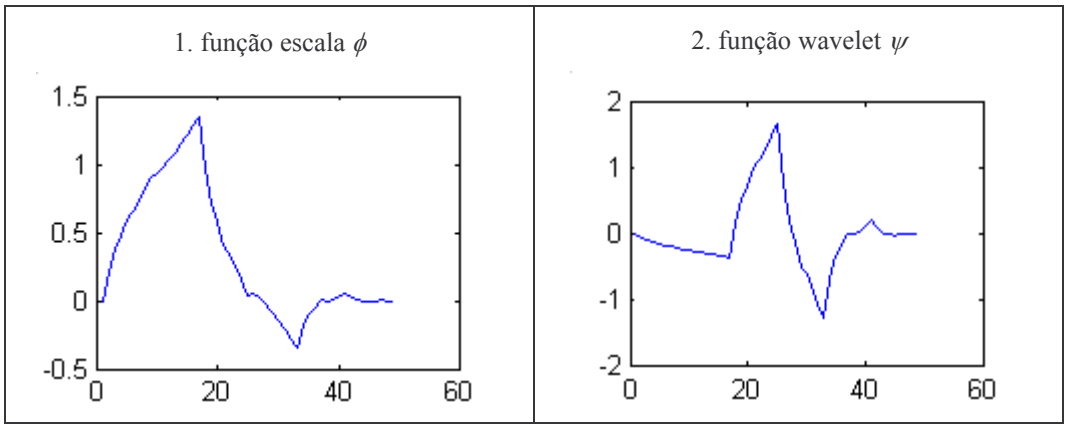


Fig. 2.6: *Daubechies W4 função escala e wavelet.*

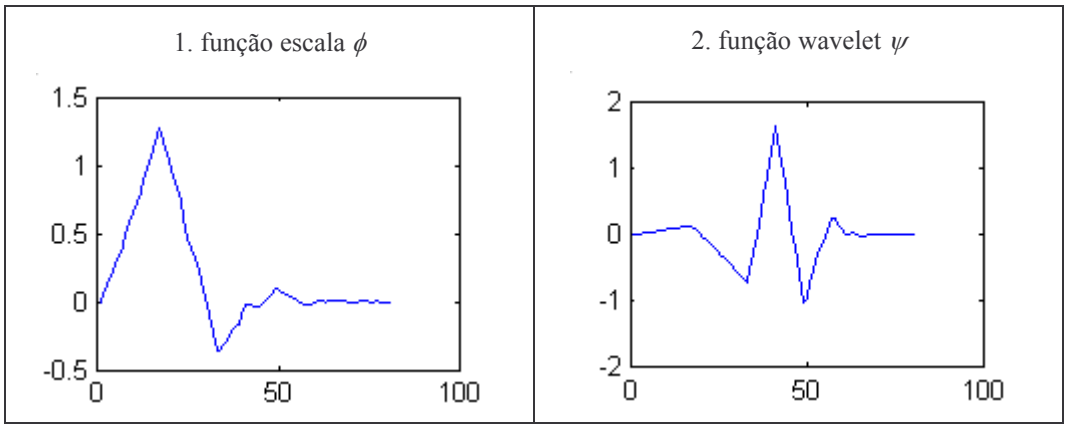


Fig. 2.7: *Daubechies W6 função escala e wavelet.*

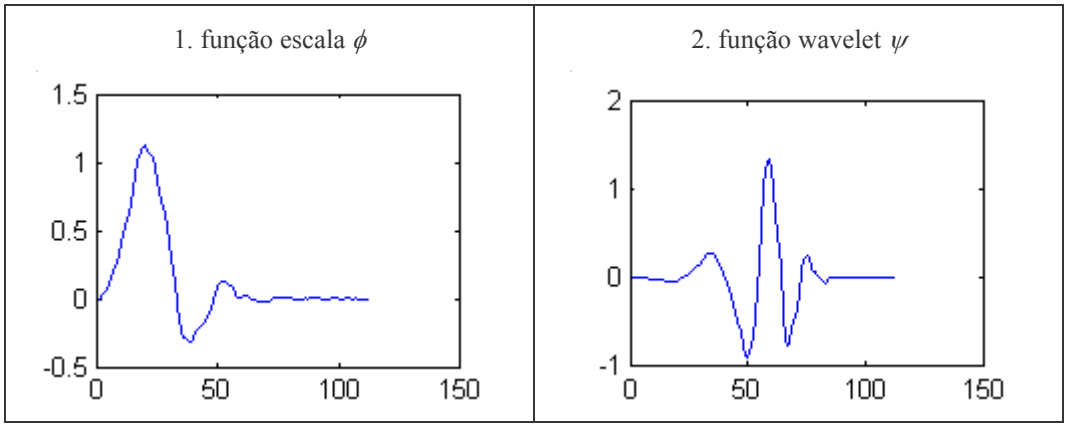


Fig. 2.8: *Daubechies W8 função escala e wavelet.*

2.4 RESUMO DO CAPÍTULO

Para entender a necessidade da transformada wavelet, neste capítulo, foi feita uma rápida revisão sobre a Transformada de Fourier, extensamente utilizada em processamento de sinais. Na seqüência do capítulo, foram apresentadas as transformadas wavelet contínua e discreta. Também foi apresentado um algoritmo para implementação computacional da transformada wavelet. O capítulo finaliza com uma formulação matemática, que nos leva a obter os coeficientes discretos dos filtros passa-baixa e passa-alta, utilizados na decomposição da imagem em múltiplas bandas. A wavelet trabalhada nesta tese é uma base ortogonal, proporcionando, portanto, algumas características interessantes. Primeiro, porque a função escala e a função wavelet são as mesmas para a transformada direta e para a transformada inversa. E segundo, porque a correlação do sinal em diferentes subbandas é removida.

CAPÍTULO 3 QUANTIZAÇÃO E CODIFICAÇÃO POR ENTROPIA

3.1 INTRODUÇÃO

O processo de compressão implementado neste trabalho consiste em três passos seqüenciais: computação da transformada wavelet, quantização e codificação por entropia. O processo de transformação da imagem foi discutido no capítulo 3. Este capítulo se concentrará na discussão dos processos de quantização e codificação. O capítulo começa com uma discussão sobre os modelos de quantização implementados, incluindo quantizadores uniformes e o algoritmo *EZW – Embedded Zerotree Wavelet* [29]. A seguir, são introduzidos alguns dos processos de codificação por entropia mais populares e eficientes existentes, incluindo o código de corrida, o código de Huffman e o código aritmético. Na seqüência, o capítulo apresenta uma alternativa efetiva de compressão: a codificação recursiva da seqüência, e finaliza apresentando as medidas de avaliação do desempenho dos sistemas compressores de imagens. O capítulo também apresenta os fluxogramas dos algoritmos de codificação.

3.2 QUANTIZAÇÃO

A quantização é um processo que consiste em mapear um conjunto contínuo de valores de entrada, x, para um conjunto discreto de valores de saída, y. Este processo consiste em reduzir o limite de precisão com que os dados estão sendo representados. Se os dados sendo quantizados serão representados por um número finito de bits, um número finito de níveis de quantização deve ser utilizado. Tipicamente, oito bits são utilizados para representar os dados quantizados. Neste caso, teremos, no máximo, 256 níveis de representação. Se cada elemento do conjunto é quantizado de forma independente, estamos falando em quantização escalar. Se dois ou mais elementos são quantizados conjuntamente, estamos falando em quantização vetorial [25], [26].

3.2.1 Quantização escalar

Existem diferentes formas de quantização escalar, destacando-se a quantização uniforme e a quantização de Lloyd-Max [27], [28]. Basicamente, o quantizador escalar divide o conjunto de valores entrada x em intervalos não sobrepostos limitados por amplitudes x_k, denominados níveis de decisão. O intervalo entre essas amplitudes x_k é denominado tamanho do passo de quantização, Δ.

Se um coeficiente wavelet cai dentro do intervalo $[x_k, x_{k+1})$, ele será especificado pelo símbolo k, o qual será então codificado para transmissão. No destino, os símbolos recebidos são mapeados para amplitudes r_k, conhecidos como níveis de reconstrução, conforme ilustra a figura 3.1.

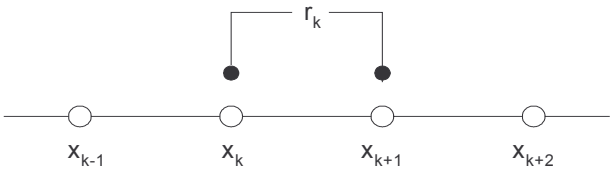


Fig. 3.1: *Descrição do quantizador.*

O processo de quantização é irreversível e introduz distorção. A distorção é diretamente proporcional ao quadrado do tamanho do passo de quantização. Quanto maior o passo de quantização, mais grosseiro é o mapeamento, portanto, mais distorcido será o sinal reconstruído. Porém, menos informação terá que ser transmitida, melhorando a taxa de bits por pixel, ou seja, aumentando o nível de compressão do sinal. O contrário também é verdadeiro. Quanto menor o passo de quantização, mais refinado é o mapeamento, menos distorcido será o sinal reconstruído, porém, maior é a taxa de bits por pixel, o que significa um menor nível de compressão do sinal.

Uma medida da distorção do sinal é o erro quadrático médio (*MSE – Mean Squared Error*), definido como:

$$\begin{aligned} MSE = E\left[(x - x_k)^2\right] &= \int_a^b (x - x_k)^2 \, p_x(x) dx \\ &= \frac{1}{M} \sum_{k=1}^{M} \int_{x_k}^{x_{k+1}} (x - x_k)^2 \, p_x(x) dx \end{aligned} \tag{3.1}$$

onde x e x_k representam, respectivamente, o sinal original e o reconstruído, $p_x(x)$ é a função densidade de probabilidade de x, e M é o tamanho do conjunto.

Quando a função densidade de probabilidade $p_x(x)$ é uniforme, a quantização ótima possui níveis de representação e decisão uniformemente distribuídos, definindo, portanto, uma *quantização uniforme*. Neste modelo, o número de níveis de quantização é o parâmetro mais relevante, determinando o tamanho do passo de quantização. A equação para computar o tamanho do passo de quantização Δ é:

$$\Delta = \left(max(x) - min(x)\right) / N \,, \tag{3.2}$$

onde *x* é o conjunto de entrada, e *N* é o número de níveis de quantização desejados. O sinal de entrada será dividido em *N+1* níveis de quantização, com cada nível de quantização cobrindo um intervalo igual ao tamanho do passo de quantização, conforme ilustra a figura 3.2**.**

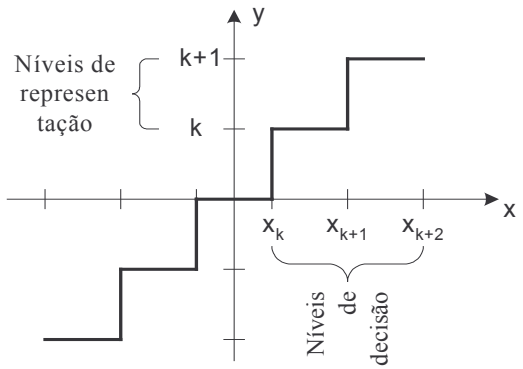


Fig. 3.2: *Quantização Uniforme.*

Neste processo, um coeficiente wavelet '*a*' será especificado pelo símbolo *k*, se residir na célula de partição:

$$k = x_k \le a < x_{k+1}, \qquad 1 \le k \le N+1. \tag{3.3}$$

O dequantizador usa o símbolo *k* para reconstruir o valor r_k, usando a seguinte equação para minimizar a distorção introduzida:

$$r_k = \left(x_k + x_{k+1}\right)/2. \tag{3.4}$$

Neste modelo de quantização, o valor do MSE, segundo [25], é aproximadamente igual a $\Delta^2/12$.

O quantizador ótimo será aquele que minimizar o MSE. A minimização da equação 4.1 foi investigada por Lloyd e Max [27], [28], que apresentaram as seguintes equações para os níveis de decisão e representação:

$$x_k = \frac{r_k + r_{k+1}}{2}, \qquad r_k = \frac{\int_{x_k}^{x_{k+1}} x.p_x(x)dx}{\int_{x_k}^{x_{k+1}} p_x(x)dx}, \qquad 1 \le k \le M \tag{3.5}$$

Esse conjunto de equações forma o algoritmo de Lloyd-Max. A primeira equação indica que os limiares de decisão devem estar no ponto médio entre os níveis de representação, enquanto a segunda indica que os níveis de representação ótimos são os centróides da função densidade de probabilidade nos intervalos apropriados. Trata-se de um procedimento iterativo para obtenção dos níveis de decisão e representação ótimos.

A maioria dos codificadores wavelet atuais utiliza quantização escalar como uma das etapas de codificação. Se a distribuição dos coeficientes em cada subbanda da imagem transformada fosse conhecida a priori, a estratégia ótima seria utilizar quantizadores Lloyd-Max em cada subbanda. Porém, em geral, não temos esse conhecimento [1].

3.2.2 Quantização uniforme com zona morta

Um quantizador mais simples e mais eficaz é o quantizador uniforme com zona morta. Embora em regimes práticos este quantizadores zona-morta são sub-ótimos, eles funcionam quase tão bem quanto os codificadores Lloyd-Max. Além disso, são simples de implementar e robustos a mudanças na distribuição dos coeficientes [1]. Conforme mostra o histograma dos coeficientes da transformada wavelet, figura 3.3, a maioria da energia da imagem está concentrada em poucos coeficientes. Os coeficientes da transformada podem ser aproximados a uma distribuição Gaussiana generalizada, ou Laplaciana [1], centrada em torno de zero. Assim, precisamos apenas manter os coeficientes de magnitude significante, fazendo o restante dos coeficientes iguais a zero, sem introduzir grande distorção na imagem reconstruída.

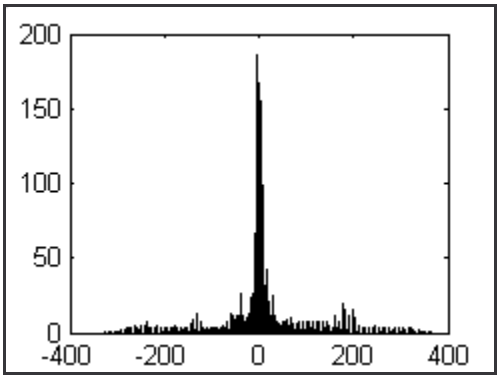


Fig. 3.3: *Histograma dos coeficientes da transformada wavelet (2 níveis de decomposição).*

A eliminação dos coeficientes de pequena magnitude é conseguida fazendo todos os coeficientes com valor abaixo de um certo limiar iguais a zero. A equação usada para esse propósito é [4]:

$$T(t,x) = \begin{cases} 0 & se\,|x| \le t \\ x & caso\,contrário \end{cases} \tag{3.6}$$

Após essa medida, aplicamos quantização uniforme nos coeficientes diferentes de zero. Esta técnica de quantização, faz com que a codificação com o código de corrida e o código aritmético, apresentem resultados interessantes na codificação das longas seqüências de zeros resultantes. A técnica é ilustrada pela figura 3.4.

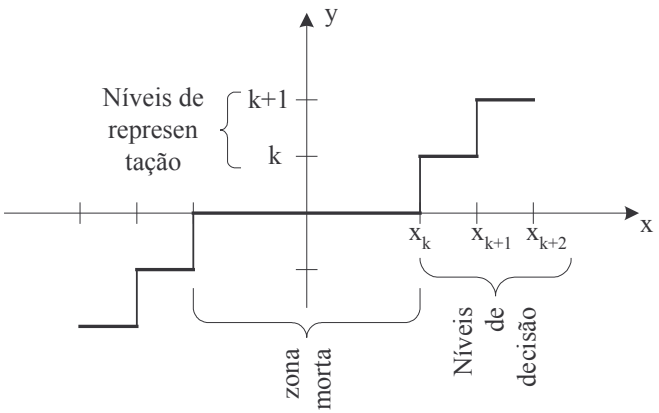


Fig. 3.4: *Quantizador uniforme com zona morta.*

3.3 O ALGORITMO EZW

O algoritmo *Embedded Zerotree Wavelet – EZW –* foi proposto por Shapiro [29], tornando-se um assunto extensivamente discutido na codificação de imagens com a transformada wavelet. É um sofisticado esquema de quantização, podendo ser considerado um tipo de quantizador vetorial, na medida que codifica vários coeficientes wavelet num único coeficiente (o coeficiente *zerotree* – árvore de zeros). É um algoritmo especialmente designado para trabalhar com a transformada wavelet, o que explica a palavra *wavelet* em seu nome. Foi originalmente proposto para codificar sinais bidimensionais (imagens), mas pode ser estendido para sinais de qualquer dimensão. É baseado num esquema de codificação progressiva, comprimindo uma imagem num fluxo de bits com precisão incremental. Isto significa que quanto mais bits são adicionados ao fluxo, mais detalhes são inseridos na imagem recuperada. Codificação progressiva é também sinônimo para codificação embarcada, o que explica a palavra *embedded* em seu nome. Com esse codificador os dados comprimidos podem alcançar a taxa de bits desejada, significando que se trata de um esquema de compressão com perdas. Também é possível utilizá-lo num esquema sem perdas.

O codificador EZW é baseado em duas importantes observações [30]:

1. as imagens, em geral, têm um espectro passa-baixa. Quando a transformada wavelet é aplicada, a energia das subbandas decresce conforme aumenta a escala. Isso significa que os coeficientes de magnitude significante estão concentrados nas subbandas de baixas freqüências, enquanto as subbandas de altas freqüências apenas adicionam detalhes, tornando a codificação progressiva uma escolha natural para compressão de imagens.

2. coeficientes wavelet de magnitude significante são mais importantes que os demais.

Estas duas observações são exploradas pelo codificador EZW, que codifica os coeficientes em ordem decrescente de magnitude, em várias passagens. A cada passagem, os coeficientes são comparados com um limiar previamente estabelecido e codificados. Se um coeficiente é maior que o limiar, em módulo, ele é especialmente codificado e retirado da imagem. Ao final de cada passagem, o limiar de comparação é reduzido e os coeficientes são novamente varridos e codificados. Esse processo é repetido até que todos os coeficientes wavelet tenham sido completamente codificados ou até outro critério ser satisfeito, como por exemplo, a taxa máxima de bits ser atingida.

A idéia é usar a dependência entre os coeficientes wavelet em diferentes escalas para codificar grandes partes da imagem que estão abaixo do limiar atual. É aqui que as árvores de zeros – *zerotrees* – aparecem. Cada coeficiente numa subbanda mais baixa é considerado como tendo quatro descendentes na próxima subbanda mais alta (de mesma orientação espacial). Cada um dos quatro descendentes também tem, por sua vez, quatro descendentes na próxima subbanda mais alta (figura 3.5). Assim temos a explicação para a palavra *zerotree*: uma árvore onde todos os nós descendentes são considerados iguais ou menores que o nó raiz. A árvore é codificada com um único símbolo e reconstruída pelo decodificador como uma árvore de zeros. Precisamos adicionar que o nó raiz será sempre menor que o limiar comparativo.

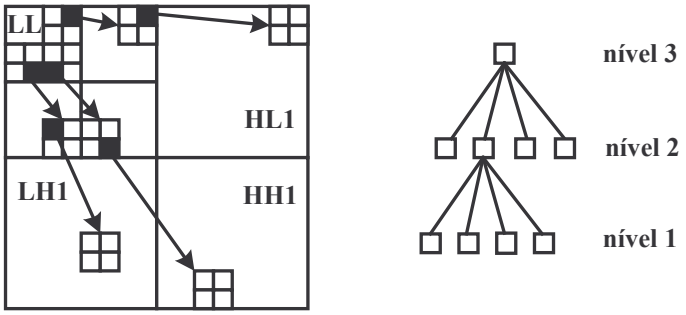


Fig. 3.5: *Relação* entre *os coeficientes wavelet em diferentes subbandas.*

Teoricamente, precisaríamos também codificar as posições dos coeficientes, o que é totalmente impraticável. Sem essa informação o codificador não é capaz de reconstruir o sinal. Porém, a codificação das posições é automática, quando se varre a imagem numa ordem pré-definida. Diferentes opções para a varredura são possíveis (figura 3.6), contanto que as subbandas mais baixas sejam completamente varridas antes das subbandas mais altas.

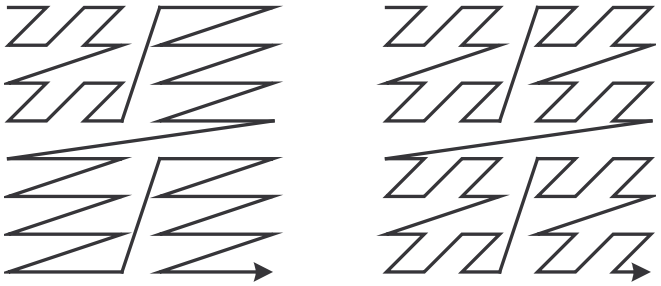


Fig. 3.6: *Técnicas de varredura.*

O primeiro passo do algoritmo EZW consiste em determinar o limiar inicial. Esse limiar inicial T_0 será:

$$T_0 = 2^{\lfloor log_2(abs(max(x))) \rfloor}, \tag{3.7}$$

onde *abs(max(x))* corresponde ao valor absoluto do coeficiente wavelet máximo, e $\lfloor . \rfloor$ significa aproximação para o menor inteiro. A seguir, os coeficientes passam por um processo de Quantização por Aproximação Sucessiva – QAS, onde dois passos são executados, um passo dominante e um passo subordinado. O passo dominante codifica todos os coeficientes com respeito ao limiar e o passo subordinado faz a QAS dos coeficientes identificados como significantes, ou seja, aqueles maiores que o limiar, em módulo. A cada passagem dominante e subordinada, o limiar é reduzido por 2 e os dois passos são novamente executados.

Durante o passo dominante todos os coeficientes são codificados de acordo com quatro símbolos:

- **POS (P):** o coeficiente é maior que o limiar.
- **NEG (N):** o coeficiente é menor que menos o limiar.
- **ZTR (T):** o coeficiente é um nó raiz de uma *zerotree*, sendo, portanto, menor que o limiar.
- **IZ (Z):** o coeficiente é um zero isolado, ou seja, o coeficiente é menor que o limiar, mas não é a raiz de uma *zerotree*, porque em sua árvore de descendentes existe um coeficiente maior que o limiar. Note que para identificar se um coeficiente é um nó raiz de uma *zerotree* ou um zero isolado, toda a árvore de descendentes precisa ser varrida, o que torna o processo computacionalmente custoso.

Finalmente, aqueles coeficientes identificados como significantes são extraídos, colocados sem sinal numa lista subordinada e substituídos por zeros, para que não sejam novamente codificados na próxima passagem dominante. A cada passagem dominante, os novos coeficientes identificados como significantes são colocados no final da lista subordinada. Cada passagem dominante é seguida por uma passagem subordinada, que faz o refinamento de todos os coeficientes colocados na lista subordinada pelo passos dominantes anteriores. Esta etapa consiste simplesmente em enviar ao fluxo de bits de saída, o próximo bit mais

significante dos coeficientes na lista subordinada. O processo termina quando o limiar atinge seu valor mínimo, ou quando uma outra condição seja satisfeita, como distorção máxima permitida ou caso a taxa de bits desejada seja atingida [29], [30].

3.3.1 Exemplo da codificação EZW

Nesta seção estaremos discutindo o algoritmo apresentado na seção anterior através de um exemplo. Considere que a matriz bidimensional da figura 3.7 é o resultado da transformada wavelet de uma imagem 8x8 aplicada até o terceiro nível de decomposição. Segundo a equação 3.7 o limiar inicial T_0 é igual a 32.

63	-34	49	10	7	-13	-12	7
-31	23	14	-13	3	4	6	-1
15	14	3	-12	5	-7	3	9
-9	-7	-14	8	4	-2	3	2
-5	9	-1	47	4	6	-2	2
3	0	-3	2	3	-2	0	4
2	-3	6	-4	3	6	3	6
5	11	5	6	0	3	-4	4

Fig. 3.7: *Coeficientes wavelet de uma imagem 8x8.*

POS	NEG	POS	ZTR	ZTR	ZTR	x	x
IZ	ZTR	ZTR	ZTR	ZTR	ZTR	x	x
ZTR	IZ	x	x	x	x	x	x
ZTR	ZTR	x	x	x	x	x	x
x	x	ZTR	POS	x	x	x	x
x	x	ZTR	ZTR	x	x	x	x
x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x

Fig. 3.8: *Resultado do primeiro passo dominante.*

A figura 3.8 mostra o resultado do primeiro passo dominante. O algoritmo implementado varre a imagem segundo um rastreamento *Morton*. Portanto, o fluxo de bits será, após o primeiro passo dominante `D1: PNZT.PTTT.TZTT.TTTT.TPTT.` Os pontos foram adicionados para simplicidade de entendimento, e para cada símbolo temos um código de dois bits correspondente. Os símbolos –31 e 14 foram identificados como *zeros isolados* (**IZ**), porque são menores que o limiar, mas em sua árvore de descendência existe um coeficiente com valor maior que o limiar, aquele com valor 47 (**POS**). Já para aqueles coeficientes classificados como *zerotree* (**ZTR**), temos que são menores que o limiar, mas sem coeficientes significantes em sua árvore descendente.

Aqueles coeficientes que foram codificados como **POS** ou **NEG**, são retirados da matriz, substituídos por zero, para que não sejam novamente codificados, e colocados numa lista subordinada para que sofram um processo de refinamento. O processo de refinamento consiste em enviar ao fluxo de bits de saída o próximo bit mais significante dos coeficientes na lista. A tabela 3.1 a seguir mostra o resultado do primeiro passo subordinado e o valor de reconstrução dos coeficientes.

Lista subordinada	63 = 00111111	34 = 00100010	49 = 00110001	47 = 00101111
Próximo bit significante: bit 5	1	0	1	0
Valor de reconstrução	48	-40	48	40

Tabela 3.1: *Primeiro passo subordinado.*

As figuras (3.9) e (3.10) a seguir, mostram o resultado do segundo passo dominante, onde o limiar agora vale 16, metade do valor original. Note que os coeficientes considerados significantes no passo anterior foram substituídos por zero.

0	0	0	10	7	-13	-12	7
-31	23	14	-13	3	4	6	-1
15	14	3	-12	5	-7	3	9
-9	-7	-14	8	4	-2	3	2
-5	9	-1	0	4	6	-2	2
3	0	-3	2	3	-2	0	4
2	-3	6	-4	3	6	3	6
5	11	5	6	0	3	-4	4

Fig. 3.9: *Matriz de coeficientes após o primeiro passo dominante.*

IZ	ZTR	x	x	x	x	x	x
NEG	POS	x	x	x	x	x	x
ZTR	ZTR	ZTR	ZTR	x	x	x	x
ZTR	ZTR	ZTR	ZTR	x	x	x	x
x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x

Fig. 3.10: *Resultado do segundo passo dominante.*

O seguinte fluxo de bits, produzido no segundo passo dominante, será adicionado ao fluxo de bits de saída: `D2: ZTNP.TTTT.TTTT.`

A tabela 3.2 mostra o resultado do segundo passo subordinado:

Lista subordinada	63	34	49	47	31	23
Próximo bit significante: bit 4	1	0	0	1	1	0
Valor de reconstrução	56	-32	48	40	-24	16

Tabela 3.2: *Segundo passo subordinado.*

Seguindo esse procedimento até sua conclusão, reduzindo o limiar por 2 e alternando entre passos dominantes e subordinados, teríamos que a matriz de coeficientes seria exatamente recuperada. Porém, geralmente o processo é interrompido assim que alguma condição desejada seja atingida. As figuras (3.11) e

(3.12) mostram como a matriz seria reconstruída após três e quatro passos dominantes e subordinados, respectivamente. Com cinco iterações a matriz seria recuperada exatamente.

Para que o processo de decodificação seja bem sucedido, duas informações precisam ser anexadas aos dados codificados: o limiar inicial e o tamanho da imagem. Com essas informações, uma matriz de zeros é construída com as mesmas dimensões da imagem original. O processo segue agora o caminho inverso, novamente executando alternadamente um passo dominante e um passo subordinado e reduzindo o limiar. Cada passo dominante e subordinado decodifica o fluxo de bits produzido pelo correspondente passo de codificação. As posições dos coeficientes foram implicitamente codificadas pela varredura Morton. A cada passo, os coeficientes vão sendo quantizados inversamente e se aproximando de seu valor original. Cada vez que um símbolo *zerotree* é decodificado, sua árvore descendente é preenchida com zeros. Quando um coeficiente **POS** ou **NEG** é decodificado, ele recebe inicialmente o valor do limiar, ou de menos o limiar, respectivamente, e é refinado pelos bits decodificados no passo subordinado correspondente. Quando um coeficiente *zero isolado* é decodificado, é atribuído a ele o valor zero. Sua árvore de descendentes está codificada no restante do fluxo e será decodificada na seqüência do processo.

60	-32	48	8	0	12	-12	0
-28	20	12	-12	0	0	0	0
12	12	0	-12	0	0	0	8
-8	0	-12	8	0	0	0	0
0	8	0	44	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	8	0	0	0	0	0	0

Fig. 3.11: *Matriz recuperada após 3 iterações.*

62	-34	48	10	6	12	-12	6
-30	22	14	-12	0	4	6	0
14	14	0	-12	4	-6	0	8
-8	-6	-14	8	4	0	0	0
-4	8	0	46	4	6	0	0
0	0	0	0	0	0	0	4
0	0	6	-4	0	6	0	6
4	10	4	6	0	0	-4	4

Fig. 3.12: *Matriz recuperada após 4 iterações.*

3.4 CODIFICAÇÃO POR ENTROPIA

As técnicas de codificação por entropia são utilizadas para reduzir as redundâncias presentes no sinal. A codificação por entropia é usada independentemente das características do sinal. O fluxo de dados a ser comprimido é considerado como uma seqüência digital simples, e a semântica dos dados é ignorada. A codificação por entropia é um exemplo de codificação sem perdas, e o processo de descompressão regenera os dados completamente. Os dados antes do processo de compressão e depois da descompressão são idênticos, nenhuma informação é perdida.

A entropia ou *incerteza*, *H*, de uma fonte de informações que gera um alfabeto discreto de símbolos *m,* é dada, segundo [2], por:

$$H(z) = -\sum_{j=1}^{m} P_j . \log_2 \left(P_j \right), \tag{3.8}$$

onde P_j é a probabilidade de ocorrência do símbolo j. A entropia define a quantidade média de informação obtida pela observação de uma única saída da fonte, em bits por símbolo, para o caso de uma fonte binária. É um limite inferior do comprimento médio das palavras de código [32]. A entropia é limitada por $0 \le H \le log_2 \left(m \right)$, ou seja, quando $H = 0$, implica que a fonte não fornece nenhuma informação, em média, de modo que não há nenhuma incerteza quanto à mensagem. No outro extremo, $H = log_2 \left(m \right)$, a entropia máxima corresponde a incerteza máxima, o que implica que todos os símbolos são igualmente prováveis [2].

As técnicas de codificação por entropia caem em duas classes: supressão de seqüências repetitivas (código de corrida) e codificação estatística (código de Huffman e código aritmético). Estaremos discutindo esses três casos nas seções a seguir.

3.4.1 Código de corrida

A supressão de seqüências repetitivas, *RLE – Run Length Encoding,* é o primeiro método baseado na entropia, sendo o mais simples e um dos mais antigos usados na computação. Consiste em detectar e codificar seqüências consecutivas de um mesmo símbolo, substituindo essas ocorrências por dois caracteres, um consistindo no símbolo sendo codificado e outro, indicando o número de ocorrências do símbolo. Existem variações na maneira de implementação, mas todas mantendo a idéia principal, que é a codificação das corridas repetidas.

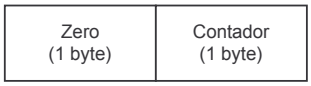


Fig. 3.13: *Código de corrida.*

No caso dos coeficientes wavelet onde a quantização uniforme com zona morta é usada, as longas seqüências de zeros resultantes serão codificadas segundo esse modelo. As corridas de zeros serão substituídas pelo código de corrida de dois símbolos mostrado na figura 3.13. Qualquer que seja o número de ocorrências de zeros, a corrida deve ser substituída pelo código indicado. A compressão efetivamente ocorre quando temos corridas com mais de três zeros, conforme ilustra o exemplo 3.1.

Exemplo 3.1:

Seqüência original:

5 0 0 0 0 6 7 8 0 0 0 0 0 0 0 8 0 0 0

Seqüência codificada:

5 [0 4] 6 7 8 [0 6] 8 [0 3]

Note que codificar zeros conjuntamente, enquanto sendo simples e inteiramente prático, está fora dos limites da quantização escalar. Na verdade, o código de corrida de zeros pode ser considerado um caso especial de quantização vetorial. Ele captura redundâncias além do que é possível, mesmo com uma transformada ótima e alocação de taxa de bits ótima [1]. Este tema de codificar zeros conjuntamente cai dentro do contexto da codificação por árvore de zeros dos coeficientes wavelet, e é usado para gerar algoritmos de codificação alternativos, como o desenvolvido em [29].

O *código* de *corrida* é uma alternativa efetiva para a codificação de áreas constantes. Embora a codificação por código de corrida seja um método efetivo de compressão de imagens por si mesmo, compressão adicional pode normalmente ser alcançada por codificação subseqüente dos próprios códigos de corrida [2]. Esta alternativa será explorada neste trabalho, quando o código de corrida precederá a codificação de Huffman e aritmética.

3.4.1.1 Fluxograma

Esta seção apresenta o fluxograma do algoritmo descrito na seção anterior. O algoritmo implementado detecta seqüências consecutivas de zeros e as substitui pelo código de corrida mostrado na figura 3.13. Este algoritmo será usado em conjunção com o código de Huffman e aritmético na tentativa de aumentar o nível de compressão do sinal.

As figuras 3.14a e 3.14b ilustram o algoritmo de codificação e decodificação.

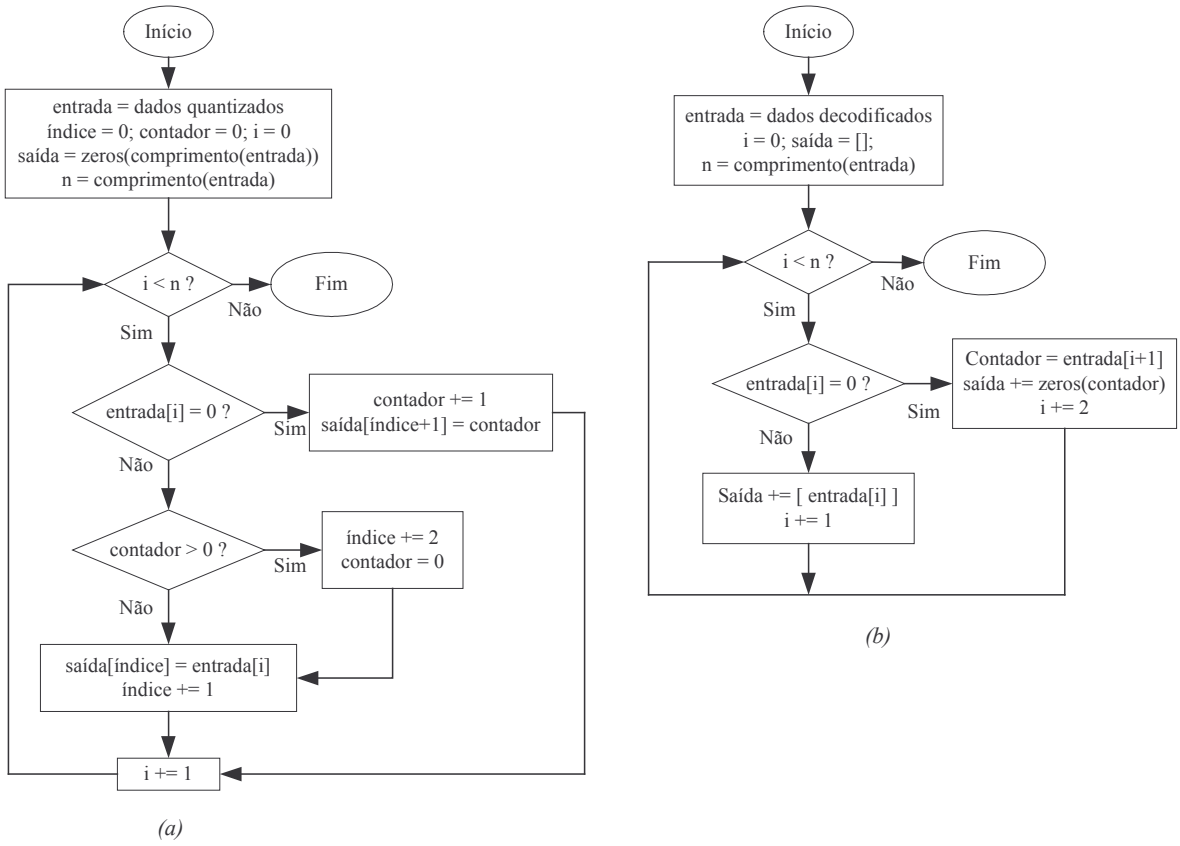


Fig. 3.14: *Fluxograma do código de corrida: a) codificador b) decodificador.*

3.4.2 Código de Huffman

O código de Huffman é uma técnica de compressão estatística. Dado um caractere que deve ser codificado, junto com sua probabilidade de ocorrência, o algoritmo de Huffman determina o código ótimo usando um número mínimo de bits. Símbolos com probabilidade de ocorrência alta são representados por palavras código curtas, enquanto símbolos com probabilidade de ocorrência baixa são representados por palavras código longas. A palavra código representativa de cada símbolo tem comprimento variável, sendo representada por um número inteiro de bits. O código de Huffman é o melhor esquema de codificação possível, sob a restrição de que os símbolos sejam codificados um por vez [2].

O modelo do codificador de Huffman, mostrado na figura 3.15, é mais bem entendido como uma árvore binária. A palavra código será formada por uma seqüência de zeros e uns que vai da "raiz" até a "folha" correspondente ao símbolo sendo codificado. Para codificar os dados quantizados, o primeiro passo na abordagem de Huffman é calcular a freqüência de ocorrência de cada símbolo. O próximo passo é a criação de uma série de reduções de fonte, através da ordenação das probabilidades dos símbolos e da combinação dos dois símbolos de menor probabilidade em um novo símbolo (nó), com probabilidade igual a soma das probabilidades dos símbolos fonte, que os substitua na próxima redução de fonte. Atribui-se aos dois símbolos

usados para gerar uma redução de fonte o código binário 0 e 1. Esse processo é repetido até que uma fonte reduzida com dois símbolos (à direita) seja atingida. O passo final é a codificação de cada fonte reduzida, começando com a menor fonte (raiz) e caminhando para trás até a fonte original (folhas).

Uma vez que o código tenha sido criado, a codificação e/ou decodificação é realizada de uma maneira do tipo 'look-up table'. O próprio código é um código de bloco instantaneamente decodificável de maneira única. É chamado de *código de bloco*, porque cada símbolo fonte é mapeado em uma seqüência fixa de símbolos de código. É *instantâneo*, porque cada palavra código em uma cadeia de símbolos pode ser decodificada sem referência aos símbolos sucessivos. É *unicamente decodificável*, porque qualquer cadeia de símbolos de código pode ser decodificada de maneira única. Portanto, uma cadeia de símbolos codificados por Huffman pode ser decodificada através do exame dos bits da cadeia da esquerda para a direita [2].

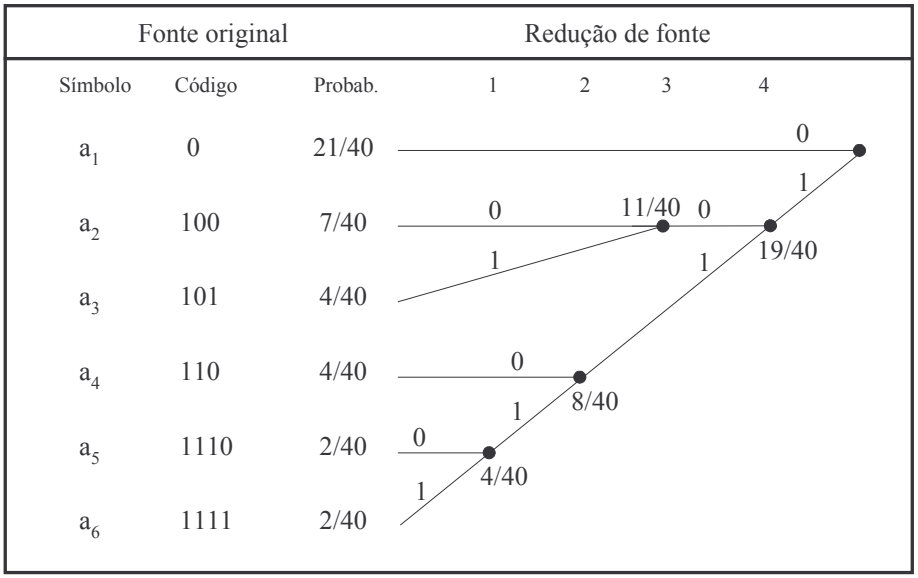


Fig. 3.15: *Código de Huffman derivado de uma árvore binária.*

Este método implica na recodificação de padrões, portanto, uma tabela de correspondência precisa ser incluída no arquivo comprimido como informação lateral. Existe um consumo extra de bits para enviar esta tabela no cabeçalho dos dados e um dos objetivos deste trabalho foi minimizar a quantidade de informação lateral produzida. Esta informação lateral consiste na codificação do comprimento das respectivas palavras código de cada símbolo. Isso permite a reconstrução das palavras código no destino e a seqüência pode ser eficientemente decodificada. Significando, portanto, que o algoritmo implementado deriva o código de Huffman a partir do comprimento de cada palavra código. Essa tabela será referenciada desse ponto em diante como *tabela de Huffman*.

3.4.2.1 Especificação eficiente da tabela de Huffman

O método proposto para codificar a tabela de Huffman usa 5 bits para codificar o comprimento da primeira palavra código, e então para cada um dos símbolos seguintes, um código para dizer seu comprimento, conforme a tabela 3.3.

Esta maneira de codificar a tabela de Huffman, utiliza o fato que símbolos adjacentes têm aproximadamente a mesma probabilidade, e assim aproximadamente o mesmo comprimento para as palavras-código [31].

Código	Significado
0	Mesmo comprimento que o símbolo anterior
10	Comprimento incrementou em 1
1100	Reduziu comprimento em 1
1101	Comprimento incrementou em 2
111xxxxx	Atribua comprimento do símbolo a xxxxx

Tabela 3.3: *Tabela de codificação dos comprimentos das palavras de Huffman.*

3.4.2.2 Fluxograma

Para codificar o sinal usando o código de Huffman, a probabilidade de ocorrência dos símbolos na imagem precisa ser calculada. Essa informação pode ser extraída do histograma. Portanto, inicialmente, o histograma é calculado, e com essa informação derivamos o comprimento de cada palavra código. O segundo passo é calcular as palavras código, a partir dos comprimentos calculados na etapa anterior. O passo seguinte consiste em codificar o sinal e toda informação lateral necessária à decodificação. Essa informação lateral consiste no comprimento do sinal, no comprimento do vetor palavras código e no comprimento de cada palavra código. Na há necessidade de codificar as palavras código porque o algoritmo é capaz de reconstruí-las no destino a partir do comprimento de cada palavra código. Finalmente, a codificação do sinal é executada através de um mapeamento símbolo palavra código.

A figura 3.16 mostra a seqüência de implementação do código de Huffman.

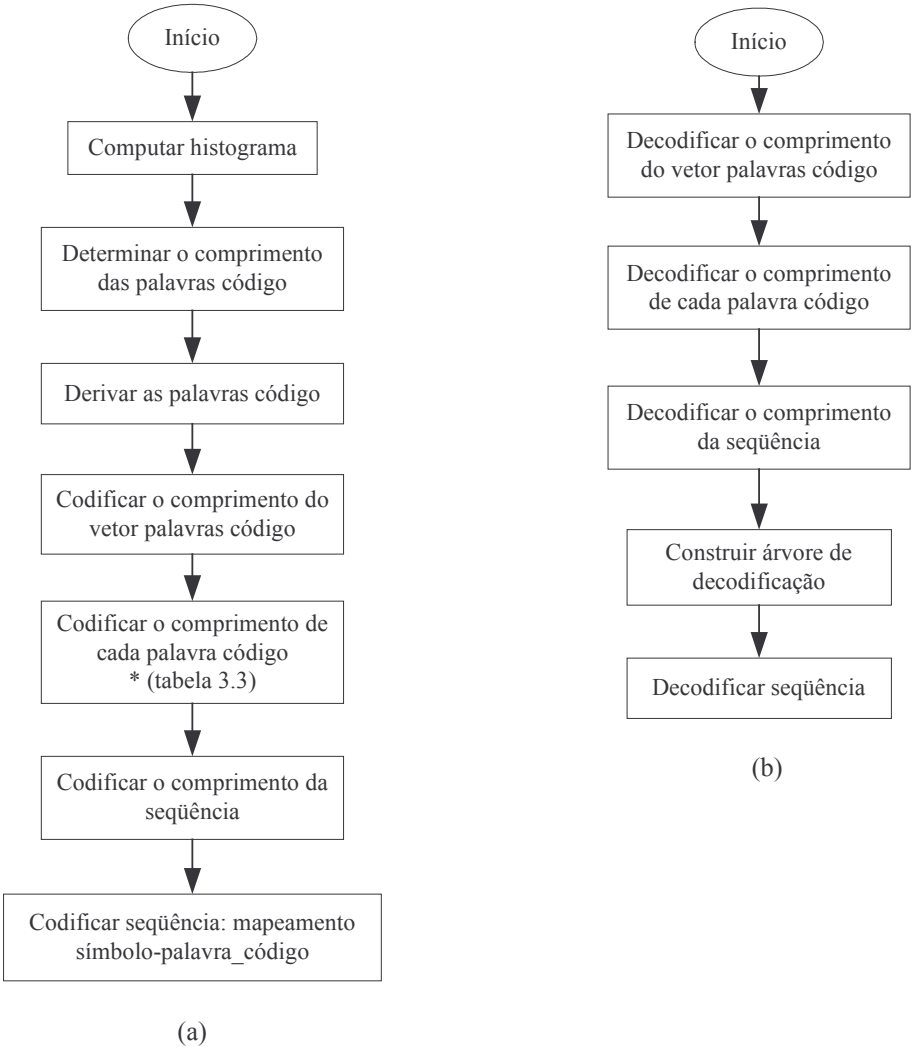


Fig. 3.16: *Fluxograma do código de Huffman: a) codificador b) decodificador.*

3.4.3 Código aritmético

O código aritmético supera completamente a idéia de substituição de um símbolo de entrada por um símbolo específico. Ao invés, ele pega um fluxo de símbolos e o substitui por um único número em ponto flutuante. Quanto maior e mais complexa a mensagem, mais bits são necessários no número de saída. A saída do código aritmético é um único número menor que 1 e maior ou igual a zero.

A idéia do código aritmético é ter uma linha de probabilidade, entre 0 e 1, e atribuir a cada símbolo, um intervalo nesta linha baseado na probabilidade do símbolo. Por exemplo, se estivéssemos codificando a mensagem aleatória “CASA”, teríamos a seguinte atribuição de intervalos a cada símbolo:

Caractere	A	C	S
Probabilidade	0.50	0.25	0.25
Intervalo	[0.00 – 0.50)	[0.50 – 0.75)	[0.75 – 1.00)

Tabela 3.4: *Atribuição de intervalos aos símbolos da mensagem "CASA".*

Cada caractere teve atribuído uma porção da faixa 0-1 correspondente as suas probabilidades. Note que cada caractere terá todo o intervalo, não incluindo o limite superior. Na dinâmica do código aritmético, o que é feito, é seguir o intervalo onde cada símbolo se encontra, sendo que cada símbolo codificado, restringe o intervalo do símbolo seguinte. Por exemplo, quando codificando a mensagem "CASA", o primeiro símbolo, "C", define que a mensagem codificada será um número entre 0.50 e 0.75. O próximo símbolo a ser codificado, "A", pertence agora, ao subintervalo 0.0-0.50 dentro do novo intervalo 0.50-0.75. Isso significa que ele estará em alguma posição entre 0 e 50% do intervalo corrente. Aplicando esta lógica, restringirá o número de saída para o intervalo 0.50-0.625.

O algoritmo para uma mensagem de qualquer comprimento [35] é descrito abaixo:

- *limite inferior = 0*
- *limite superior = 1*
- *enquanto existirem símbolos de entrada :*
  - *intervalo = limite superior – limite inferior*
  - *limite superior = limite inferior + (intervalo * limite superior do símbolo)*
  - *limite inferior = limite inferior + (intervalo * limite inferior do símbolo)*
- *fim enquanto*
- *mensagem codificada = limite inferior*

Seguindo esse processo até sua conclusão, a codificação da mensagem "CASA" seria:

Símbolo	intervalo	limite superior	limite inferior
		1.0	0.0
C	1.0	0.75	0.50
A	0.25	0.625	0.50
S	0.125	0.625	0.59375
A	0.03125	0.609375	0.59375

Tabela 3.5: *Codificação aritmética da mensagem "CASA".*

Então, o limite inferior final, 0.59375, codificará a mensagem “CASA” .

Na decodificação, encontra-se o primeiro símbolo codificado, observando o intervalo da mensagem codificada. Desde que o número codificado 0.59375, está entre 0.50 e 0.75, o intervalo do símbolo “C”, este será o primeiro caractere decodificado. Sabendo-se os limites superior e inferior de “C”, pode-se remover seus efeitos pela inversão do processo que o colocou no número codificado, primeiro, subtraindo o limite inferior do símbolo (0.50) da palavra codificada, resultando em 0.09375, e a seguir dividindo pelo intervalo do símbolo (0.25), resultando em 0.375. Agora, repita o processo para esse novo número, decodificando “A” e assim sucessivamente.

O algoritmo de decodificação para o número de chegada será algo como:

- *número codificado = mensagem codificada*
- *para todos os símbolos:*
    - *intervalo = limite superior do símbolo – limite inferior do símbolo*
    - *número codificado = (número codificado – limite inferior do símbolo) / intervalo*
- *fim para*

3.4.3.1 Implementação

Na implementação computacional, nenhuma operação em ponto flutuante se faz necessária. Demanda-se a conversão da linha de probabilidade para trabalho com registradores inteiros, com 16 bits por exemplo. Assim, o limite superior será convertido para 0xFFFF, que em binário corresponde a tudo “1” no registro de 16 bits, e o limite inferior para zero, onde *0x* denota número em formato hexadecimal. Desse modo, teríamos a seguinte atribuição de intervalos a cada símbolo:

Caractere	A	C	S
Intervalo	[0 – 0x8000)	[0x8000 – 0xC000)	[0xC000 – 0xFFFF)

Tabela 3.6: *Atribuição de intervalos inteiros.*

Prova:

- 0x8000/0xFFFF = 32768/65536 = 0.50
- 0x4000/0xFFFF = 16384/65536 = 0.25
- 0xC000/0xFFFF = 49152/65536 = 0.75

Então se aplica o algoritmo descrito, observando-se alguns detalhes. Primeiro, o cálculo do intervalo precisa ser incrementado, porque se assume que o registro superior tem um número infinito de “uns”.

Segundo, deve-se tomar o cuidado de não atribuir ao limite superior do símbolo o extremo do intervalo. Assim, o limite superior do símbolo será decrementado. Devido à natureza deste algoritmo, os limites superior e inferior crescem próximos um ao outro sem nunca se encontrarem. Isto significa que uma vez que casarem no bit mais significante (*MSB – Most Significant Bit*), esse bit nunca mudará. Codifica-se então esse bit, fazendo o deslocamento dos limites superior e inferior para a esquerda um dígito, e incrementando o MSB do limite superior, pois esse registro tem um número infinito de "uns".

3.4.3.2 Situação crítica

Existe uma situação crítica em potencial, que ocorre quando os limites superior e inferior estão convergindo para um valor, mas sem seus MSB casarem. Por exemplo, quando o limite superior é 7004 e o limite inferior é 6995. Neste ponto, o intervalo calculado vai ter apenas um dígito, o que significa que a palavra de saída não tem precisão suficiente para ser codificada. Ainda pior, depois de algumas iterações, o limite superior será 7000 e o inferior 6999. Neste ponto, os valores estão permanentemente travados, e não podemos codificar nenhum dígito.

Para detectar essa situação, testamos o segundo MSB dos limites. Se for zero para o limite superior e 1 para o limite inferior, a situação crítica foi detectada. Para superá-la, excluímos o segundo MSB e, deslocamos o restante dos dígitos à esquerda para ocupar esse espaço. Os MSB dos limites não mudam. Finalmente, quando os MSB casarem, aqueles dígitos que foram descartados serão agora codificados.

3.4.3.3 Modelamento estatístico adaptativo

Para comprimir os dados usando o código aritmético, um modelo para os dados é necessário. O modelo precisa ser capaz de fazer duas coisas para efetivamente comprimir os dados:

- Predizer precisamente a probabilidade ou freqüência de ocorrência dos símbolos, e;
- Que as probabilidades geradas pelo modelo desviem de uma distribuição uniforme.

A necessidade de precisamente predizer a probabilidade dos símbolos de entrada é inerente da natureza do código aritmético. O objetivo do código é reduzir o número de bits necessário para codificar um caractere conforme sua probabilidade de aparência aumenta. A segunda condição é que o modelo precisa ser capaz de fazer predições que desviem de uma distribuição uniforme de probabilidades. Quanto melhor é o modelo em predizer as probabilidades que desviem da distribuição uniforme, melhores serão as taxas de compressão. Apenas encontrando corretamente as probabilidades que desviem de uma distribuição normal é que o número de bits pode ser reduzido, levando à compressão. O incremento das probabilidades tem que precisamente refletir a realidade, desde que respeitando a primeira condição.

Para que o modelo se adapte precisamente à ocorrência de símbolos, duas tabelas são necessárias, contendo contadores que são utilizados na predição das probabilidades. A tabela 1 mantém registro dos símbolos que foram codificados, enquanto a tabela 2 é utilizada para codificar novos símbolos que aparecem na seqüência. Quando um novo símbolo aparece, é emitido um código de escape, e o símbolo é codificado pela tabela 2. Se o símbolo a ser codificado já foi anteriormente codificado, utiliza-se a tabela 1.

Para a rotina do código aritmético, o símbolo é passado como um contador alto, um contador baixo e uma escala, ao invés dos intervalos de probabilidades tradicionais. A escala registra o total de símbolos que já foram codificados (na tabela 1), ou quantos símbolos ainda precisam ser codificados (na tabela 2). Esses valores são extraídos das tabelas na codificação de cada símbolo; novos símbolos são codificados pela tabela 2. Sugerimos ao leitor consultar o exemplo 3.2, que descreve a criação das tabelas e a atualização dos contadores de símbolo e do fator escala. Nos parágrafos seguintes, estaremos descrevendo a função de cada uma das tabelas.

A tabela 1 tem comprimento igual ao valor máximo da seqüência mais dois, com contadores iniciados em 1. Cada vez que um símbolo é codificado usando esta tabela, seus contadores e escala são atualizados. O contador alto e a escala são incrementados, enquanto o contador baixo se mantém constante. O valor do símbolo será o endereço para o contador alto; o contador baixo é o valor contido no endereço seguinte; a escala é sempre a primeira posição da tabela.

Na dinâmica de codificação de um símbolo, seus contadores são verificados. Se forem iguais, trata-se de um símbolo diferente dos anteriores sendo codificado. Então, é emitido um código de escape, e o símbolo é codificado pela tabela 2. Quando o valor dos contadores é diferente, seus valores e a escala são extraídos e passados para a rotina de codificação, que identifica o intervalo do símbolo e os seus limites superior e inferior. O próximo passo é a atualização da tabela, que consiste no incremento dos valores de todas as posições até o endereço do contador alto. Com essa dinâmica, a diferença entre os contadores estará sempre aumentando e os contadores de outros símbolos não serão alterados, apenas a escala, que, nesta tabela, registra o total de símbolos já codificados. Note que o importante é a diferença entre os contadores, não seus valores absolutos. Apenas para a escala, seu valor absoluto é importante. O algoritmo de codificação será modificado agora, para incluir a variável escala. Portanto:

- *intervalo = limite* superior – *limite inferior + 1*
- *limite superior* = limite *inferior + {(intervalo x contador alto do símbolo / escala) – 1}*
- *limite inferior* = limite *inferior + {intervalo x contador baixo do símbolo / escala}*

A tabela 2 tem comprimento igual à tabela 1, mas, neste caso, os contadores são iniciados do valor máximo mais um, decrescendo até 0, onde o valor máximo está na primeira posição e 0 na última posição da

tabela. Na codificação de um novo símbolo, primeiro emite-se um código de escape, servindo para identificar no fluxo de saída o novo símbolo (para decodificação). Para o código de escape, os contadores alto e baixo são estabelecidos em 1 e 0, respectivamente, e são passados para a rotina do código aritmético. Após a codificação do código de escape, os contadores alto e baixo e a escala são identificados na tabela 2 e passados para a rotina de codificação. Do mesmo modo que na tabela 1, o valor do símbolo é o endereço para os contadores, e a escala é a primeira posição da tabela. Neste caso, o valor dos contadores será tal que a diferença entre eles será sempre 1. Na seqüência, a tabela 2 é atualizada, consistindo, neste caso, no decremento no valor de todas as posições até o endereço do contador alto, de modo que reflita corretamente os valores necessários na predição do símbolo.

Exemplo 3.2: Atualização das tabelas de codificação aritmética adaptativa.
Seqüência: [3, 2, 2, 4, 3, 1, 4, 3, 3, 4, 4, 3, 3, 4, 4, 4]

Tabela 1:

- *contadores*: iniciados em 1, com comprimento igual ao valor máximo (4) da seqüência, mais 2. Assim, tabela1 = [1, 1, 1, 1, 1, 1].
- *atualização*: incremento até o endereço do contador alto do símbolo sendo codificado. O valor do símbolo é o endereço para seus contadores. Assim, por exemplo, sendo "2" o símbolo sendo codificado, seus contadores serão (onde zero corresponde à primeira posição):
  - contador alto = tabela1[2]
  - contador baixo = tabela1[3].

Tabela 2:

- *contadores*: iniciados do valor máximo da seqüência mais um, decrescendo até 0. Assim, tabela2 = [5, 4, 3, 2, 1, 0]
- *atualização*: decremento até o endereço do contador alto do símbolo sendo codificado.

Comentários (consultar tabela 3.7)

* Início da codificação. Inicialização das tabelas 1 e 2. A escala é sempre a primeira posição da tabela em questão.

(2) Quando um símbolo diferente dos anteriores está sendo codificado, verifica-se que seus contadores são iguais. Sua codificação é momentaneamente abortada, e é codificado um código de escape.

(3) Para codificar o código de escape, seus contadores alto e baixo são fixados em 1 e 0, respectivamente e o valor da escala é extraído da tabela 1.

(4) Após a codificação do código de escape, deve-se retornar à codificação do símbolo. Novos símbolos são codificados a partir da tabela 2. Extrai-se, então, dessa tabela, os valores dos contadores e da escala. Esses valores são passados para a rotina de codificação e as tabelas são atualizadas.

(5) Neste caso, o símbolo já foi anteriormente codificado. Extrai-se, portanto, os valores dos seus contadores e escala a partir da tabela 1; os valores são passados para a rotina de codificação e as tabelas são atualizadas.

Símbolo	Contador alto	Contador baixo	escala	Atualização tabela 1	Atualização tabela 2	Comentários
Início	1	1	1	[1, 1, 1, 1, 1, 1]	[5, 4, 3, 2, 1, 0]	*
3	1	1				(1)
escape	1	0	1			(2)
3	2	1	5	[**2, 2, 2, 2,** 1, 1]	[**4, 3, 2, 1,** 1, 0]	(3)
2	2	2				(1)
escape	1	0	2			(2)
2	2	1	4	[**3, 3, 3,** 2, 1, 1]	[**3, 2, 1,** 1, 1, 0]	(3)
2	3	2	3	[**4, 4, 4,** 2, 1, 1]		(4)
4	1	1				(1)
escape	1	0	4			(2)
4	1	0	3	[**5, 5, 5, 3, 2,** 1]	[**2, 1, 0, 0, 0,** 0]	(3)
3	3	2	5	[**6, 6, 6, 4,** 2, 1]		(4)
1	6	6				(1)
escape	1	0	6			(2)
1	1	0	2	[**7, 7,** 6, 4, 2, 1]	[**1, 0,** 0, 0, 0, 0]	(3)
4	2	1	7	[**8, 8, 7, 5, 3,** 1]		(4)
3	5	3	8	[**9, 9, 8, 6,** 3, 1]		(4)
3	6	3	9	[**10, 10, 9, 7,** 3, 1]		(4)
4	3	1	10	[**11, 11, 10, 8, 4,** 1]		(4)
4	4	1	11	[**12, 12, 11, 9, 5,** 1]		(4)
3	9	5	12	[**13, 13, 12, 10,** 5, 1]		(4)
3	10	5	13	[**14, 14, 13, 11,** 5, 1]		(4)
4	5	1	14	[**15, 15, 14, 12, 6,** 1]		(4)
4	6	1	15	[**16, 16, 15, 13, 7,** 1]		(4)
4	7	1	16	[**17, 17, 16, 14, 8,** 1]		(4)

Tabela 3.7: *Atualização das tabelas para codificação adaptativa.*

3.4.3.4 Fluxograma

A codificação aritmética demanda um modelamento estatístico para a codificação dos símbolos. Portanto, o fluxograma desta seção descreve em detalhes o modelamento apresentado no exemplo 3.2. As tabelas apresentadas no exemplo são identificadas como *counter_table* e *index_table*, respectivamente para as tabelas 1 e 2. Quando os valores da *escala* e dos *contadores* de símbolo são identificados, a rotina do código aritmético é chamada para a codificação do símbolo.

A figura 3.17 apresenta o fluxograma do codificador aritmético, enquanto a figura 3.18 apresenta o fluxograma do decodificador.

3.5 CODIFICAÇÃO RECURSIVA

Ao invés de subdividir a imagem em blocos menores, como no esquema de compressão JPEG [34], o esquema aqui desenvolvido consiste em dividir uma longa seqüência – neste caso, a imagem será tratada como uma seqüência unidimensional na codificação – em seqüências menores e codificar recursivamente cada subseqüência resultante.

Foram pensados dois modelos de dividir a seqüência: um modelo dividindo a seqüência pelo valor médio dos símbolos, outro modelo dividindo pelo valor mediano dos símbolos. Ambos apresentam desempenho similar.

A técnica de divisão recursiva segue o seguinte procedimento:

1. Dividir a seqüência de entrada em duas subseqüências. A divisão é realizada tendo como parâmetro para divisão o valor médio ou mediano dos símbolos. O símbolo anterior da seqüência decide em qual das subseqüências resultantes o símbolo seguinte deverá ser colocado. A seguir, testar se o processo de divisão é lucrativo.
2. Em caso positivo, o método chama a si mesmo duas vezes, executando o passo 1, com cada uma das duas subseqüências como argumento, ou, em caso negativo,
3. Executa a codificação de Huffman ou aritmética da seqüência de entrada.

Dividir a seqüência pelo símbolo anterior tenta utilizar a correlação entre símbolos adjacentes, fazendo com que o símbolo anterior decida em qual subseqüência o símbolo seguinte deve ser colocado. A primeira subseqüência contém os símbolos seguintes aos símbolos com valor menor ou igual a um valor limite (média ou mediana), e a segunda seqüência os símbolos remanescentes (incluindo o primeiro símbolo).

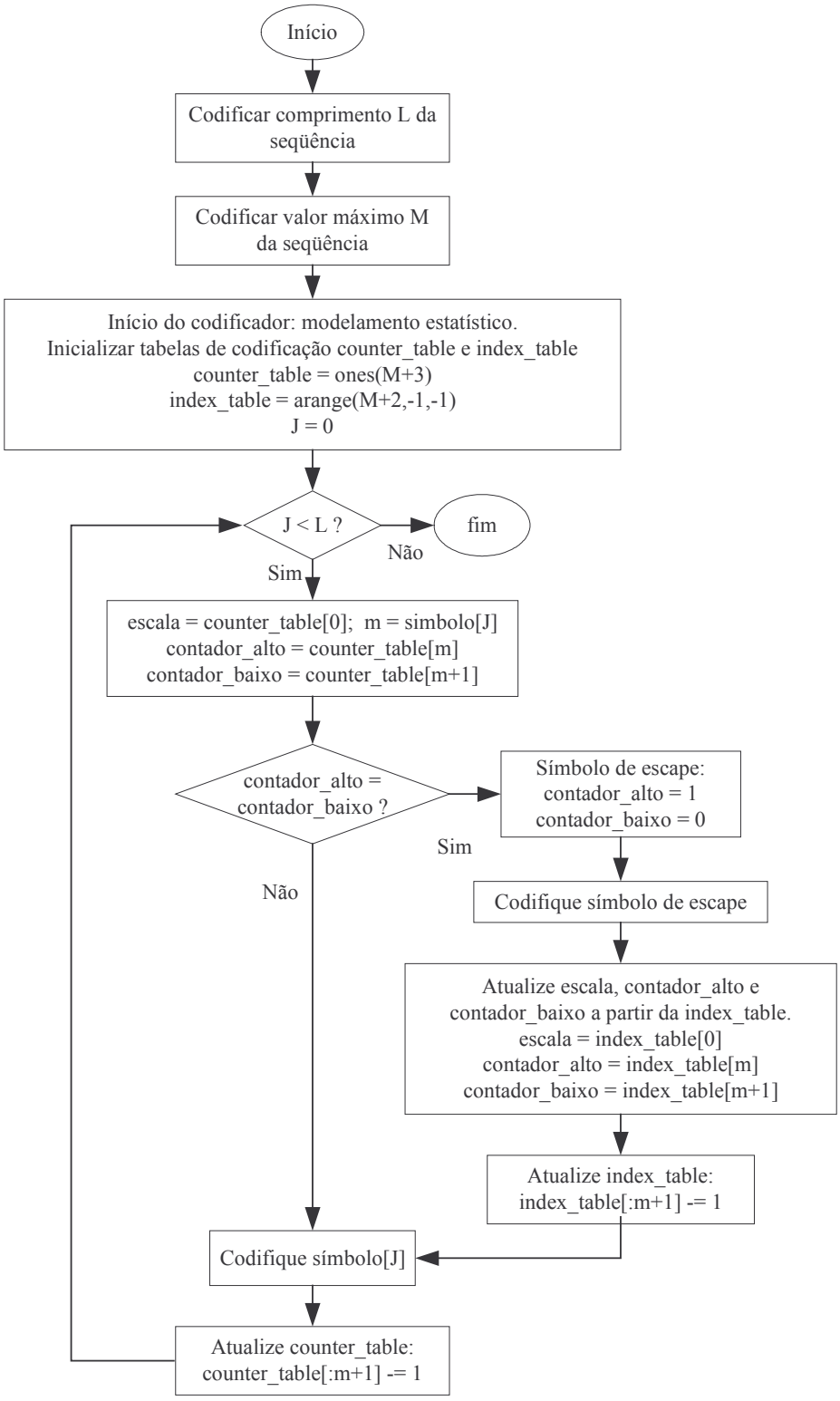


Fig. 3.17: *Fluxograma do codificador aritmético.*

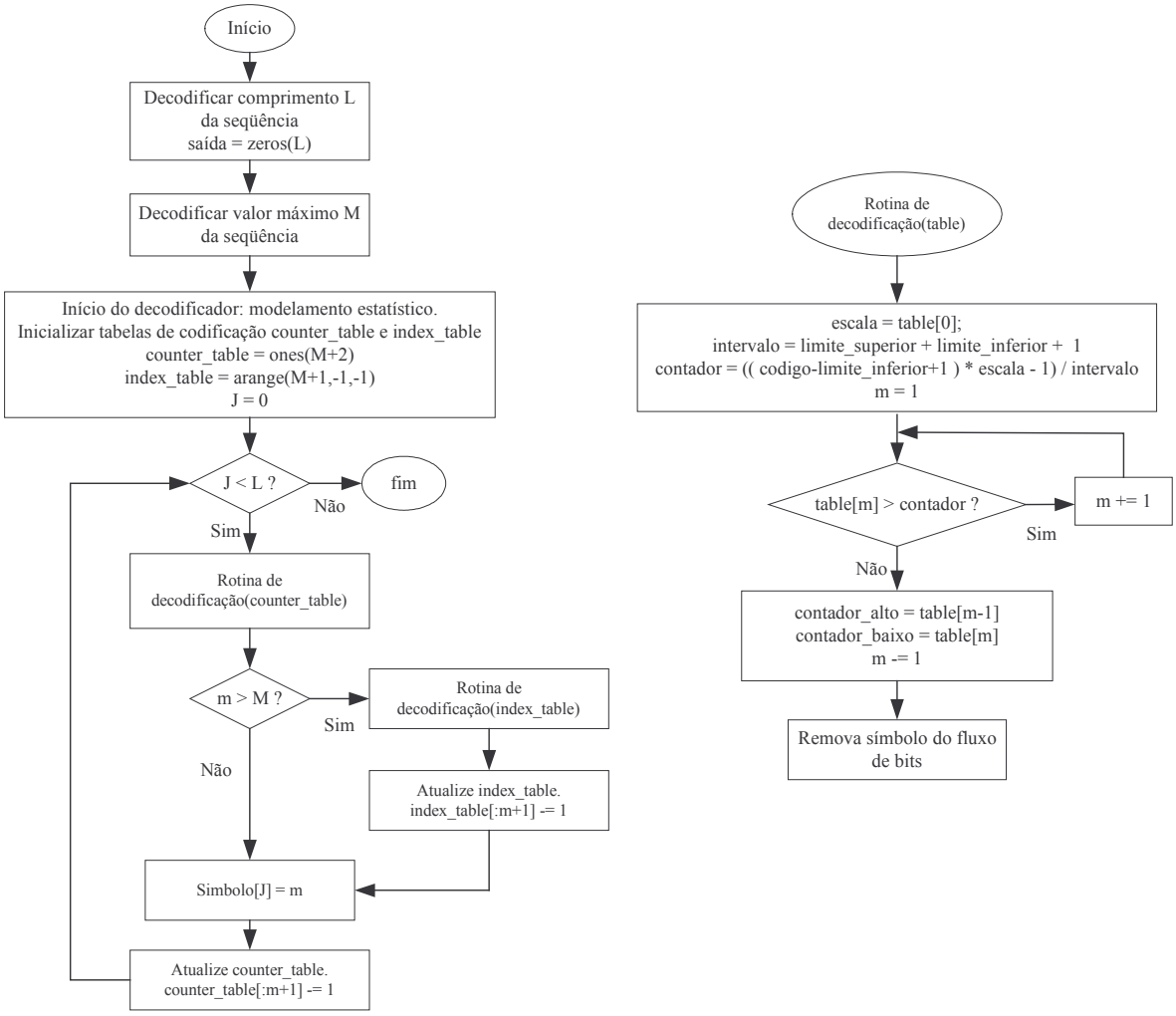


Fig. 3.18: *Fluxograma do decodificador aritmético.*

O exemplo 3.3 ilustra a divisão de uma seqüência com valor limite igual a 4 (mediana).

Exemplo 3.3:

Seqüência original: [10, 12, 0, 8, 4, 5, 0, 2, 3, 4, 8, 10, 0, 4, 3, 2, 12, 8]

Seqüência 1: [8, 5, 2, 3, 4, 8, 4, 3, 2, 12]

Seqüência 2: [10, 12, 0, 4, 0, 10, 0, 8]

Processo de divisão: O primeiro símbolo é colocado na seqüência 2. Para decidir em qual seqüência o símbolo seguinte será colocado, o símbolo anterior, 10, será comparado com a mediana, 4. Como 10 é maior que a mediana, a decisão é que o símbolo 12 seja colocado na seqüência 2. Para o terceiro símbolo, 0, o símbolo anterior, 12, será comparado com 4 e a decisão será por colocá-lo na seqüência 2. Para o quarto símbolo, 8, o

símbolo anterior, 0, será comparado com 4 e a decisão será por colocá-lo na seqüência 1, já que 0 é menor que a mediana. O processo se repete até que a seqüência seja completamente dividida.

Para cada um dos modelos de codificação executados no passo 3, uma regra de decisão para divisão da seqüência foi elaborada. Para o código de Huffman, a regra de decisão baseia-se na contagem do número de bits produzidos para codificar a seqüência com e sem divisão. Para o código aritmético, o número de bits produzidos durante a codificação não pode ser conhecido *a priori*, assim, a regra de decisão baseia-se numa fórmula que leva em consideração o valor da entropia das seqüências envolvidas na divisão. Maiores detalhes são fornecidos nas seções a seguir.

3.5.1 Código de Huffman recursivo

A regra de decisão para codificar uma seqüência recursivamente com o código de Huffman é a contagem do número de bits exigidos na codificação. Um dos primeiros passos, quando usando o código de Huffman, é a atribuição de palavras-código para o alfabeto. De posse do histograma dos coeficientes quantizados, o produto do número de ocorrências dos símbolos pelo comprimento das respectivas palavras-código, fornece o número de bits exigidos para codificar a seqüência.

Além desses bits, um número extra de bits precisa ser anexado aos dados comprimidos para codificar a tabela de Huffman, de modo que a decodificação seja bem sucedida. Essa informação lateral é eficientemente codificada de acordo com a tabela 4.3. Portanto, os bits exigidos para codificar a tabela de Huffman também são conhecidos a priori.

Portanto, quando o procedimento de divisão executa o passo 1, a regra de decisão testa, se com a divisão em subseqüências, ocorre redução no número de bits. Enquanto o teste indicar uma demanda menor de bits para codificar as subseqüências, o passo 2 é executado. Quando o teste indicar que o processo de divisão não mais compensa, o passo 3 é executado, codificando recursivamente todas as subseqüências produzidas no procedimento.

No modo como a divisão foi implementada, um mínimo de informação lateral é necessário para especificar quando uma seqüência foi dividida. Na verdade, apenas um bit é usado para dizer se a seqüência foi dividida ou codificada. Entretanto, para cada seqüência codificada, alguns bits são exigidos para especificar o tamanho da seqüência, além da exigência de incluir a respectiva tabela de Huffman como informação lateral.

3.5.2 Código aritmético recursivo

Para o código aritmético, o número de bits produzidos durante a codificação não pode ser conhecido *a priori*, assim, a regra de decisão baseia-se numa fórmula que leva em consideração o valor da entropia e do comprimento das seqüências envolvidas na divisão. A regra de decisão é a seguinte:

$$(L*entropia - L1*entropia1 - L2*entropia2) > 5*size\_histogram \tag{3.9}$$

onde *L* é o comprimento da seqüência de entrada, *entropia* é a entropia da seqüência de entrada, *L1* e *L2* são os comprimentos das seqüências divididas, *entropia1* e *entropia2*, são as entropias das seqüências divididas, e *size_histogram*, é o comprimento do histograma da seqüência original.

Portanto, quando o procedimento de divisão executa o passo 1, a regra de decisão testa se, com a divisão em subseqüências, a equação 3.9 é verdadeira. Enquanto o teste indicar verdadeiro, o passo 2 é executado. Quando o teste indicar que o processo de divisão não mais compensa, o passo 3 é executado, codificando recursivamente todas as subseqüências produzidas no procedimento.

Na codificação aritmética recursiva, a informação lateral anexada aos dados codificados é insignificante. Apenas um bit precisa ser codificado para dizer quando a seqüência foi dividida, e também mais alguns bits para codificar o comprimento de cada seqüência. O código aritmético não demanda qualquer outra informação lateral.

3.6 MEDIDAS DE DISTORÇÃO EM COMPRESSÃO DE IMAGENS

Para avaliar o desempenho dos sistemas de compressão implementados, são utilizadas duas medidas objetivas. A primeira é o erro quadrático médio, *MSE*, que é calculado fazendo-se a extensão bidimensional da equação 4.1, resultando na seguinte equação:

$$MSE = \frac{1}{MN} \sum_{x=1}^{M} \sum_{y=1}^{N} \left[I(x,y) - \hat{I}(x,y) \right]^2 , \tag{3.10}$$

onde $I(x,y)$ corresponde a imagem original e $\hat{I}(x,y)$ é a imagem reconstruída, e *M* e *N* são as dimensões das imagens original e reconstruída.

A segunda medida objetiva é a relação sinal ruído de pico, *PSNR – Peak Signal to Noise Ratio*, calculada pela equação seguinte:

$$PSNR(dB) = 20 . log_{10}\left(\frac{255}{\sqrt{MSE}}\right). \quad (3.11)$$

Enquanto estas duas medidas podem não ser exatamente adequadas, elas providenciam um guia da qualidade da imagem reconstruída. Em geral, uma boa imagem reconstruída é aquela com baixo MSE e alto PSNR, significando que temos uma imagem recuperada com alto grau de fidelidade.

3.7 RESUMO DO CAPÍTULO

Neste capítulo começamos discutindo técnicas de quantização escalar. Discutimos, na seqüência, o algoritmo de Shapiro, considerado um sofisticado sistema de codificação. O algoritmo de Shapiro foi elaborado especialmente para codificar os coeficientes da transformada wavelet de um sinal bidimensional. Após a apresentação dos modelos de quantização, foram discutidas técnicas de codificação por entropia. Os códigos discutidos foram o código de corrida, o código de Huffman e o código aritmético. O capítulo apresentou também o fluxograma dos métodos de codificação, e na seqüência, foi discutida a codificação recursiva da seqüência como técnica alternativa de codificação na busca por um melhor nível de compressão e finalizou apresentando as medidas usuais para avaliar o desempenho dos sistemas compressores de imagens.

CAPÍTULO 4 SIMULAÇÕES E RESULTADOS

4.1 INTRODUÇÃO

A principal razão para usar a transformada wavelet é a sua capacidade de análise em multiresolução, que permite analisar um sinal em várias escalas e resoluções. Com a ajuda dos programas de simulação, fomos capazes de capturar como as wavelets transformam uma imagem bidimensional. Neste capítulo, estaremos discutindo os resultados apresentados pela combinação de diferentes esquemas de quantização e codificação. Estaremos apresentando os resultados das simulações em gráficos de bits por pixel versus PSNR. Para que o sinal recuperado apresente um nível de distorção aceitável, a faixa de valores para a relação sinal ruído de pico deve estar entre 25 e 35 dB, sendo 25 dB o limite inferior, ou seja, abaixo desse valor, o sinal recuperado apresenta níveis de ruído inaceitáveis.

4.2 SIMULAÇÕES

Com o objetivo de alcançar melhores resultados em nível de compressão, testamos algumas técnicas. A primeira delas, consistiu em retirar a média do sinal antes de transformá-lo. Na reconstrução, após a transformada inversa, a média é novamente inserida. O padrão de compressão JPEG [34], baseado na transformada discreta de cosseno, usa essa combinação.

A segunda medida consiste na divisão recursiva da seqüência. Como discutido no capítulo 3, com esta técnica, dividimos a seqüência em alguns grupos, e codificamos cada um desses grupos separadamente. No agrupamento das seqüências, procuramos utilizar a correlação entre amostras adjacentes. Assim, como as seqüências resultantes apresentam alta correlação, o desempenho dos algoritmos de compressão é otimizado. Essa técnica é interessante, porque a seqüência só é dividida se a codificação das seqüências resultantes produzirem menos bits que a seqüência original.

Quando aplicamos a DWT até o primeiro nível de decomposição (figura 4.1), a imagem é dividida em quatro subbandas, LL, LH, HL e HH. A versão aproximada da imagem encontra-se na subbanda LL, enquanto os detalhes da imagem são encontrados nas demais subbandas. Isso sugere que a maioria dos coeficientes de magnitude significante encontra-se na região LL, enquanto os demais coeficientes serão, na maioria, insignificantes. A análise do histograma da DWT, figura 4.1(e), sugere esse resultado. O grande número de coeficientes na região centrada em zero, é resultante das subbandas que adicionam detalhes à imagem. Conforme o nível de decomposição aumenta, o intervalo de variação dos coeficientes wavelet também aumenta, assim como o número de coeficientes insignificantes centrados em torno de zero, como

mostrado na figura 4.1(f). As figuras 4.1(h) e 4.1(i), mostram os mesmos resultados, porém, agora aplicamos a DWT no sinal de média nula. Nota-se que o intervalo de variação dos coeficientes é modificado, porém, nem sempre essa medida apresentará bons resultados, conforme será verificado nos próximos tópicos deste capítulo, onde os resultados das simulações serão apresentados.

Fig 4.1a: *Imagem original*

Fig. 4.1b: *DWT 1-nível*

Fig. 4.1c: *DWT 2-níveis*

Fig. 4.1d: *Histograma imagem original*

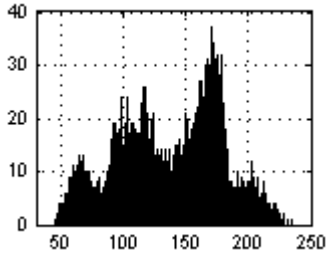


Fig. 4.1e: *Histograma do 1º. nível de decomposição*

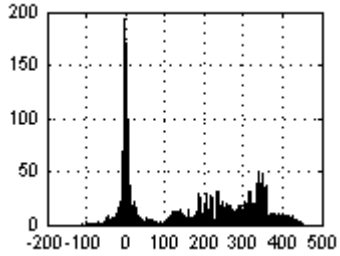


Fig. 4.1f: *Histograma do 2º. nível de decomposição*

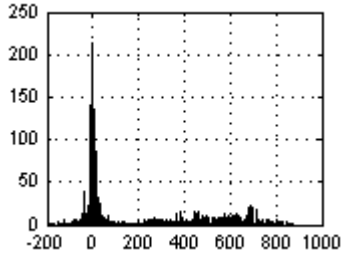


Fig. 4.1g: *Histograma imagem original*

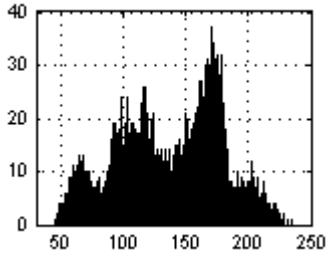


Fig. 4.1h: *Histograma do 1º. nível de decomposição (média nula)*

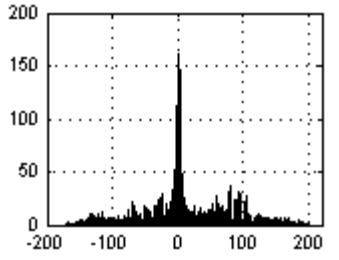
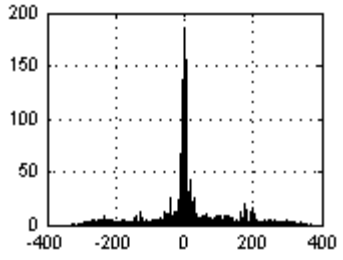


Fig. 4.1i: *Histograma do 2º. nível de decomposição (média nula)*



Optamos por aplicar a DWT até o segundo nível de decomposição, de modo a obter um bom retorno em PSNR e atingir bons níveis de compressão. Quanto mais alto o nível de decomposição, maior é o intervalo de variação dos coeficientes, portanto, mais refinada precisa ser a quantização para atingir o mesmo retorno em PSNR que o nível anterior, resultando, porém, numa taxa de compressão maior.

Investigaremos o desempenho das implementações em quatro diferentes imagens teste: Lena, Cameraman, Peppers e Barbara (figura 4.2).

Fig. 4.2a: Imagem original Lena 256x256

Fig. 4.2b: Imagem original Cameraman 256x256

Fig. 4.2c: Imagem original Peppers 256x256

Fig. 4.2d: Imagem original Barbara 256x256

As configurações do sistema utilizado nas simulações foram as seguintes:

Processador	AMD Athlon 1.2 GHz
Memória RAM	256 MBytes
Sistema operacional	Windows XP
Linguagem de programação	Python

4.3 SIMULAÇÕES IMAGEM LENA

Nesta seção estaremos apresentando e discutindo os resultados das simulações para diferentes combinações de quantização e codificação para a imagem Lena. Para cada modelo, foram realizadas simulações aplicando a DWT sobre a imagem original e aplicando a DWT sobre a imagem de média nula. Também foram realizadas simulações codificando a imagem diretamente (sem divisão recursiva), e aplicando a divisão recursiva (DR). Os resultados estão apresentados nas seções a seguir.

4.3.1 Algoritmo EZW

Esta seção apresenta o desempenho do algoritmo de codificação EZW. São apresentados dois gráficos, um para a simulação do código de Huffman e outro para a simulação do código aritmético. Quatro curvas existem em cada técnica de codificação, duas para o sinal codificado sem divisão recursiva, sendo uma delas para a codificação do sinal de média nula e outra para a codificação do sinal original; e de modo análogo, duas curvas para o sinal codificado utilizando a divisão recursiva.

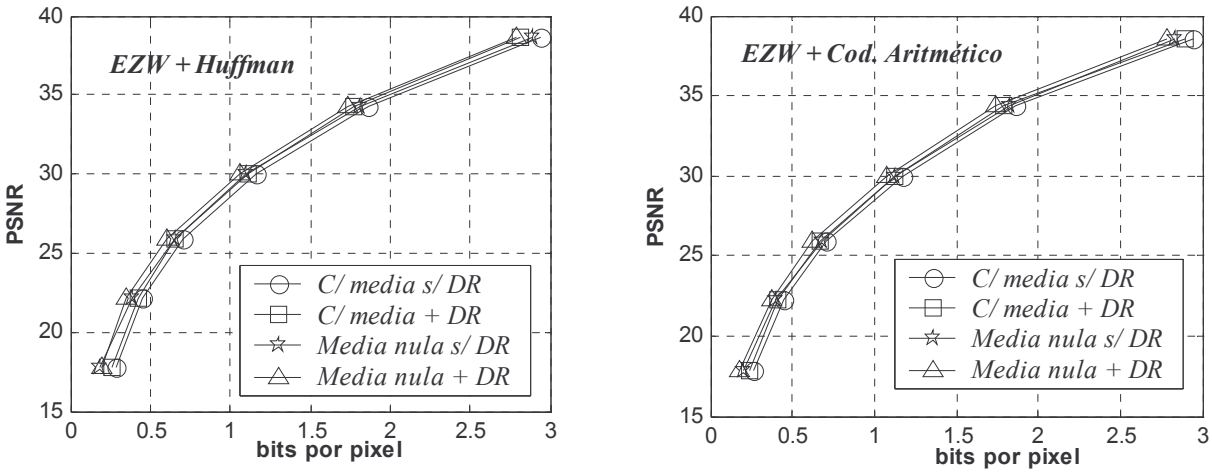


Fig. 4.3: *Algoritmo EZW – Lena.*

As simulações mostram que os codificadores têm desempenho muito similar. A combinação que apresentou desempenho ligeiramente superior é a combinação transformada do sinal de média nula e a codificação recursiva do sinal.

4.3.2 Quantização Uniforme

Aplicando quantização uniforme nos coeficientes transformados, o código aritmético é a técnica de codificação que apresenta o melhor desempenho. Para este codificador, os resultados apresentados pela codificação dos coeficientes da DWT da imagem de média nula são inferiores a aqueles apresentados pela codificação dos coeficientes da DWT da imagem original. A codificação recursiva tem desempenho superior

à codificação direta do sinal. O código de Huffman mostrou-se ineficiente neste modelo de quantização. Não se pode esquecer que o código de Huffman está restrito a taxas de 1 bit por pixel.

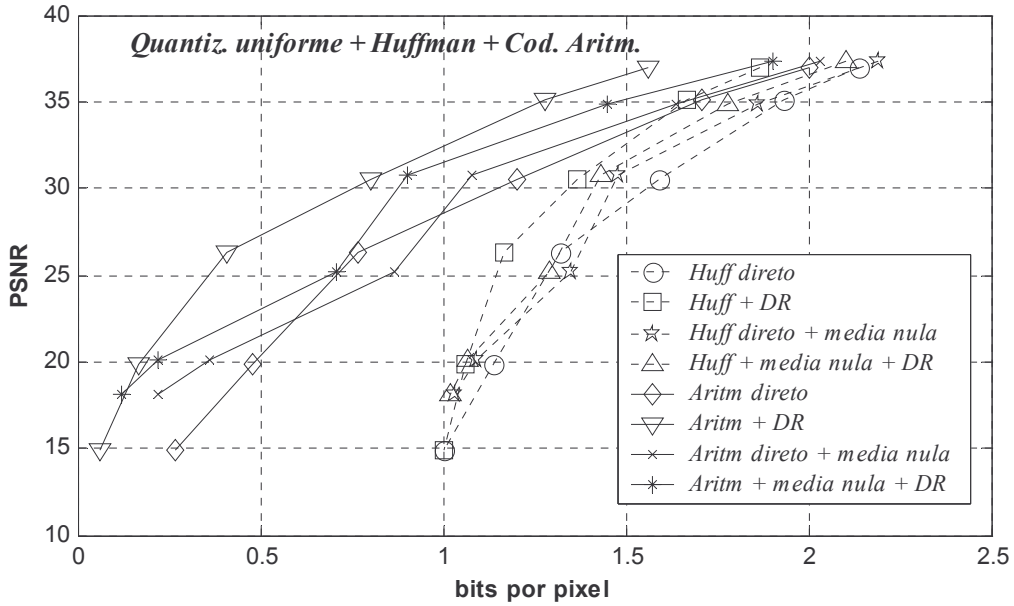


Fig. 4.4: *Quantização uniforme – Lena.*

4.3.3 Quantização uniforme com zona morta

Para a técnica de quantização uniforme com zona morta, as simulações são apresentadas em dois gráficos, um gráfico apresentando os resultados da codificação aritmética e outro do código de Huffman.

Para aumentar o desempenho do código de Huffman, simulamos o código de corrida como um passo anterior à codificação do sinal. Com essa medida, conseguimos reduzir as taxas de bits para valores abaixo de 1 bit por pixel. Os resultados mostram que a medida é adequada, pois essa combinação mostra que a combinação média nula, divisão recursiva e código de corrida como um passo anterior à codificação de Huffman, apresenta desempenho superior em relação as demais técnicas. A codificação do sinal original, com média, apresenta desempenho ligeiramente inferior nesta combinação. Os resultados estão apresentados na figura 4.5.

Na simulação do código aritmético, testamos se a execução do código de corrida como um passo anterior à codificação, implica no aumento dos níveis de compressão do sinal. Essa medida, porém, não apresentou o mesmo resultado que no código de Huffman. A justificativa é que o código aritmético já é potente por si mesmo. O melhor resultado, ou seja, aquele que apresenta o maior nível de compressão, é a codificação do sinal de média nula, combinando a técnica de divisão recursiva.

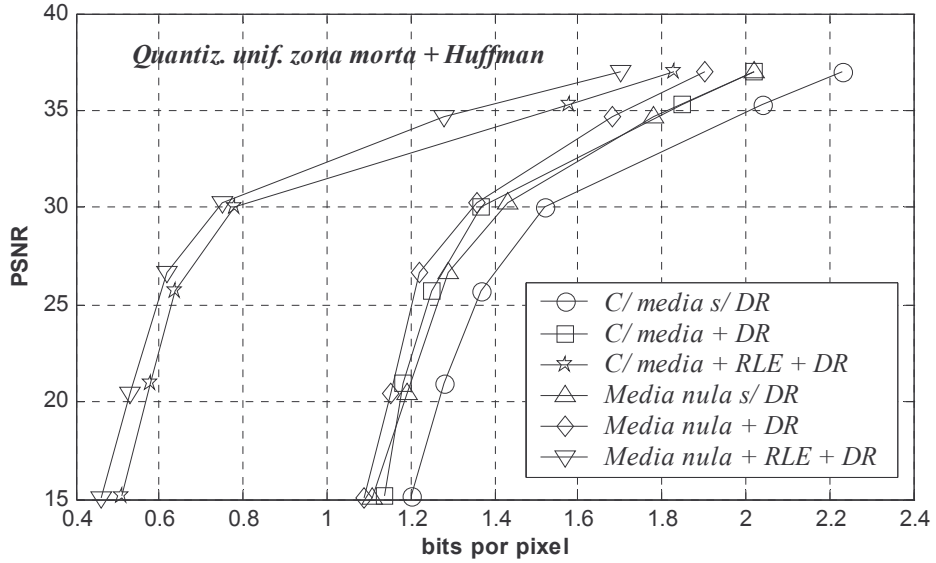


Fig. 4.5: *Quantização uniforme zona morta + código de Huffman – Lena.*

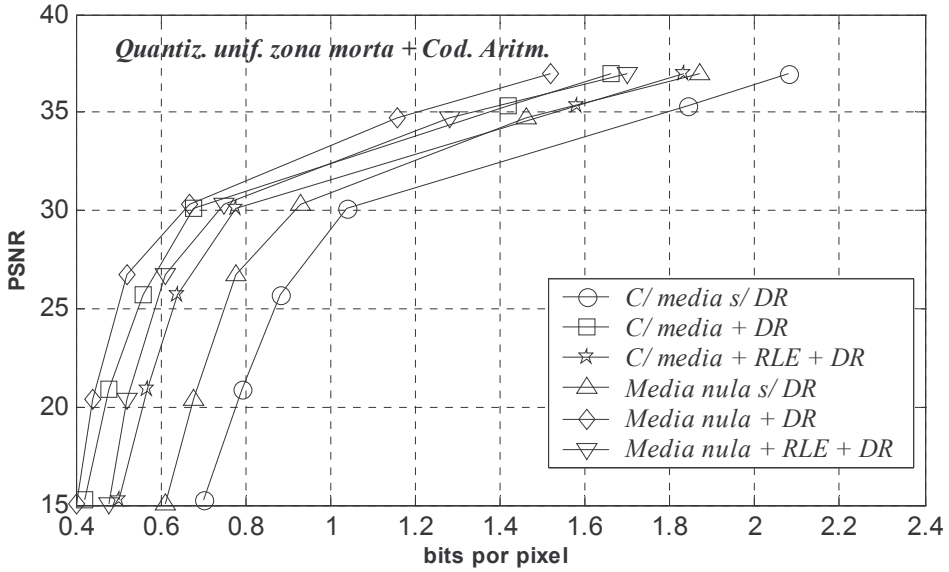


Fig. 4.6: *Quantização uniforme zona morta + código aritmético – Lena.*

4.3.4 Resultado final

Esta seção é dedicada a comparar todas as técnicas de quantização simuladas nas seções anteriores, com o intuito de verificar qual é o modelo que apresenta o melhor desempenho, ou seja, aquele que apresenta o maior nível de compressão do sinal. Com esta finalidade, os melhores resultados de cada técnica serão amostrados e confrontados. Quando mais de um gráfico foi utilizado para simular o modelo, um resultado por gráfico será amostrado.

Na maioria das técnicas testadas, o maior nível de compressão foi atingido quando a combinação sinal de média nula e codificação recursiva foi utilizada. A única exceção, foi na quantização uniforme dos coeficientes, quando a codificação do sinal original foi superior à codificação do sinal de média nula. Constatamos que os sinais que utilizam a técnica da divisão recursiva têm sempre um maior nível de compressão.

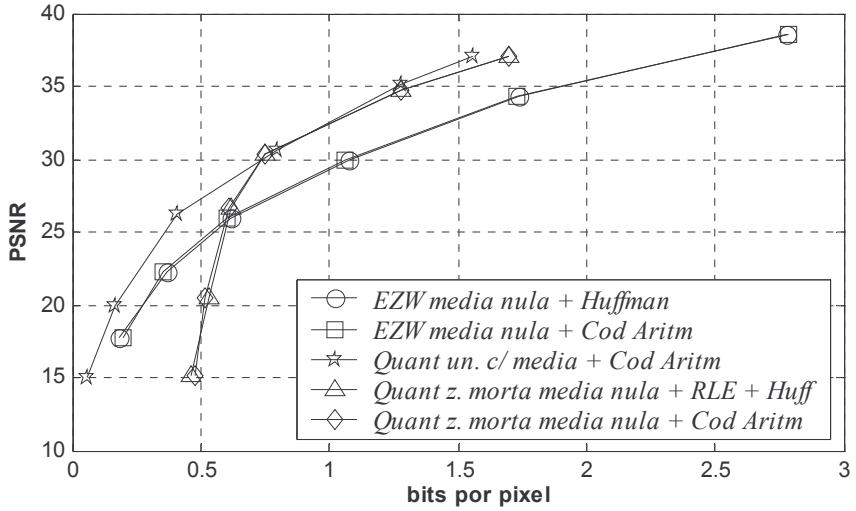


Fig. 4.7: *Resultado final – Lena.*

Conclusões: Estes resultados mostram que o desempenho do algoritmo EZW é inferior a qualquer combinação, já que resultados abaixo de 25 dB não são de interesse. Os demais codificadores têm desempenho similar, porém, a combinação transformada do sinal original, quantização uniforme e codificação aritmética recursiva apresenta desempenho ligeiramente superior. As figuras a seguir mostram o resultado do sinal recuperado quando o melhor codificador é utilizado.

Fig. 4.8a: *Imagem recuperada. 1,5 bits/pixel – 37 dB.*

Fig. 4.8b: *Imagem recuperada. 0,7 bits/pixel – 30 dB.*

4.4 SIMULAÇÕES IMAGEM BARBARA

Os resultados das simulações para a imagem Barbara serão apresentados e discutidos nesta seção. As diferentes técnicas serão agora utilizadas para codificar a imagem Barbara.

4.4.1 Algoritmo EZW

No algoritmo EZW, sempre dois gráficos são apresentados, um para simular o codificador aritmético e outro para simular o código de Huffman. Para ambos os codificadores, nenhuma técnica adicional, isto é, a codificação do sinal de média nula e a divisão recursiva não apresentam qualquer incremento no nível de compressão. Os resultados mostram que a combinação média nula e codificação recursiva é ligeiramente superior, porém os resultados são muito similares, e neste caso, qualquer combinação é satisfatória.

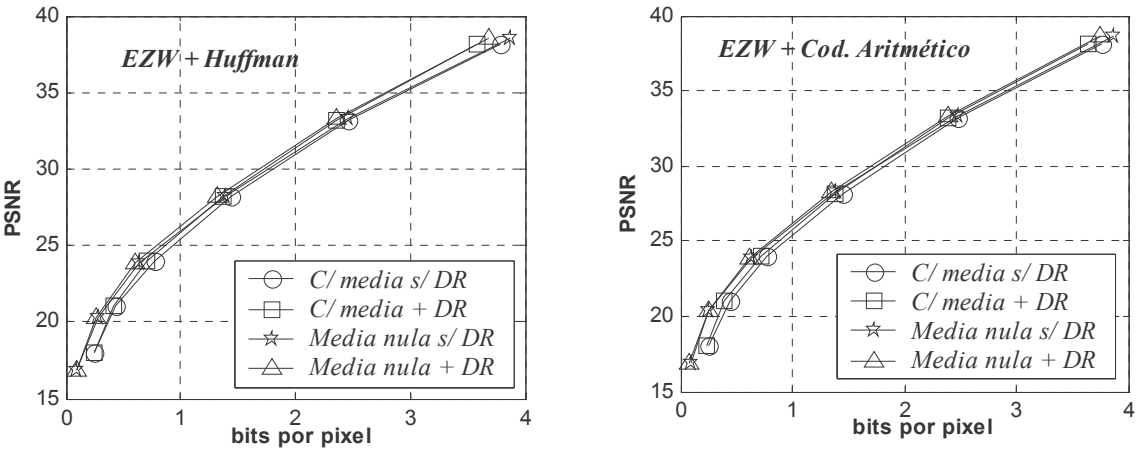


Fig. 4.9: *Algoritmo EZW – Barbara.*

4.4.2 Quantização Uniforme

Na quantização uniforme dos coeficientes, os resultados para os dois codificadores são apresentados num mesmo gráfico. Eles mostram que a utilização do código aritmético resulta num maior nível de compressão que o código de Huffman. Entre os resultados do código aritmético, aqueles apresentados pela codificação dos coeficientes da DWT da imagem de média nula são inferiores àqueles apresentados pela codificação dos coeficientes da DWT da imagem original. A utilização da divisão recursiva resulta num aumento efetivo do nível de compressão.

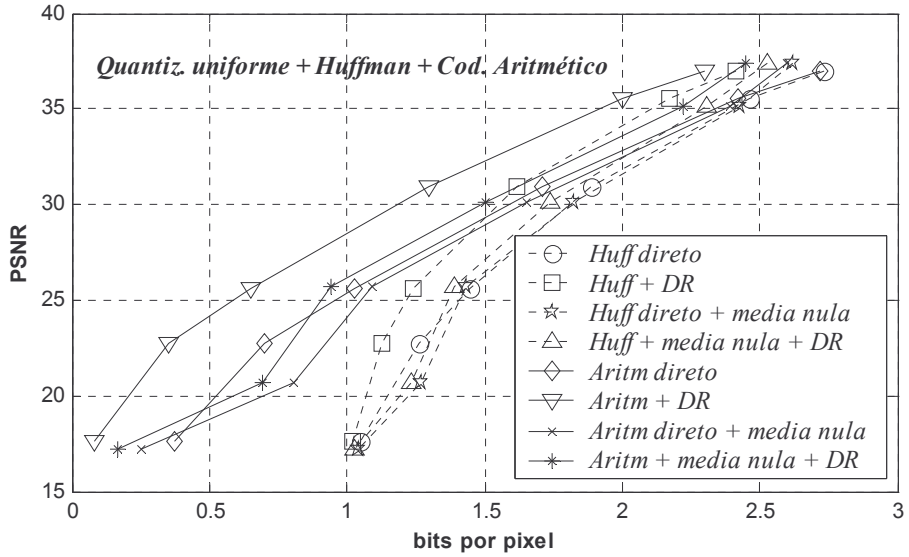


Fig. 4.10: *Quantização uniforme – Barbara.*

4.4.3 Quantização uniforme com zona morta

Na quantização uniforme com zona morta, os resultados do codificador de Huffman mostram que a combinação média nula, divisão recursiva e código de corrida como um passo anterior ao código de Huffman, é a que resulta no maior nível de compressão.

Para o codificador aritmético, a combinação divisão recursiva e média nula é a que apresenta o melhor desempenho.

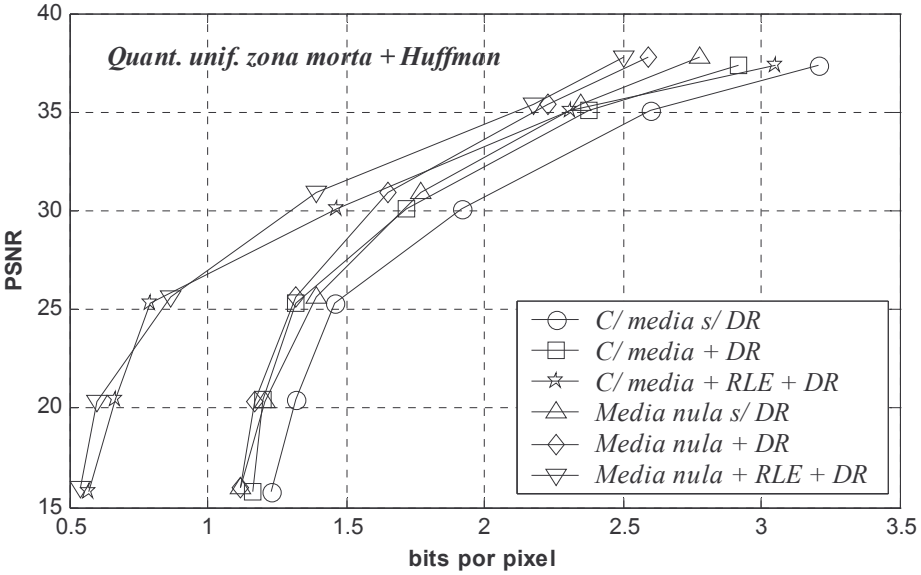


Fig. 4.11: *Quantização uniforme zona morta + código de Huffman – Barbara.*

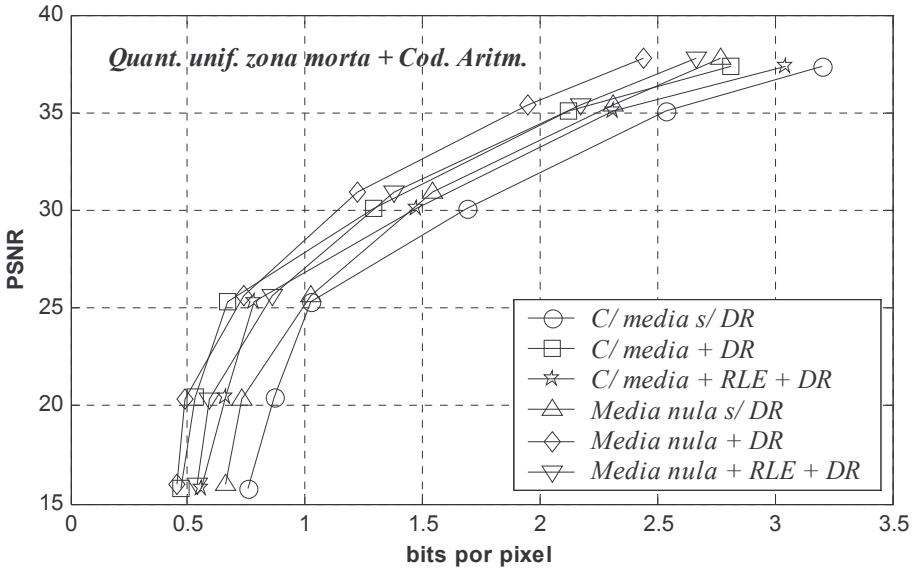


Fig. 4.12: *Quantização uniforme zona morta + código aritmético – Barbara.*

4.4.4 Resultado Final

Esta seção é dedicada a comparar os resultados das diferentes combinações de técnicas de quantização e codificação. O melhor resultado em cada modelo, ou seja, aquele que apresenta a maior compressão do sinal, é extraído e comparado com as demais técnicas, na busca pela melhor técnica de codificação.

Em todos os modelos, o melhor desempenho foi naquele onde a divisão recursiva foi utilizada como técnica final de codificação. Exceto para a quantização uniforme, a codificação do sinal de média nula foi sempre superior à codificação do sinal original. Portanto, estaremos amostrando esses resultados das simulações.

Para a quantização uniforme o sistema escolhido é a combinação divisão recursiva, sinal com nível DC e codificação aritmética. Para o restante, o modelo escolhido é aquele com divisão recursiva e média nula. Para a combinação quantização zona morta e código de Huffman, o modelo escolhido é aquele que tem o código de corrida como um passo anterior na codificação.

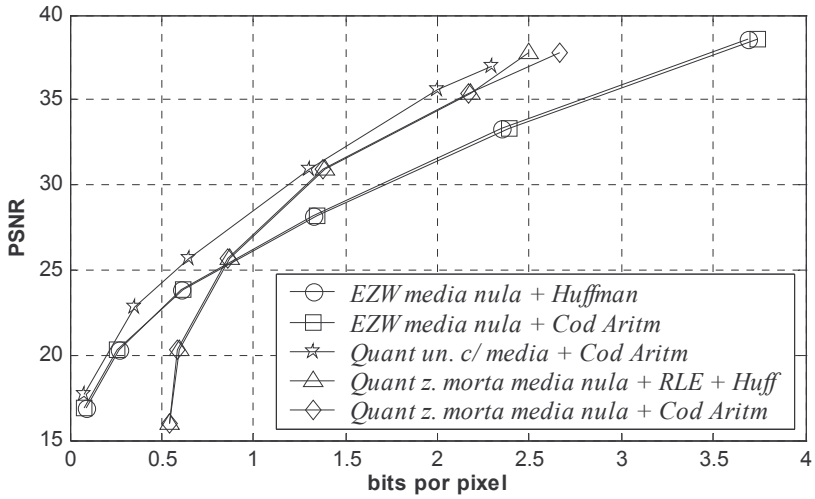


Fig. 4.13: *Resultado final – Barbara.*

Conclusões: O algoritmo EZW é a técnica que tem desempenho inferior. Para os demais modelos, aqueles com codificação aritmética recursiva apresentam desempenho superior, tanto para codificação do sinal com média nula quanto para codificação do sinal original. Como a imagem Barbara é difícil de ser codificada, devido ao grande número de componentes de alta freqüência, aqueles codificadores de melhor desempenho apresentam PSNR de 28 dB para uma taxa de 1 bit por pixel, em contraste com 0,7 bits por pixel e 30 dB para a imagem Lena. Os resultados das imagens recuperadas são apresentados na figura 4.14 para taxas iguais a 1,5 bits por pixel e 1 bit por pixel.

Fig. 4.14a: *Imagem recuperada 1,5 bits/pixel – 33 dB.*

Fig. 4.14b: *Imagem recuperada 1 bit/pixel – 28 dB.*

4.5 SIMULAÇÕES IMAGEM CAMERAMAN

Nesta seção estaremos discutindo a performance de diferentes combinações de quantização e codificação para a imagem Cameraman.

4.5.1 Algoritmo EZW

Os resultados da codificação utilizando o algoritmo EZW são apresentados em dois gráficos. Ambos os codificadores, tem desempenho similar, em qualquer tipo de combinação. As simulações mostram que a combinação média nula e codificação recursiva é ligeiramente superior, porém os resultados são equivalentes, ou seja, mesma taxa de bits por pixel.

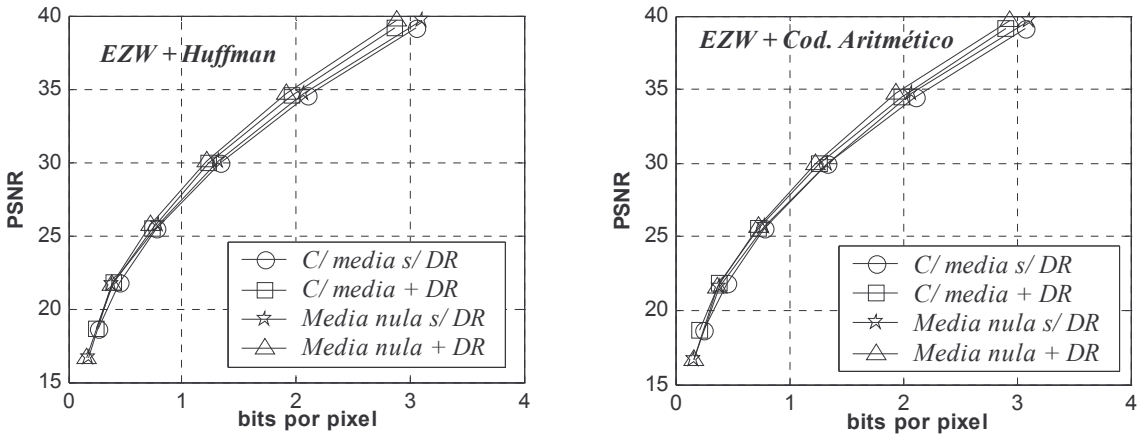


Fig. 4.15: *Algoritmo EZW – Cameraman.*

4.5.2 Quantização Uniforme

Neste modelo de quantização, os resultados para os dois codificadores estão apresentados num mesmo gráfico, o que simplifica a análise.

O código de Huffman tem desempenho muito inferior ao código aritmético, isto é, apresenta menor compressão do sinal, o que resulta em taxas de bits por pixel mais elevadas para um mesmo retorno na relação sinal ruído de pico, principalmente porque o maior nível de compressão conseguido por este codificador é 1 bit por pixel. Isto nos leva a inserir o código de corrida, na quantização por zona morta, para superar esta limitação. O código aritmético não apresenta esta restrição. Portanto, tem o melhor desempenho, em qualquer combinação. Para este codificador, os resultados da codificação dos coeficientes da DWT da imagem de média nula são inferiores aqueles apresentados pela codificação dos coeficientes da DWT da imagem original. A divisão recursiva implica em ganhos que justificam sua aplicação para ambos os codificadores.

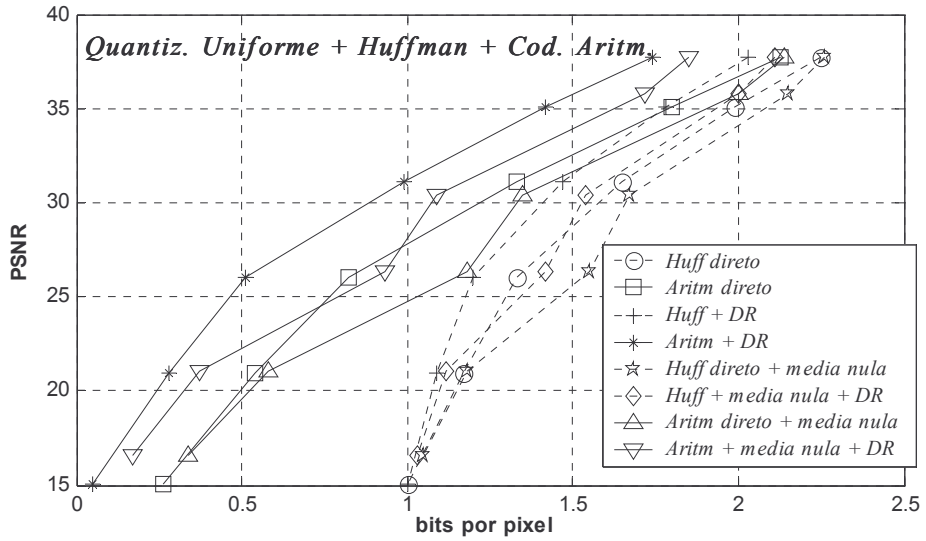


Fig. 4.16: *Quantização uniforme – Cameraman.*

4.5.3 Quantização uniforme com zona morta

Os resultados deste quantizador estão separados em dois gráficos, um para cada codificador. Os resultados do código de Huffman mostram que a combinação média nula, divisão recursiva e código de corrida como um passo anterior ao codificador, é a que resulta na maior compressão do sinal, enquanto a codificação do sinal original, ou seja, com média, apresenta desempenho inferior nesta combinação. Para o código aritmético, a combinação divisão recursiva e média nula é a que apresenta o melhor retorno em taxas de bits por pixel.

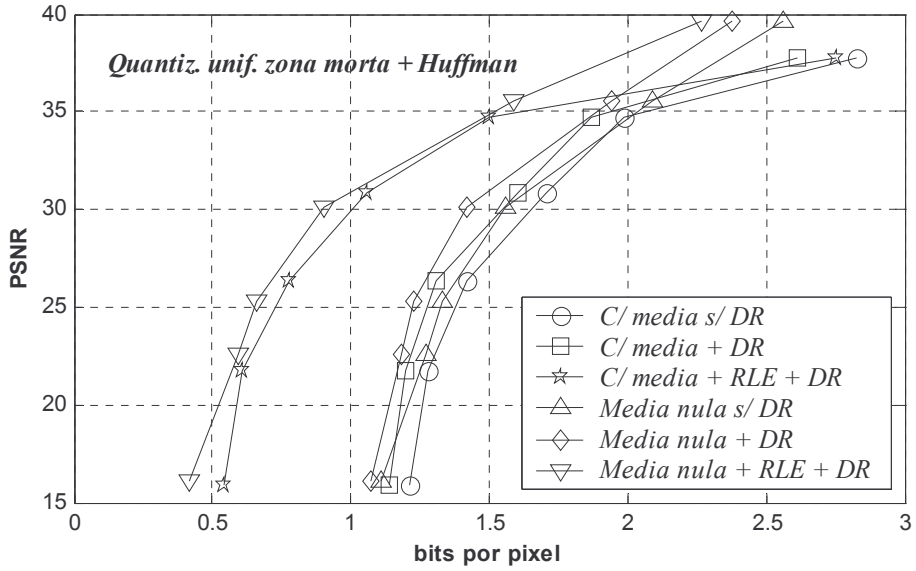


Fig. 4.17: *Quantização uniforme zona morta + código de Huffman – Cameraman.*

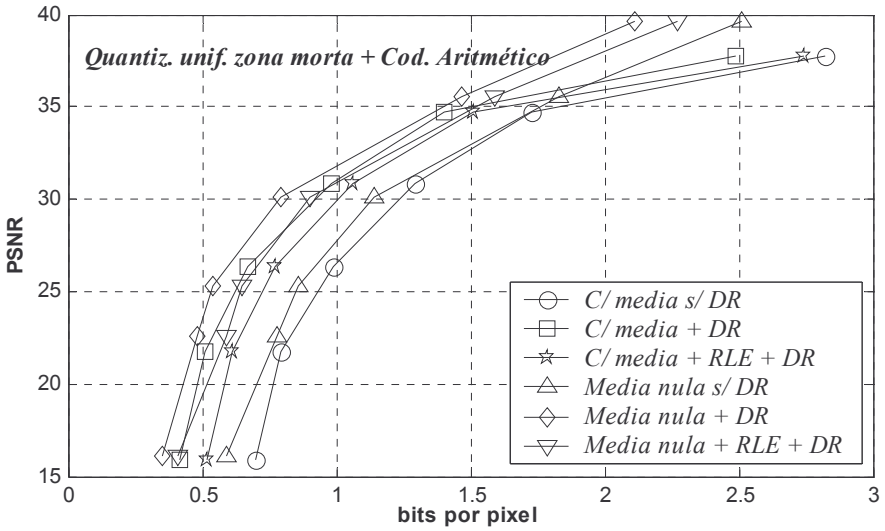


Fig. 4.18: *Quantização uniforme zona morta + código aritmético – Cameraman.*

4.5.4 Resultado Final

Estaremos comparando agora os codificadores que apresentam o maior nível de compressão para as diferentes técnicas de quantização. O resultado amostrado para a quantização uniforme é a combinação divisão recursiva, sinal com nível DC e codificação aritmética. Para as demais técnicas de quantização, o modelo amostrado é aquele com divisão recursiva e média nula. Para a combinação quantização zona morta e código de Huffman, o modelo que apresenta o melhor desempenho é aquele que tem o código de corrida como um passo anterior à codificação.

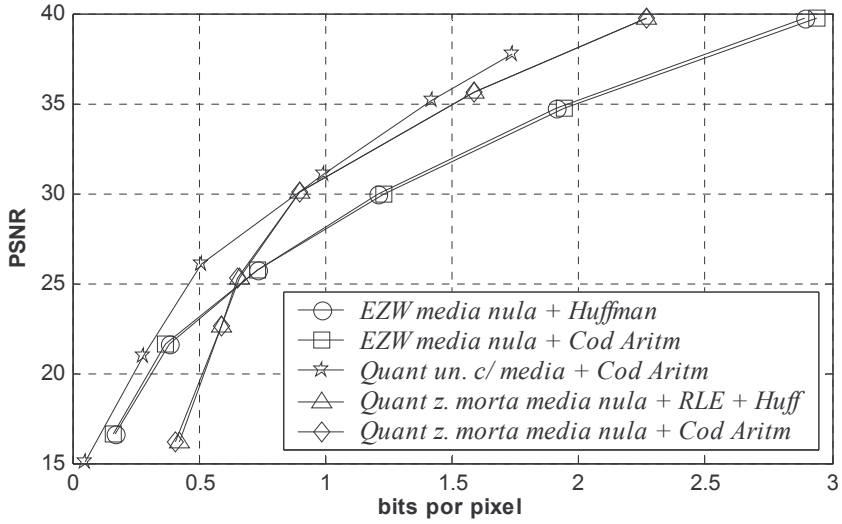


Fig. 4.19: *Resultado final – Cameraman.*

Conclusões: esse gráfico mostra que o modelo que apresenta o melhor desempenho é a combinação quantização uniforme, transformada do sinal original e codificação aritmética recursiva. Na quantização zona morta, os diferentes codificadores apresentam níveis de compressão similares para a mesma relação sinal ruído de pico. O desempenho do melhor codificador apresenta PSNR de 30 dB para uma taxa em torno de 0.8 bits por pixel. Os resultados das imagens recuperadas estão apresentados na figura 4.20.

Fig. 4.20a: *Imagem recuperada 1,5 bits/pixel – 36 dB.*

Fig. 4.20b: *Imagem recuperada 0.8 bits/pixel – 30 dB.*

4.5 SIMULAÇÕES IMAGEM PEPPERS

Nesta seção estaremos discutindo as resultados dos algoritmos nas diferentes combinações de quantização e codificação para a imagem Peppers.

4.5.1 Algoritmo EZW

As simulações do algoritmo EZW estão separadas em dois gráficos, para simplicidade de entendimento e análise.

Os resultados do codificador aritmético e do código de Huffman são muito similares. A combinação média nula e codificação recursiva é ligeiramente superior para os dois codificadores, resultando, portanto, num maior nível de compressão do sinal.

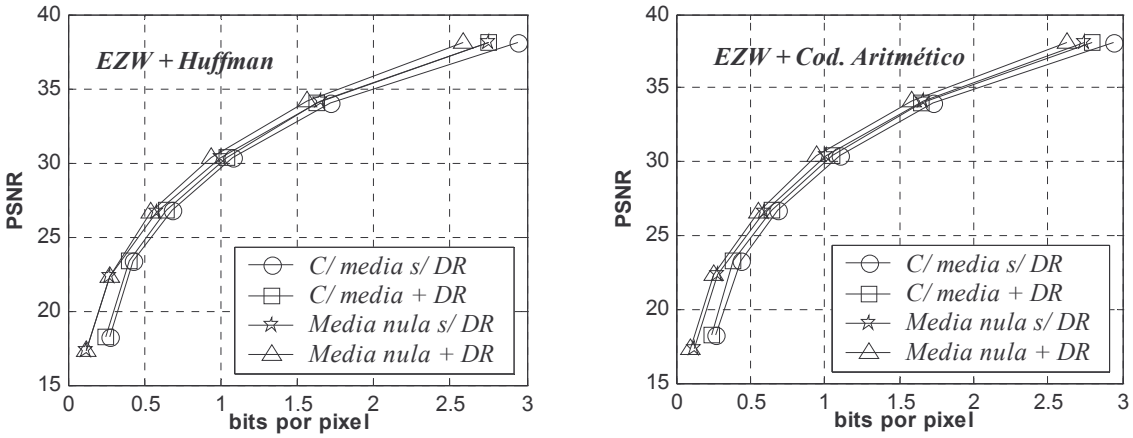


Fig. 4.21: *Algoritmo EZW – Peppers.*

4.5.2 Quantização Uniforme

Na quantização uniforme dos coeficientes, os resultados mostram que o código aritmético tem desempenho superior ao código de Huffman. Assim como discutido nas imagens anteriores, o código de Huffman está restrito a taxas máximas de 1 bit por pixel. No código aritmético, o resultado da codificação dos coeficientes da DWT da imagem original apresenta o maior nível de compressão entre todas as combinações.

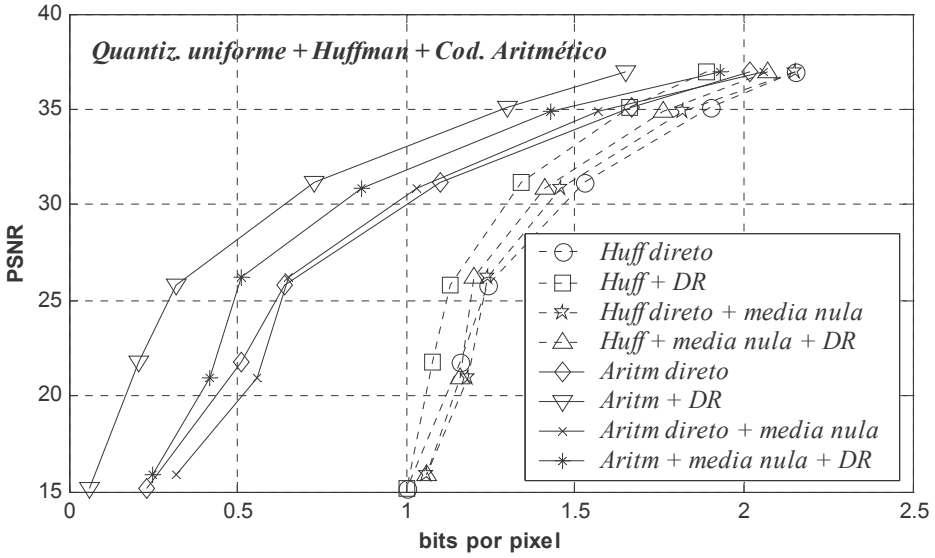


Fig. 4.22: *Quantização uniforme – Peppers.*

4.5.3 Quantização uniforme com zona morta

Na quantização uniforme com zona morta, para melhorar a compressão apresentada pelo código de Huffman, o código de corrida é utilizado como um passo anterior à codificação, o que resulta em taxas de bits inferiores a 1 bit por pixel.

Para o código de Huffman, os resultados deste quantizador mostram que a combinação média nula, divisão recursiva e código de corrida como um passo anterior ao código de Huffman, é a que apresenta o melhor resultado, ou seja, maior nível de compressão. Para o código aritmético, a combinação divisão recursiva e média nula tem desempenho superior.

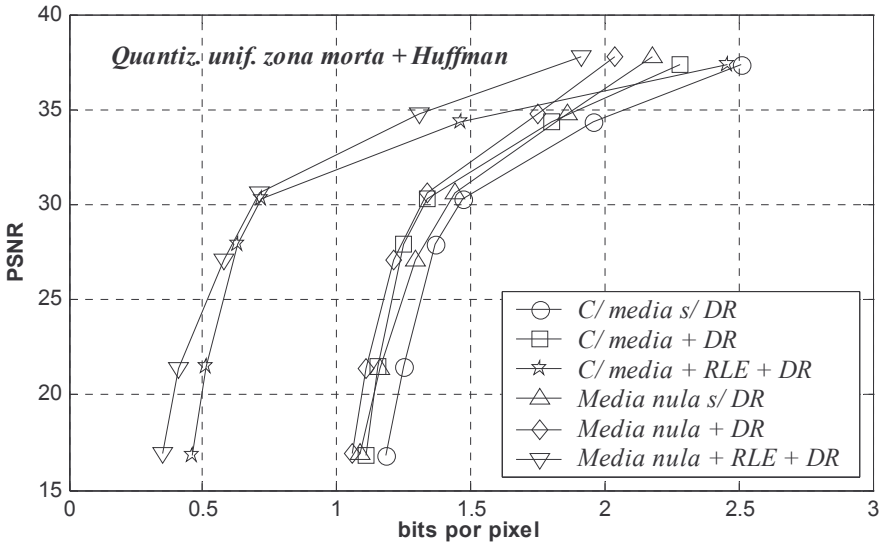


Fig. 4.23: *Quantização uniforme zona morta + código de Huffman – Peppers.*

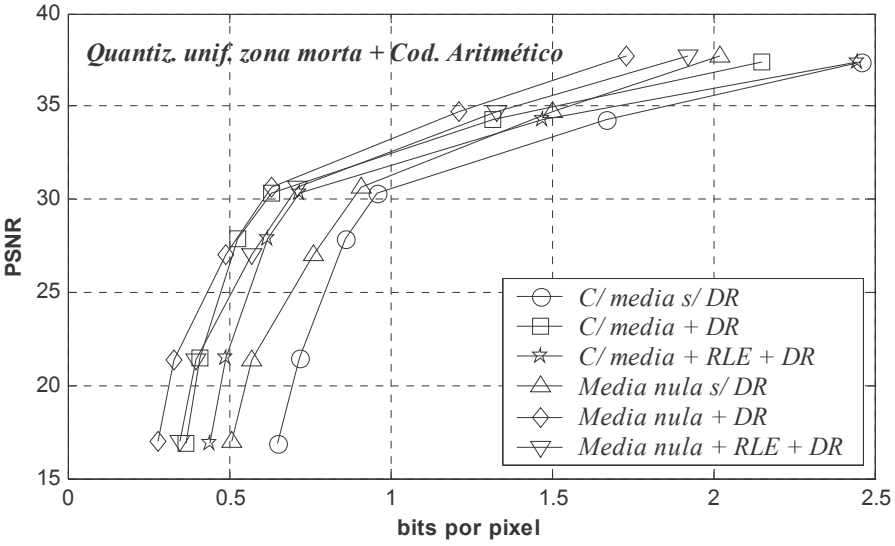


Fig. 4.24: *Quantização uniforme zona morta + código aritmético – Peppers.*

4.5.4 Resultado Final

Nesta seção estaremos colhendo os melhores resultados de cada técnica e confrontando para verificar qual modelo apresenta melhor desempenho. Para a quantização uniforme o sistema escolhido é a combinação divisão recursiva, sinal com nível DC e codificação aritmética. Para as demais combinações de quantização e codificação, o modelo escolhido é aquele com divisão recursiva e média nula. Para a combinação quantização zona morta e código de Huffman, o modelo que apresenta o melhor desempenho é aquele que tem o código de corrida como um passo anterior à codificação.

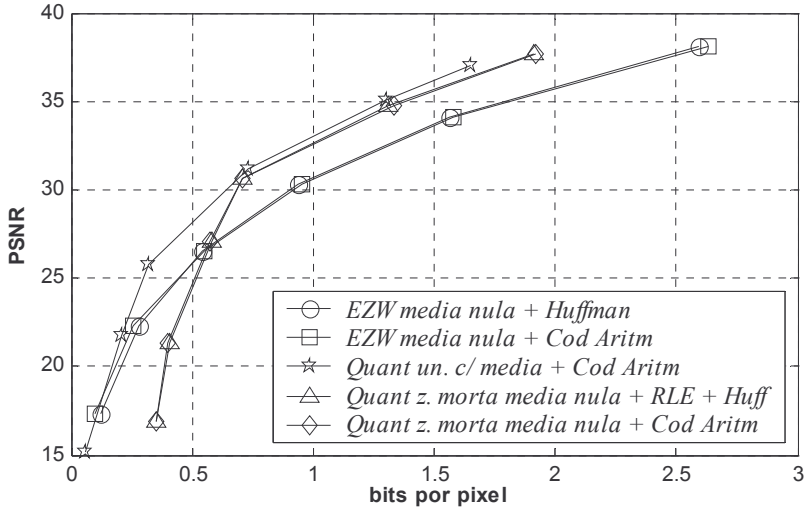


Fig. 4.25: *Resultado final – Peppers.*

Conclusões: esse gráfico mostra que o modelo que apresenta o melhor desempenho é novamente a combinação quantização uniforme, sinal original e codificação aritmética com divisão recursiva. O algoritmo EZW tem desempenho inferior. Na quantização zona morta, os codificadores se equivalem, porém, com desempenho ligeiramente inferior para PSNR acima de 30 dB. O desempenho do melhor codificador apresenta PSNR de 30 dB para uma taxa de 0.7 bits por pixel.

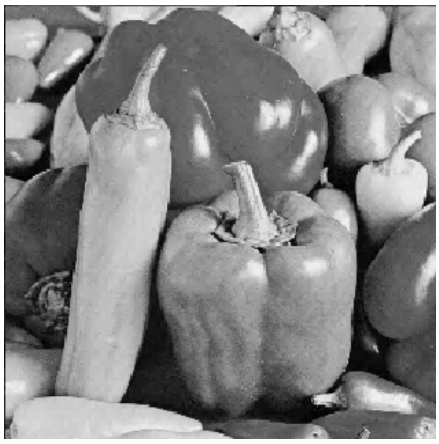


Fig. 4.26a: *Imagem recuperada 1,5 bits/pixel – 36,5 dB.*

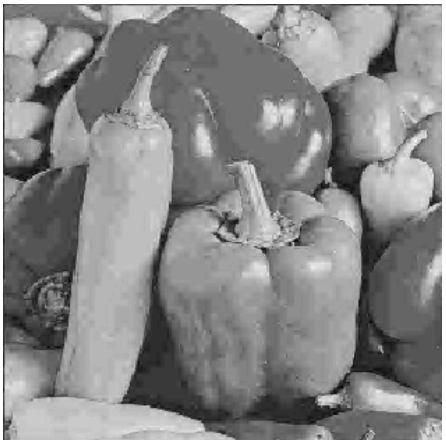


Fig. 4.26b: *Imagem recuperada 0,7 bits/pixel – 30 dB.*

5.6 RESUMO DO CAPÍTULO

Neste capítulo apresentamos os resultados das simulações dos diferentes algoritmos necessários para codificar uma imagem. Os algoritmos foram implementados na linguagem Python, para utilizar a facilidade de programação da linguagem e o pacote de processamento de imagens Numerical Python. Foram implementados os três passos do modelo compressor de imagens proposto: a transformada linear, a quantização e a codificação por entropia. As simulações mostraram que a codificação recursiva é uma técnica que apresenta desempenho superior em comparação com a codificação direta do sinal. Os resultados também mostram que o modelo que apresenta o melhor desempenho em taxa de bits é aquele que combina a transformada wavelet do sinal original, quantização uniforme e codificação aritmética recursiva. As técnicas de transformada sobre o sinal de média nula e quantização uniforme com zona morta, combinadas com o código aritmético recursivo, ou com o código de corrida antecedendo o código de Huffman recursivo, também apresentam bom retorno em taxa de bits. As simulações também mostraram que quando a imagem tem muitos componentes de alta freqüência, os desempenho das técnicas é inferior, ou seja, o resultado será um menor nível de compressão do sinal, o que é esperado.

CAPÍTULO 5 CONCLUSÃO

O objetivo deste trabalho foi estudar e implementar um modelo compressor com perdas de imagens bidimensionais na linguagem *Python*, disponibilizando um software contendo um conjunto de ferramentas para compressão do sinal numa linguagem de código aberto. Existem várias linguagens de programação que dispõem seus códigos para livre uso, apesar de pertencerem a companhias privadas, como a linguagem *Java*. Optamos por trabalhar com a linguagem *Python*, que não pertence a nenhum grupo em particular e oferece bons pacotes para processamento de sinais bidimensionais. Verificamos se os modelos de compressão simulados apresentaram bom desempenho na codificação dos sinais.

Implementamos um software que executa os três passos exigidos no modelo compressor de imagens proposto: transformada linear, quantização e codificação por entropia. Para a transformação do sinal, estudamos e implementamos a transformada wavelet de Daubechies. A transformada wavelet oferece uma estrutura elegante para a representação do sinal, no qual áreas suaves e descontínuas podem ser representadas compactamente no domínio transformado. Esta habilidade vem das propriedades de análise em multiresolução das wavelets. Após a transformada, os coeficientes foram uniformemente quantizados. Também foi testada uma variação da quantização uniforme, onde uma zona morta foi estabelecida. A zona morta é a região centrada em zero, onde um grande número de coeficientes de pequena magnitude é quantizado para um mesmo símbolo. Um outro quantizador estudado e implementado, foi o algoritmo de Shapiro, que tem uma técnica de quantização mais sofisticada, onde um símbolo especial, o *zerotree*, codifica grandes regiões da imagem. Finalizando o processo, os coeficientes quantizados foram codificados por entropia. Dentro da codificação por entropia, estudamos e implementamos o código de corrida, o código de Huffman e o código aritmético. Também testamos codificar o sinal de média nula, para verificar se o desempenho dos codificadores seria incrementado, em contraste com a codificação do sinal original. Os resultados das simulações mostraram que codificar um sinal de média nula não apresenta ganho considerável. Também apresentamos a técnica de codificação recursiva, que mostrou desempenho superior às técnicas de codificação direta.

Dois parâmetros foram utilizados na análise do software desenvolvido: a taxa de bits por pixel e a relação sinal ruído de pico – PSNR. Analisamos qual a taxa de compressão, isto é, a taxa média de bits por pixel, que cada técnica apresentou para valores de PSNR entre 25 e 35 dB, faixa de valores onde o nível de ruído é considerado aceitável.

O modelo que apresentou o melhor taxa de compressão foi aquele que combinou quantização uniforme e codificação aritmética do sinal original transformado. A combinação do código de corrida com o código de Huffman também apresentou bom nível de compressão, porém com desempenho inferior à combinação anterior.

O algoritmo EZW apresentou o pior nível de compressão entre todos os modelos, além de exigir grande tempo de processamento, devido à varredura executada na classificação dos símbolos. O desempenho deste algoritmo não confere com aquele apresentado em [1], onde taxas de 0,25 bits por pixel são apresentadas para PSNR de 34 dB.

As simulações mostraram que a codificação recursiva tem desempenho superior à codificação direta do sinal. A exceção se deu no algoritmo EZW, onde nenhuma medida adicional na tentativa de aumentar a compressão do sinal surtiu efeito. Também mostraram que a codificação do sinal com média nula não apresenta ganhos que justifiquem sua aplicação em substituição a codificação do sinal original.

Para desenvolvimento futuro novos modelos precisam ser testados e desenvolvidos na tentativa de incrementar o desempenho dos codificadores. Diferentes técnicas são apresentadas em [1], servindo como ponto de partida para novas implementações. A transformada wavelet apresenta um paradigma interessante para a compressão de imagens que poderia continuar sendo explorada objetivando-se otimizar os quantizadores e codificadores. Na verdade, a grande questão seria encontrar códigos que apresentem um casamento perfeito para estas duas etapas.

BIBLIOGRAFIA

[1] G. Davis, A. Nosratinia, "Wavelet-based image coding: An overview", *Applied and Computational Control, Signals, and Circuits*, 1(1), 1998.

[2] R. C. Gonzalez, R. E. Woods, *Digital Image Processing*, Addison-Wesley, Reading, MA, 1993.

[3] Crosswinds, "An Introduction to Image Compression", http://www.crosswinds.net/~sskr/imagecmp/index.htm.

[4] F. Z. Qureshi, "Image Compression Using Wavelet Transform". http://citeseer.nj.nec.com/425270.html.

[5] M. P. Nunes, "Aspectos Formais da Linguagem Python", http://lula.dmat.furg.br/~python/aspectos.html.

[6] M. P. Nunes, "Linguagens de Script", http://lula.dmat.furg.br/~python/linguagens.html.

[7] G. V. Rossum, "Python Reference Manual", Corporation for National Research Initiatives (CNRI), Reston, VA, USA, 1997.

[8] G. V. Rossum, "Python Tutorial", Corporation for National Research Initiatives (CNRI), Reston, VA, USA, 1997.

[9] G. V. Rossum, "Python Library Reference", Corporation for National Research Initiatives (CNRI), Reston, VA, USA, 1997.

[10] M. Lutz, *Programming Python*, O'Reily & Associates, Inc., New York, NY, 1996.

[11] A. Waters, G. V. Rossum, *Internet Programming with Python*, M&T Books, MIS Press, Inc., New York, NY, 1996.

[12] www.python.org.

[13] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, T. Oliphant, "Numerical Python", Lawrence Livermore National Laboratory, Livermore, CA 94566, September 7, 2001.

[14] C. Valens, "A Really Friendly Guide to Wavelets", http://perso.wanadoo.fr/polyvalens/clemens/wavelets/wavelets.html.

[15] W. J. Phillips, "Wavelets and Filter Banks Course Notes", http://www.engmath.dal.ca/course/engm6610/notes.

[16] K. R. Castelman, *Digital Image Processing*, 2ª. edição, Mc. Graw Hill, 1995.

[17] I. Daubechies, *Ten lectures on wavelets,* Philadelphia, Pensylvania, Siam, 1992.

[18] M. Antonini, M. Barlaud, P. Mathieu, and I. Daubechies, "Image coding using wavelet transform", *IEEE Transactions on Image Processing*, vol. 1, 205-220, 1992.

[19] S. G. Mallat, "A theory for multiresolution signal decomposition: The wavelet representation", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, No. 7, 674-693, 1989.

[20] L. F. de Oliveira, A. F. de Oliveira, P. R. Cavalcanti, C. Esperança, "Compressão de Imagens Usando Transformada Wavelet e Curva de Peano-Hilbert", 8º. Encontro Português de Computação Gráfica, http://virtual.inesc.pt/virtual/8epcg/actas/c22/index.htm#Índice.

[21] B.B. Hubbard, *The World According to Wavelet*s, A. K. Peters Wellesley, Massachusetts, 1995.

[22] M. Vetterli, J. Kovacevic, *Wavelets and Subband Coding*, Englewood Cliffs, NJ, Prentice Hall, 1995.

[23] M. L. Hilton, B.D. Jawerth, A. Sengupta, " Compressing Still and Moving Images with Wavelets", *Multimedia Systems*, Vol. 2 and No. 3, Apr. 18, 1994.

[24] E.J. Stollnitz, T.D. DeRose and D.H. Salesin, "Wavelets for Computer Graphics: A Primer", TR 94-09-11, Computer Science and Engineering Dept., Washington Uni., Seattle, Washington, 1994.

[25] N. S. Jayart, *Waveform Quantization and Coding*, IEEE Press, NY, 1976.

[26] A. Gersho, R. M. Gray, *Vector Quantization and Signal Compression*, Kluwer Academic Publishers, 1991.

[27] S. P. Lloyd, "Least Squares Quantization in PCM", *IEEE Transactions on Information Theory*, vol. 28, 129-137, Março, 1982.

[28] J. Max, "Quantization for Minimum Distortion", *IEEE Transactions on Information Theory*, 7-12, 1960.

[29] J. M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients", *IEEE Transactions on Signal Processing*, vol.41, no. 12, Dezembro 1993.

[30] C. Valens, "EZW Encoding", http://perso.wanadoo.fr/polyvalens/clemens/ezw/ezw.html.

[31] K. Skretting, J. H. Husoy, S. O. Aase, *Improved Huffman Coding Using Recursive Splitting*. www.norsig.no/norsig99/Articles/skretting.pdf.

[32] N. Abramson, *Information Theory and Coding*. Mc Graw Hill Book Company, 1963.

[33] K. Rao, P. Yip, *The Discrete Cosine Transform.* New York. Academic Press, 1990.

[34] W. Pennebaker, J. Mitchell, *JPEG still image data compression standard,* Van Nostrad Reinhold, NY, 1993.

[35] M. Nelson. Arithmetic Coding + Statistical Modeling = Data Compression. http://dogma.net/markn/articles/arith/.

[36] J. Rissanen, GG. Langdon, "Arithmetic Coding", IBM J. Research Development, vol. 23, 149-162, 1979.

[37] M. W. Marcellin and T. R. Fischer, "Trellis coded quantization of memoryless and Gauss-Markov sources", *IEEE Transactions on Communications*, vol. 38, 82–93, Janeiro 1990.

[38] Z. Xiong, K. Ramchandran, M. T. Orchard, "Space-frequency quantization for wavelet image coding," *IEEE Transactions on Image Processing*, vol. 6, 677–693, Maio 1997.

[39] W. Sweldens, "The lifting scheme: A new philosophy in biorthogonal wavelet constructions", *Wavelet Applications in Signal and Image Processing III* (A. F. Laine and M. Unser, eds.), 68–79, Proc. SPIE 2569, 1995.

APÊNDICE A TUTORIAL PYTHON

A.1 INTRODUÇÃO

O material deste apêndice cobre a linguagem de programação Python. Informações complementares podem ser obtidas consultando [7], [8], [9], [10] e [11]. O Python é uma linguagem de alto nível, interpretada e orientada a objetos e está disponível para as plataformas mais usadas atualmente, como Windows, Unix e Linux. É uma linguagem de código aberto e está disponível para cópia gratuita em [12]. Este apêndice, na seção A.2, começa traçando um contexto para as linguagens de alto nível e as linguagens de script. A seção A.3 faz uma breve análise léxica da linguagem. A seção A.4 descreve como é a sintaxe da linguagem, como se comportam as variáveis, instruções de controle, como criar funções e módulos. A seção A.5 resume o apêndice.

A.2 LINGUAGENS DE ALTO NÍVEL E LINGUAGENS DE SCRIPT

As linguagens de programação tradicionais surgiram no final da década de 1950 como uma opção para a programação em Assembler. Estas linguagens conseguiram otimizar o trabalho dos programadores pois abstraíram o processo de alocação de registradores e de memória, abstraíram as chamadas a procedimentos e abstraíram as iterações e as condições com comandos tipo *while* e *if*. Também surgiu neste período o conceito de tipagem. Além disso, dados e memória tornaram-se entidades separadas incapazes de comunicarem-se entre si. Essas linguagens foram denominadas *linguagens de alto nível*.

As conseqüências destas inovações foram a diminuição da complexidade das soluções, a diminuição na quantidade de código necessário para implementação dos programas e a maior documentação dos códigos. Por outro lado, também houve uma diminuição no desempenho dos programas, já que foram adotadas soluções genéricas para problemas de baixo nível, não sendo, muitas vezes, as mais adequadas para uma determinada situação. Esse fator foi suprido pela rápida evolução dos sistemas computacionais. Apesar disto, algumas soluções ainda são exclusivas do Assembler, pois alguns problemas ultrapassam o escopo das linguagens de alto nível.

As linguagens de script (Basic, Perl, Python) vêm tomar seu lugar ao lado das linguagens de alto nível e das linguagens de baixo nível, considerando a existência de um conjunto de componentes suficientemente completos, escritos em linguagens de alto nível tradicionais, que lhes permita criar sistemas simplesmente aglutinando estes módulos em um sistema maior. São também utilizadas para estender recursos de um componente.

Como já referido no capítulo 1, seção 1.2, as linguagens de script são interpretadas, ao invés de compiladas. Nas linguagens compiladas, o texto (ou código-fonte) do programa é lido por um programa chamado compilador, que cria um arquivo binário, executável. Em contrapartida, programas escritos em linguagens interpretadas não são convertidos em um arquivo executável. Eles são executados utilizando um outro programa, o interpretador, que lê o código-fonte e executa o programa diretamente.

A linguagem Python se encaixa dentro desse contexto de linguagem de script interpretada. É considerada como o estado da arte em sintaxe e semântica. Possui uma característica extremamente simples e ao mesmo tempo completa. É uma linguagem orientada a objetos.

Orientação a objetos é uma forma de estruturar um programa: ao invés de definirmos variáveis e criarmos funções, passando parâmetros entre elas, definimos objetos que possuem dados e ações associadas. O programa orientado a objetos é resultado da 'colaboração' entre esses objetos. Para a orientação a objetos ser utilizada, a linguagem de programação deve dar suporte a objetos (e aos seus tipos, as classes). Em Python, há suporte completo a orientação a objeto; aliás, a linguagem vai além de simples suporte: todos os elementos básicos em Python são objetos.

No Python, não precisamos declarar variáveis; para criar uma, basta atribuir um valor a ela. Dizemos que a linguagem possui tipos de variáveis dinâmicos, significando que o tipo de valor ao qual um nome está associado pode variar durante a execução de um programa. Não quer dizer que não exista tipo – embora em Python não o declaremos, as variáveis assumem um tipo – apenas que este tipo pode variar durante o curso da execução. Esta propriedade por si só não é tão importante, embora realmente torne a escrita dos programas mais ágil: não precisamos decidir de imediato qual informação será armazenada, e podemos converter uma informação entre diversos tipos sem definir uma nova variável [5], [6].

A.3 PYTHON: ANÁLISE LÉXICA

O interpretador Python interpreta sentenças recebidas de algum dispositivo de entrada (console ou arquivo). Esta entrada é lida pelo analisador léxico que divide o código recebido em *identificadores*, repassando-o para o parser que interpreta o programa. É totalmente definido utilizando a tabela ASCII de sete bits.

A.3.2 Identação

O Python não apresenta, como em outras linguagens, delimitadores de bloco, como *begin* e *end* no Pascal e o conjunto de chaves, {.}, no C. Ao invés disso, o Python utilizada a própria identação do programa como referência para determinação do escopo dos blocos. Cada linha lógica possui um nível de identação, determinado pelos espaços em branco no início de cada linha. Os níveis de identação de linhas consecutivas

são usados para gerar os identificadores INDENT e DEDENT, com o auxílio de uma pilha. Antes que a primeira linha do arquivo seja lida, um zero é colocado na pilha e de lá não será retirado. Os números colocados na pilha estarão rigorosamente incrementando da base até o topo. No começo de cada linha lógica, o nível de identação é comparado com o número no topo da pilha. Se for igual, nada acontece. Se for maior, a pilha recebe o valor do nível de identação da linha e um identificador INDENT é gerado. Se for menor, a pilha é desempilhada até que se encontre o valor do nível da linha. Se este valor não for encontrado, o interpretador acusa um erro de sintaxe. Para cada valor que sai da pilha é gerado um identificador DEDENT. No final de cada arquivo um identificador DEDENT é gerado para cada número restante na pilha maior que zero.

O exemplo a seguir ilustra esse conceito.

```
      def perm(N):                                          NEWLINE
            # Compute the list of all permutations of N
INDENT      if ( len(N) <= 1 ):                             NEWLINE
INDENT           return [N]                                 NEWLINE
DEDENT      r = []                                          NEWLINE
            for i in range( len(N) ):                       NEWLINE
INDENT           s = N[:i] + N[i+1:]                        NEWLINE
                 p = perm(s)                                NEWLINE
                 for x in p:                                NEWLINE
INDENT                r.append(N[i:i+1] + x)                NEWLINE
DEDENT
DEDENT      return r
DEDENT
```

O exemplo a seguir ilustra erros de identação:

```
   def ex1(arg):            # erro: primeira linha identada
 for i in range(5):         # erro: não identada
     x = 5*x
        y = y/2             # erro: identação inesperada
  for x in range(2): print x
return x + y                # erro: dedentação inconsistente
```

A.3.1 Linhas

Um programa em Python é dividido em linhas lógicas que são separadas pelo identificador *NEWLINE* e pode ser constituída por uma ou mais linhas físicas. Estas linhas físicas são trechos de programas divididos pelo caractere *ENTER*. Uma linha lógica não pode ultrapassar uma linha física, exceto em dois casos especiais. O primeiro caso consiste no agrupamento de linhas físicas em uma única linha lógica com a utilização de uma barra invertida no final de cada linha.

```
>>> soma = a + b + c + d + e + f + g \
           + h + i + j + k + l + m
```

O segundo consiste no agrupamento de linhas físicas em uma única linha lógica com a utilização de delimitadores de expressões, como colchetes e parênteses.

```
def compressor( wavelet, type, level, quantization_level,
                quantization_type, compressor )

daub4_coef = [ 0.33267055, 0.80689151, 0.45987751, -0.13501102,
              -0.08544127, 0.03522629 ]
```

Os comentários são identificados pelo caractere sustenido `(#)`. O conteúdo à direita deste símbolo é desconsiderado. Linhas em branco também são desconsideradas.

A.3.3 Outros identificadores

Além dos identificadores NEWLINE, INDENT e DEDENT, existem outras categorias de identificadores: *delimitadores, operadores, palavras chave e literais* (strings, inteiros, inteiros longos, números em ponto flutuante e números imaginários).

Delimitadores

Os seguintes identificadores servem como delimitadores na gramática:

```
(...)      [...]      {...}      ,      :      .      `
  =          ;          +=       -=     *=     /=     %=
 **=        &=          |=       ^=     >>=    <<=
```

Os caracteres `@, $` e `?` não são usados em Python.

Operadores

Os seguintes identificadores são operadores:

```
+    -    *    **    /    %    >>    <<    &    |
^    ~    >    <    <=    >=    ==    !=    <>
```

Palavras chave

Os identificadores a seguir são usados como palavras reservadas:

```
and       del       for       is        raise
assert    elif      from      lambda    return
break     else      global    not       try
class     except    if        or        while
continue  exec      import    pass
def       finally   in        print
```

Strings

Uma string é uma cadeia de caracteres, uma forma de dado muito comum e possui um tipo específico. É delimitada por aspas simples ou duplas. Pode ocorrer também com o delimitador aspas triplas.

Números inteiros simples e inteiros longos

Um inteiro é uma seqüência de dígitos decimais, sucedidos de ‘X’ se hexadecimal, ou precedido de ‘0’ (zero) se octal. Inteiros longos são sucedidos pela letra ‘L’ ou ‘l’. Ou seja, 123 é um inteiro simples, 0x80 um inteiro hexadecimal, 0123 um inteiro octal e 3L um inteiro longo.

Números em ponto flutuante e números imaginários

Números em ponto flutuante são números representados por um separador decimal ‘.’ e uma parte fracionária. Números imaginários são representados como um par de números em ponto flutuante seguidos pela letra ‘j’ ou ‘J’. Exemplos:

```
3.14     10.     1e100     (3+4j)     3.14j     10j
```

A.4 PYTHON BÁSICO: SINTAXE E VARIÁVEIS

A.4.1 Criando um módulo

Em Python, um arquivo contendo instruções da linguagem é chamado de módulo. Um programa pode ser dividido em diversos arquivos e facilmente integrar as funções definidas neles. Para criar um arquivo contendo um programa, basta usar qualquer editor de arquivos texto. O shell do Python já vem com editor próprio, que facilita o trabalho de criação de módulos.

A.4.2 Variáveis e valores

Nomes de variáveis devem começar sempre com uma letra, e assim como tudo em Python, são sensíveis a caixa (*case-sensitive*) – em outras palavras, minúsculas e maiúsculas são coisas diferentes. A variável não precisa ser declarada anteriormente, e nem ter um tipo fixo. São criadas quando um valor lhes é atribuído, e destruídas quando não forem mais usadas. A seguir alguns exemplos de variáveis:

```
>>> quantidade = 1              # inteiro
>>> produto = "detergente"      # string
>>> preco = 2.23                # ponto flutuante, ou float.
```

Para ser considerado um float, o número deve possuir um ponto e uma casa decimal, mesmo que seja zero. O fato de ser considerado um float é muito importante para a operação divisão, pois dependendo do tipo dos operandos, a divisão é inteira ou em ponto flutuante.

```
>>> a = 2           # um inteiro
>>> b = 2.0         # um float
>>> print 5 / a
2
>>> print 5 / b
2.5
```

Para descobrir o tipo atual de uma variável, pode-se usar a função interna `type()`. Note, no exemplo a seguir que o tipo da variável mudou automaticamente na segunda atribuição:

```
>>> a = 1
>>> print type(a)
<type 'int'>
>>> a = "um"
>>> print type(a)
<type 'string'>
```

A.4.3 Listas

Uma lista é um vetor de valores indexados por um número inteiro. O primeiro índice é sempre zero, e os índices são atribuídos seqüencialmente. Como não há tipos definidos, a lista pode conter quaisquer valores, inclusive de tipos mistos, e até outras listas. Para declarar a lista, usamos colchetes:

```
>>> numeros = [ 1, 2, 3 ]
>>> opcoes = [ "nao", "sim", "talvez" ]
>>> modelos = [ 3.1, 3.11, 95, 98, 2000, "Millenium", "XP" ]
>>> listas = [ numeros, opcoes, modelos ]
```

Neste exemplo, quatro listas diferentes foram criadas, com elementos de diversos tipos. A quarta lista é uma lista de listas:

```
>>> listas
[[1, 2, 3], ['nao', 'sim', 'talvez'],
[1, 2, 3.1, 3.11, 95, 98, 2000, 'Millenium', 'XP']]
```

Para acessar um elemento específico de uma lista, usamos o operador colchetes:

```
>>> opcoes[2]
talvez
>>> modelos[4]
95
```

Usando índices negativos, as posições são acessadas a partir do final da lista:

```
>>> numeros[-1]
3
>>> modelos[-2]
Millenium
```

As listas, assim como as strings (seção 2.4.5), podem ser fatiadas como o uso do delimitador dois pontos. Uma fatia é uma porção de uma lista, e é produzida indexando-se a lista com um ou dois índices e o delimitador dois pontos. A fatia contém os elementos do primeiro ao segundo índice, não incluindo o segundo. Se omitirmos um dos índices, assume-se início ou fim da lista, dependendo da posição do delimitador.

```
>>> sm = modelos[4:8]
>>> sm
[95, 98, 2000, 'Millenium']
>>> numeros[:1]
[1]
>>> opcoes[1:]
['sim', 'talvez']
```

A lista possui funções próprias, que podem ser acessadas digitando-se um ponto após seu nome, e a seguir o nome da função. Podem ser modificadas, ao contrário das strings. A função `append()` adiciona um elemento ao final da lista; a função `sort()`, ordena a lista. Seguindo o exemplo anterior:

```
>>> numeros.append(0)
>>> numeros
[1, 2, 3, 0]
>>> numeros.sort()
>>> numeros
[0, 1, 2, 3]
>>> numeros[2] = 100
>>> numeros
[0, 1, 100, 3]
```

A.4.4 Strings

A string, como dito anteriormente, é uma cadeia de caracteres. Podem ser delimitadas tanto com aspas simples quanto com aspas duplas. Aspas triplas também podem ocorrer. Para usar o mesmo tipo de aspas que o delimitador como parte da string, deve-se prefixá-la com uma contra-barra `\`. Podem ser usados caracteres especiais para denotar quebra de linha `(\n)`, tabulação `(\t)` e outros.

```
>>> a = """ Hoje\n\t é o primeiro dia """
>>> b = '\n\t do resto de nossas \'vidas\'.'
>>> print a,b
 Hoje
     é o primeiro dia
     do resto de nossas 'vidas'.
```

Assim como as listas, as strings podem ser concatenadas com o operador + e repetidas com o operador *:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

O primeiro índice é zero, como em C. Podem ser fatiadas com o uso do delimitador dois pontos e não podem ser modificadas, ao contrário das listas:

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

A string possui um operador particular, a porcentagem (%), utilizada para substituição de símbolos. Consiste em colocar um dos símbolos seguintes:
%d : para inteiros, **%f** : para float, **%s** : para strings
na posição da string onde deseja inserir um dado, seguido novamente pelo operador e pelo dado ou pela variável que contém o dado. Exemplos:

```
>>> total = 10
>>> custo = 5.50
>>> a = "Total de itens: %d" % total
>>> b = "Custo: %f"
>>> print a
Total de itens: 10
>>> print b % custo
Custo: 5.500000
>>> print "Fornecedor: %s, Custo %f" % ( "hungry.com", 40.30 )
Fornecedor: hungry.com, Custo 40.300000
```

Pode-se usar um número logo após a porcentagem para reservar um tamanho total à string. Se este número é maior do que o tamanho da string, espaços à esquerda são inseridos; caso contrário, a impressão ocorre normalmente:

```
>>> print "Nome: %20s\nNome: %10s" % ( "Voltaire", "Rousseau" )
Nome:             Voltaire
Nome:   Rousseau
```

É possível controlar a formatação dos dados, usando símbolos no formato *%m.n*. Como acima, *m* é o total de caracteres reservados à string. Para ponto flutuante, *n* indica o número de casas decimais; para inteiros, indica o tamanho total do número, mas preenchendo com zeros à esquerda caso necessário.

```
>>> e = 2.7313
>>> pi = 3.1415
>>> sete = 7
>>> print "Euler = %.7f" % e       # 7 casas decimais
Euler = 2.7313000
>>> print "Pi = %10.3f" % pi       # 10 espaços, 3 casas decimais
Pi =      3.142
>>> print "Sete = %10.2d" % sete   # 10 espaços, 2 dígitos
Sete =         07                 # (é um inteiro)
```

A.4.5 Operadores e Delimitadores

Os operadores e delimitadores apresentados na seção 2.3.3 são muito simples de serem utilizados. Seguem alguns exemplos:

```
>>> a = 7              # atribuição
>>> a += 3             # adição e atribuição a = 10
>>> a / 2              # divisão inteira: resultado inteiro
3
>>> a / 2.5            # divisão float: pelo menos
2.8                    # um argumento deve ser float
>>> a % 4              # resto da divisão
3
>>> a ** 2             # exponenciação
49
>>> a *= 5             # multiplicação e atribuição, a = 50
>>> a = 0x80           # atribuição em hexadecimal
>>> a >> 1             # deslocamento de 1 posição à direita
64
>>> a <<= 2            # deslocamento de 2 posições à esquerda e
                       # atribuição, a = 512.
```

Operadores condicionais

Em Python, falso é denotado por zero ou None *(nada)*. Os operadores condicionais podem ser aplicados em inteiros e floats e também em strings e listas, não sendo, porém de muita utilidade no último caso. Retornam 0 se o resultado da operação for falso e 1, caso contrário. Exemplos:

```
>>> 1 < 2             # menor
1
>>> 3 > 5             # maior
0
>>> 2 == 4            # igualdade
0
>>> 2 != 4            # diferença
1
>>> 2.5 == 5 / 2      # divisão inteira e igualdade
0
>>> 2.5 == 5 / 2.     # divisão com float e igualdade
1
```

A.4.6 Instruções de controle

Os operadores condicionais descritos acima são especialmente úteis quando utilizados junto com instruções de controle. Podem ser separadas em três tipos: *condicionais*, *laços* e *exceções.*

Condicionais

A instrução condicional básica do Python é o *if*. A sintaxe é descrita a seguir (lembrando que a identação é que delimita o bloco):

```
if condição:
    # bloco 1 de código
elif condição:
    # bloco 2
else:
    # bloco final
```

O código acima tem o seguinte significado. Se a condição testada com `if` for verdadeira, o bloco 1 será executado, senão `(elif)`, o bloco 2 será executado. Se nenhum dos dois (ou mais) testes for verdadeiro, será executado o bloco final `(else)`.

Laços

Existem dois tipos de laços em Python: for e `while`. O laço `for` é um iterador que opera sobre os itens de uma seqüência (lista ou string), executando as operações dentro do laço. Utiliza a palavra chave `in`. Exemplo:

```
>>> a = ['gato', 'janela']
>>> for x in a: print x, len(x)
gato 4
janela 6
```

Para fazer um laço com um número fixo de iterações, costuma-se usar o `for` em conjunto com a função range, que gera seqüências de números. Exemplo:

```
>>> range(3)
[0, 1, 2]
>>> range(1,10,2)
[1, 3, 5, 7, 9]
>>> for j in range(1,4): print "%da. volta" % j
1a. volta
2a. volta
3a. volta
```

O segundo tipo de laço, `while`, é utilizado quando necessitamos fazer um teste de condição a cada iteração do laço. Exemplo:

```
>>> m = 3 * 19
>>> n = 5 * 13
>>> contador = 0
>>> while ( m < n ):
        m = n / 0.5
        n = m / 0.5
        contador += 1
>>> print "Iteramos %d vezes." % contador
Iteramos 510 vezes.
```

Exceções

Mesmo que uma linha de código ou bloco de código esteja sintaticamente correto, ele pode causar um erro quanto tentamos executá-lo. Erros detectados durante a execução são chamados exceções. As exceções em Python são definidas num módulo chamado `exceptions`. Este módulo não precisa ser explicitamente importado (módulos são cobertos no item 2.4.6): as exceções estão pré-definidas diretamente no espaço de trabalho. Muitas exceções não são tratadas por programas e resultam em erros:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1
ZeroDivisionError: integer division or modulo
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1
NameError: spam
```

A última linha da mensagem de erro indica o que aconteceu. Exceções aparecem em diferentes tipos, e o tipo é mostrado como parte da mensagem: no exemplo os tipos são `ZeroDivisionError` e `NameError`. A string mostrada no tipo de exceção é o nome da exceção pré-definida que ocorreu. O restante da linha é um detalhe cuja interpretação depende do tipo de exceção: seu significado é dependente do tipo de exceção. A parte precedente da mensagem de erro mostra o contexto onde a exceção ocorreu.

Exceções são meios de interromper o fluxo normal de controle de um bloco de código para tratamento de erros e outras condições excepcionais. Uma exceção é *levantada* no ponto onde o erro é detectado; ela pode ser *tratada* pelo código ao seu redor ou por qualquer código que possa direta ou indiretamente ser invocado pelo código onde o erro ocorreu. O controle de exceções é especificado com um bloco `try...except`. Um bloco `try...finally`, quando utilizado, especifica um código de limpeza que não trata a exceção, mas que é executado se uma exceção ocorrer ou não no bloco. O exemplo a seguir, ilustra os conceitos acima:

```
>>> a = [ 1, 2, 3 ]
>>> try: a[5]
    except IndexError: print "Atenção, posição inválida!"
    finally: print "Fim do teste"
Atenção, posição inválida!
Fim do teste
```

A instrução `try` captura as exceções no código do bloco seguinte e as testa de acordo com o bloco `except`.

A.4.7 Funções

Funções, como seria de se esperar numa linguagem de programação, também são suportadas em Python. Dividem-se em funções pré-definidas e funções definidas pelo usuário.

Funções nativas

A linguagem possui uma série de funções pré-definidas que já estão disponíveis quando executamos o interpretador, sem ter que recorrer a bibliotecas externas. Algumas funções importantes seguem:

- `range(a,b,c)`: recebe um, dois ou três inteiros como argumentos. Retorna uma lista de inteiros. Se chamada com um argumento (*a*), a lista de inteiros criada será de zero até o valor de '*a*' menos um. Se chamada com dois argumentos (*a,b*), a lista criada vai de *a* e *b*, não incluindo *b*. Com três argumentos (*a,b,c*), a lista criada vai de *a* até *b*, pelo passo definido em *c*, não incluindo *b*. Esta função é comumente utilizada para iterar laços for.
- `len(a)`: retorna o comprimento do argumento a: para listas retorna o número de elementos; para strings, o número de caracteres.
- `round(a,n)`: recebe um float e um número, retorna o float arredondado com este número de casas decimais.
- `pow(a,n)`: recebe dois inteiros; retorna o resultado da exponenciação de *a* à ordem *n*.
- `min(a,b)`: retorna o menor entre *a* e *b*, sendo aplicável tanto a tipos numéricos quanto a strings.
- `max(a,b)`: retorna o maior entre *a* e *b*.
- `dir(a)`: retorna uma lista de atributos válidos para o objeto definido no argumento. Útil para listar o conjunto de funções contidas num determinado módulo.

Python tem também um conjunto de funções para manipulação de arquivos:

- `open(modo)`: abre um arquivo. Fornece uma referência, com a qual o arquivo pode ser manipulado. As demais funções para manipulação de arquivos devem ser instanciadas por esta referência. Os modos de abertura podem ser 'r', 'w' ou 'a', para leitura, escrita ou expansão. O arquivo será criado, se ele não existe, quando aberto para leitura e expansão; será truncado quando aberto para escrita. Adicione um 'b' para arquivos binários e um '+' para leitura e escrita simultânea. Exemplo: f = open('dados_pessoais.txt')
- `f.close()`: para fechar o arquivo e liberar os recursos do sistema.

- `f.readline()`: lê uma única linha do arquivo.
- `f.write(string)`: escreve o conteúdo de uma string no arquivo.
- `f.seek()`: para controlar a posição do objeto no arquivo.

Existem outras funções importantes implementadas em módulos independentes. A maior parte das funções relevantes ao desenvolvimento desta tese encontra-se no módulo *Numerical Python*, e serão detalhadas na seção 2.4.6.

Funções definidas pelo usuário

A palavra chave `def` é utilizada na definição de uma função. Deve ser seguida pelo nome da função, mais uma lista de argumentos. O bloco de código que forma o corpo da função começa na próxima linha, e deve estar identado. As primeiras linhas de código podem ser opcionalmente strings, servindo como uma documentação para a função. Os argumentos ficam disponíveis na função como variáveis locais. A sintaxe para definir uma função é mostrada a seguir:

```
def nome_da_funcao (arg1, arg2, arg3 ):
    """Documentação da função."""
    # corpo da função
    # ...
    #
    return valor_de_retorno       # opcional
```

O exemplo abaixo ilustra como criar uma série de Fibonacci. Nesta função, é apresentado um tipo de sintaxe suportada pelo Python, que é atribuição de várias variáveis numa mesma linha. As variáveis a serem atribuídas são posicionadas à esquerda do sinal de igual, separadas por vírgula. Após o sinal de igual, estão os valores de atribuição, também separados por vírgula.

```
>>> def fib(n):
        "Imprime na tela a série de Fibonacci até n"
        a, b = 0, 1
        while b < n:
            print b,
            a, b = b, a+b
>>> # Agora execute uma chamada da função:
>>> fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Uma forma bastante útil é especificar valores aos argumentos da função. Isto cria uma função que pode ser chamada com menos argumentos do que os definidos. Exemplo:

```
>>> def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
      while 1:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'): return 1
        if ok in ('n', 'no', 'nop', 'nope'): return 0
        retries -= 1
        if retries < 0: raise IOError, 'refusenik user'
        print complaint
```

Esta função poderia ser chamada como: `ask_ok('Do you really want to quit?')` ou `ask_ok('OK to overwrite the file?', 2)`.

A.4.8 Módulos

Conforme referenciado na seção 2.4.1, cada arquivo contendo código Python é considerado um módulo. Existem diversas ocasiões que necessitamos usar mais de um arquivo: o programa pode ser grande demais, ou tem partes reutilizáveis que devem ser separadas, ou ainda utiliza módulos escritos por terceiros. A instrução básica para manipular módulos é o `import modulo`. O módulo deve ser um arquivo com a extensão `.py` e deve estar no caminho de busca do Python. Um exemplo de importação:

```
>>> import math
>>> math.sqrt(16)
4.0
```

Importamos o módulo `math` e invocamos a função `sqrt` nele contida. Quando um módulo é importado com a instrução `import`, o nome do módulo deve preceder a utilização das funções nele contidas, seguidas de ponto. Isto pode ser evitado com a utilização da instrução `from` *`modulo`* `import *` ou `from` *`modulo`* `import` *`função`*. Quando utilizando a primeira forma, todas as funções contidas no módulo são trazidas para o ambiente de trabalho e não precisam mais ser referenciadas pelo módulo original. A segunda forma, importa apenas a função, ou grupo de funções definidas no campo *`função`*. Esta função também não precisa mais ser referenciada ao módulo original. Portanto:

```
>>> from math import sqrt, log10
>>> sqrt(16)
4.0
>>> log10(10)
1.0
```

Módulos importantes

Há uma lista grande de módulos que se instalam juntamente com o interpretador Python, e ainda outros módulos disponíveis externamente. Segue uma lista dos mais comumente utilizados.

- **`sys:`** oferece várias operações referentes ao próprio interpretador. Inclui: `path`, uma lista dos diretórios de busca de módulos do python, `argv`, a lista de parâmetros passados na linha de comando e `exit()`, uma função que termina o programa.
- **`time:`** oferece funções para manipular valores de tempo. Inclui: `clock()`, que retorna o tempo desde que o Python foi inicializado, em segundos; `sleep(n)`, que pausa a execução por n segundos; e `ctime(n)`, que converte o tempo em segundos para uma string.
- **`os:`** oferece funções que se referem ao ambiente de execução do sistema. Inclui: `mkdir()`, que cria diretórios; `rename()`, que altera nomes e caminhos de arquivos; e `system`, que executa comandos do sistema.
- **`string:`** oferece funções de manipulação de string (que também estão disponíveis como funções membros da string). Inclui: `split(c,s,p)`, que divide a string c em até p partições separadas pelo separador s, retornando-as numa lista; `lower(c)`, que retorna a string c convertida em minúsculas; e `strip(c)`, que retorna c removendo espaços e quebras de linha do seu início e fim.
- **`math:`** funções matemáticas gerais. Inclui: `cos(x)`, que retorna o cosseno de x; `hypot(x,y)`, que retorna a distância euclidiana entre x e y; e `exp(x)`, que retorna o exponencial de x.
- **`random:`** geração de números aleatórios. Inclui: `random()`, que retorna um número randômico entre 0 e 1; `randrange(m,n)`, que retorna um randômico entre m e n; `choice(s)`, que retorna um elemento randômico de uma seqüência s.
- **`MLab:`** uma série de funções importadas do MatLab. Inclui `mean(x)`, para calcular a média da seqüência x; `median(x)`, para obter a mediana de x; entre outras.

A documentação do Python inclui uma descrição detalhada de cada um dos módulos e de suas funções. Existem, além dos módulos distribuídos com o Python, no entanto, vários módulos auxiliares, distribuídos e lançados independentemente.

Módulos independentes

Há uma série de módulos externos importantes que não serão descritos na sua totalidade; terão apenas uma breve introdução. Exceção feita ao módulo *Numerical Python*, que tem um conjunto de funções relevantes a esta tese. São eles:

- **Win32pipe**: permite em Windows executar programas win32 e capturar sua saída em uma string para manipulação posterior.
- **Ia636**: conjunto de funções para processamento de imagens desenvolvido pelo Prof. Dr. Lotufo, do Departamento de Computação e Automação da Faculdade de Engenharia Elétrica e Computação da Unicamp. Inclui funções para processamento, manipulação e exibição de imagens.
- **HTMLgen**: uma biblioteca de classes que gera documentos HTML conforme alguns padrões pré-definidos. Oferece classes para manipular tabelas, listas, e outros elementos de formatação.
- **DB-API**: Database-Application Programming Interface, na verdade, um conjunto de módulos que acessam bases de dados de uma forma padronizada. A API especifica uma forma documentada de se fazer consultas e operações em bases de dados relacionais (SQL); os módulos implementam esta API para cada tipo de base de dados.
- **mx**: oferece uma série de extensões à linguagem, incluindo operações complexas de data e hora, funções nativas estendidas, e ferramentas para processamento de texto.
- **WxPython**: uma biblioteca de classes que permite construir aplicações gráficas multi-plataforma usando Python.
- **Numpy**: provê mecanismos simples e de alta performance para manipular matrizes multidimensionais; ideal para operações numéricas de alto volume que necessitem de velocidade.

O módulo Numerical Python [13]

As extensões Numerical Python (NumPy) consistem de um conjunto de funções que permitem a manipulação de matrizes multidimensionais. O conjunto de objetos com o qual o Numeric trabalha é denominado *array*. Os objetos array são geralmente coleções homogêneas de grandes números. Todos os números num multiarray devem ser do mesmo tipo. Os objetos tipo array não podem ser nulos. Operações matemáticas realizadas em arrays retornam novos arrays contendo o resultado destas operações realizadas *elemento a elemento*. Os arrays suportam todas as operações de fatiamento (*slicing*) descritas para as listas. As principais funções para criação e manipulação desses multiarrays serão descritas a seguir.

- `array()`**:** função utilizada para criar arrays. Uma lista quando for alvo de manipulação por uma função do Numeric, e atribuída a uma variável, também retornará um array. Exemplo:

  ```
  >>> a = array([1,2,3]).
  ```

  O tipo de elementos pode ser definido no momento que o array é criado:

  ```
  >>> a = array([1,2,3], Float)
  ```

  Arrays multidimensionais também podem ser criados:

  ```
  >>> mult_array = array([[1,2,3],[4,5,6]])
  ```

- `shape(array)`: retorna o formato do array. Esta função também suporta a sintaxe `nome_do_array.shape`.
- `typecode()`: retorna uma descrição do tipo de elemento contido no array. Somente suporta a sintaxe `nome_do_array.typecode()`. Não recebe argumentos.
- `arange()`: função similar a `range()`, exceto que retorna um array, ao invés de uma lista. Recebe um, dois ou três argumentos.
- `reshape(a,(dimensions))`: para mudar as dimensões de um array *a* para as dimensões especificadas em `dimensions`. O formato do array pode ser modificado para qualquer formato desde que o produto de todos os comprimentos de todos os eixos seja mantido constante.
- `resize(a, (new_shape))`: recebe um array *a* como primeiro argumento e retorna um novo array com o formato especificado no argumento `new_shape`. O array retornado, diferentemente de `reshape()`, pode assumir qualquer formato.
- `zeros((shape),typecode)`: cria um array composto somente por zeros no formato especificado em `shape`, e no tipo especificado em `typecode`. Se `typecode` não for especificado, o tipo inteiro é assumido.
- `ones((shape),typecode))`: cria um array composto somente por uns no formato especificado em `shape`, e no tipo especificado em `typecode`. Se `typecode` não for especificado, o tipo inteiro é assumido.
- `astype(type)`: converte um array para o tipo especificado em `type`. Somente suporta a sintaxe `nome_do_array.astype(type)`.
- `nonzero(a)`: retorna um array contendo os índices dos elementos em *a* que são diferentes de zero. Funciona somente em arrays unidimensionais.
- `ravel(a)`: retorna o array *a* como um array unidimensional.
- `sort(a, axis=-1)`: retorna um array contendo uma cópia dos dados em *a*, com o mesmo formato, mas com os valores ordenados ao longo do eixo especificado. Para ordenar nas linhas, use `sort(a)`. Para ordenar as colunas, use `sort(a,0)`.
- `argsort(a, axis=-1)`: retorna os índices dos elementos de *a* necessários para produzir `sort(a)`.
- `concatenate((`a_0`,`a_1`,...,`a_n`), axis=0)`: retorna um novo array contendo cópias dos dados nos arrays $a_0, a_1, \ldots, a_n$. Os arrays estarão concatenados ao longo do eixo especificado. Para concatenar ao longo das linhas, use `concatenate((a,b))`. Para concatenar ao longo das colunas, use `concatenate((a,b),1)`. A concatenação ao longo das linhas somente ocorre quando *a* e *b* possuem o mesmo número de colunas; e ocorre ao longo das colunas somente quando possuem o mesmo número de linhas.
- `transpose(a, axis=None)`: retorna um array que corresponde ao array de entrada com a ordem dos eixos especificada pelo segundo argumento.

- `sum(a, axis=0)`: retorna a soma de todos os elementos na seqüência dada ao longo do eixo especificado. Para somar as colunas, use `sum(a)`, para somar nas linhas, use `sum(a,1)`.

Existem ainda outras funções que não foram incluídas. Aquelas funções que utilizam o parâmetro eixo como segundo argumento precisam ser revisadas, de modo que a sintaxe utilizada seja a mesma em todos os casos, evitando casos de ambigüidade, como aqueles que ocorrem com as funções `sum` e `concatenate`, onde o parâmetro eixo igual a 1 na função `concatenate`, executa a concatenação ao longo das linhas, enquanto o parâmetro eixo igual a 1 na função `sum`, executa a soma ao longo das colunas, o que aumenta a complexidade para uso de algumas funções.

A.5 RESUMO DO APÊNDICE A

O material apresentado neste apêndice cobre a linguagem de programação Python. Inicialmente foi traçado um contexto para as linguagens de script. Na seção A.2 foi feita uma análise léxica da linguagem, onde temas como identação, delimitadores, operadores e palavras chave foram discutidos. A seção A.3 fez uma breve introdução da linguagem, mostrando como utilizar os operadores e delimitadores apresentados na seção anterior, como criar e manipular listas e strings, como utilizar laços for e while e tratar exceções. Também foram cobertos temas como funções e módulos. As funções foram separadas em funções nativas da linguagem e funções criadas pelo usuário. Ainda nesta seção, foram introduzidos os módulos. Especial atenção foi dada ao módulo Numerical Python, devido à variedade de funções que facilitam o processamento de imagens. Temas mais avançados, como as classes, não foram discutidos neste tutorial, já que o protótipo desenvolvido não utilizou esta ferramenta.

APÊNDICE B1 CÓDIGO FONTE TRANSFORMADA WAVELET

```
#=======================================================
from Numeric import *
import time
import MLab
#=======================================================
def getfilters(type):
    """ getfilters(): coeficientes dos filtros digitais para a wavelet de Daubechies
    Entrada:
        type: tipo do filtro: db4, db6 ou db8
    Saída: quatro filtros
        LP_D: coeficientes do filtro passa-baixa para transformada direta
        HP_D: coeficientes do filtro passa-alta para transformada direta
        LP_R: coeficientes do filtro passa-baixa para transformada inversa
        HP_R: coeficientes do filtro passa-alta para transformada inversa"""
    SQ2 = sqrt(2)
    if ( type == 'db4' ):
        LP_D = array([ 0.48296291, 0.83651631, 0.22414386, -0.12940952])
        [LP_D,HP_D,LP_R,HP_R] = taps(LP_D,1)
    elif ( type == 'db6' ):
        LP_D = array([ 0.33267055, 0.80689151, 0.45987751, -0.13501102,
                    -0.08544127, 0.03522629 ])
        [LP_D,HP_D,LP_R,HP_R] = taps(LP_D,1)
    elif ( type == 'db8' ):
        LP_D = array([ 0.23037781, 0.71484657, 0.63088077, -0.02798377,
                    -0.18703481, 0.03084138, 0.03288301, -0.01059741 ])
        [LP_D,HP_D,LP_R,HP_R] = taps(LP_D,1)
                     0.37740285, -0.85269867,  0.37740285,  0.11062440,
                    -0.02384946, -0.03782845 ] )
    return array([LP_D, HP_D, LP_R, HP_R])

#=======================================================

def taps(LP_D, HP_D):
    """ taps(): operações para determinar os coeficientes dos filtros
        da transformada inversa a partir dos filtros da transformada direta
    Entrada:
        LP_D: coeficientes do filtro passa-baixa da transformada direta
        HP_D: coeficientes do filtro passa-alta da transformada direta (somente
            para Biortogonal Splines - para Daubechies, este parâmetro é 1)
```

```
    Saída: ver getfilters."""

    LP_R = zeros(len(LP_D), Float)
    HP_R = zeros(len(LP_D), Float)
    if ( HP_D == 1 ):
        HP_D = zeros(len(LP_D), Float)
        for j in range(len(LP_D)): HP_R[j] = ((-1)**(j+1)) * LP_D[j]
        HP_D = HP_R[::-1]
        LP_R = LP_D[::-1]
    return [LP_D, HP_D, LP_R, HP_R]

#=====================================================

def dwt(img, wavelet='db4', level=2):
    """ dwt(): Transformada Wavelet Direta
    Entrada:
        img: imagem original
        wavelet: tipo de wavelet: db4, db6 ou db8
        level: nível de decomposição
    Saída:
        f: imagem transformada """

    img = array(img)
    size = array(img.shape)
    img = img.astype(Float)
    [ lp, hp ] = getfilters(wavelet)[0:2]
    f = 1*img-128
    for i in range(level):
        subimg = f[:size[0],:size[1]]
        decomp_img = multres(multres(subimg,lp,hp),lp,hp)
        f[:size[0],:size[1]] = 1*decomp_img
        size = size/2
    return f

#=====================================================

def multres(img,lp,hp):
    """ multres(): analise em multiresolução
    Entrada:
        img: imagem a ser transformada. Para mais de um nível de decomposição,
            este parâmetro será o conteúdo da subbanda LL mais alta.
        lp: filtro passa-baixa direto
        hp: filtro passa-alta direto
    Saída:
```

```
        img: imagem filtrada e dizimada por 2. """
    x,y = img.shape
    n = len(lp);
    newimg = concatenate((img, img[:,:n]), 1)
    lp_result = conv2(newimg,lp)[:,n-1:-n:2]
    hp_result = conv2(newimg,hp)[:,n-1:-n:2]
    img = transpose( concatenate( (lp_result,hp_result), 1) )
    return img

#=====================================================

def idwt(f,wavelet='db4',level=2):
    """ idwt(): Transformada wavelet inversa
    Entrada:
        f: imagem transformada
        wavelet: tipo do filtro: db4, db6 ou db8. Deve ser idêntico ao filtro
            da transformada direta
        level: nível de reconstrução. Idêntico ao nível de decomposição da
            transformada direta
    Saída:
        f: imagem recuperada. """
    s = sqrt(f.shape[0])
    f.shape = (s,s)
    f = f.astype(Float)
    [lp_r, hp_r] = getfilters(wavelet)[2:]
    size = array(f.shape)/(2**level)
    for i in range(level):
        size = 2*size
        subimg = f[:size[0],:size[1]]
        reconstr_img = imultres( subimg, lp_r, hp_r )
        f[:size[0],:size[1]] = 1*reconstr_img
    f = f+128
    f = where(f<0,0,f)
    f = where(f>255,255,f)
    return f.astype(Int)

#=====================================================

def imultres(img, lp_r, hp_r):
    """ imultres(): analise em multiresolução inversa
    Entrada:
        img: imagem transformada. Para mais de um nível de decomposição,
            esta imagem é o conteúdo da subbanda LL mais alta.
        lp_r: filtro passa-baixa inverso
```

```
        hp_r: filtro passa-alta inverso
    Saída:
        recov_img: imagem no nível anterior de decomposição recuperada."""
    x,y = img.shape
    n = len(lp_r)
    half = x/2
    #----------columns----------#
    lowres = transpose( img[:half,:] )     # take top-half and transpose
    lowres = concatenate( (lowres[:, half-(n/2) : half], lowres), 1 ) # padding
    highres = transpose( img[half:,:] )    # take bottom-half and transpose
    highres = concatenate( (highres[:, half-(n/2) : half], highres), 1 )#padding
    img = upnconv(lowres,lp_r) + upnconv(highres,hp_r)
    img = transpose(img[:,n:n+x])
    #----------rows----------#
    lowres = img[:,:half]                  # take top-half
    lowres = concatenate( (lowres[:, half-(n/2) : half], lowres), 1)
    highres = img[:,half:]                 # take bottom-half
    highres = concatenate( (highres[:, half-(n/2) : half], highres), 1)
    recov_img = upnconv(lowres,lp_r) + upnconv(highres,hp_r)
    recov_img = recov_img[:,n:n+y]
    return recov_img

#=========================================================

def upnconv(img,filter):
    """ upnconv(): interpolação e filtragem inversa
    Entrada:
        img: conteúdo da banda sendo recuperada
        filter: filtro correspondente.
    Saída:
        convolved: imagem interpolada e filtrada. """
    x,y = img.shape
    upsampling = zeros((x,2*y),Float)
    upsampling[:,:2*y:2] = img
    convolved = conv2(upsampling,filter)
    return convolved

#=========================================================

def conv2(f, h):
    """ conv2(): convolução bidimensional
    Entrada:
        f: sinal a ser filtrado
        h: coeficiente do filtro
```

```
    Saída:
        y: sinal filtrado. """
    f, h = f.astype(Float), h.astype(Float)
    if len(f.shape) == 1: f = reshape(f, (1, f.shape[0]))
    if len(h.shape) == 1: h = reshape(h, (1, h.shape[0]))
    if product(f.shape) < product(h.shape): f, h = h, f
    y = zeros((f.shape[0]+h.shape[0]-1, f.shape[1]+h.shape[1]-1),Float)
    for i in range(h.shape[0]):
        for j in range(h.shape[1]):
            y[i:i+f.shape[0],j:j+f.shape[1]] += h[i,j] * f
    return y

#=============================================================

def psnr(f, g):
    """ psnr(): relação sinal ruído de pico
    Entrada:
        f: imagem original
        g: imagem recuperada
    Saída:
        PNSR: relação sinal ruído de pico
        MSE: erro quadrático médio. """
    error = f - g
    if ( len(f.shape) == 2 ):
        m,n = f.shape
        MSE = (sum(sum(error * error))) / (m*n)
    else: MSE = sum(error*error) / len(f)
    if ( error ):
        PSNR = 20*log10(255/sqrt(MSE))
        return [PSNR, MSE]
    else: return MSE

#=============================================================

def histogram(dt):
    """ histogram(): histograma """
    dt = ravel(dt)
    counter = zeros(max(dt)+1)
    for j in range(len(dt)): counter[dt[j]] += 1
    return counter

#=============================================================

def log2(x):
```

```
    """ log2(): logaritmo na base 2 """
    return log10(x) / log10(2)

#=========================================================
def entropia(counter):
    """ entropia(): entropia do sinal """
    if ( len(counter.shape) == 1 ):
        L = sum(counter)
        nz_counter = compress(counter>0, counter)
        entropy = log2(L) - sum( nz_counter*log2(nz_counter) ) / L
    return entropy, len(nz_counter)

#=========================================================

def output_bit(bit):
    """ output_bit(): codifica um bit no vetor de saída """
    global bit_stream, bitpos, buffer
    buffer >>= 1
    if ( bit ): buffer |= 0x80
    bitpos -= 1
    if ( not bitpos ):
        bit_stream.append(buffer)
        buffer = 0
        bitpos = 8
    return

#=========================================================

def input_bit():
    """ input_bit(): lê um bit do vetor de entrada"""
    global bit_stream, bitpos, buffer
    if ( not bitpos ):
        buffer = bit_stream.pop(0)
        bitpos = 8
    bit = buffer & 1
    buffer >>= 1
    bitpos -= 1
    return bit

#=========================================================
```

APÊNDICE B2

CÓDIGO FONTE QUANTIZAÇÃO UNIFORME

```
#=======================================================
def uq(dt, qlevels):
    """ uq(): quantização uniforme
        Entrada:
            dt: sinal a ser quantizado
            qlevels: níveis de quantização
        Saída:
            data2coder: sinal quantizado
            mmin: valor mínimo do sinal a ser quantizado
            mmax: valor máximo do sinal a ser quantizado
            step: passo de quantização. """

    try: dt = ravel(dt)
    except: pass
    mmin = min(dt)
    mmax = max(dt)
    step = abs((mmax-mmin)/qlevels)
    table = arange( mmin, mmax, step )
    data2coder = quantization(dt, table)
    return data2coder, mmin, mmax, step

#=======================================================

def quantization(f, table):
    """ quantization(): mapeamento dos dados sendo quantizados
    Entrada:
        f: sinal sendo quantizado
        table: tabela de quantização
    Saída:
        index: sinal quantizado. """
    index = zeros(len(f))
    for i in range(len(table)): index += ( f >= table[i] )
    return index

#=======================================================

def iuq(dt, mmin, mmax, step):
    """ iuq(): quantização uniforme inversa
    Entrada:
```

```
        dt: dados quantizados
        mmin: valor mínimo do sinal quantizado
        mmax: valor máximo do sinal quantizado
        step: passo de quantização.
    Saída:
        f: sinal dequantizado. """
    try: dt = ravel(dt)
    except: pass
    dt = dt.astype('l')
    table = arange( mmin, mmax, step )
    f = zeros(len(dt))
    for i in range(len(dt)): f[i] = table[dt[i]-1] + step/2
    return f

#=====================================================

def dzq(f, qlevels, deadzone):
    """ dzq(): quantização zona morta
    Entrada:
        f: sinal a ser quantizado
        qlevels: numero de níveis de quantização
        deadzone: zona morta
    Saída:
        data2coder: sinal quantizado
        mmin: valor mínimo do sinal a ser quantizado
        mmax: valor máximo do sinal a ser quantizado
        step: passo de quantização. """
    try: f = ravel(f)
    except: pass
    data2coder = zeros(len(f))
    aux1 = compress(f > deadzone, f)
    aux2 = compress(f < -deadzone, f )
    dt1, mmin1, mmax1, step1 = uq(aux1, qlevels)
    dt2, mmin2, mmax2, step2 = uq(aux2, qlevels)
    dt2 += qlevels
    mmin = [mmin1, mmin2]
    mmax = [mmax1, mmax2]
    step = [step1, step2]
    put(data2coder, nonzero(f > deadzone), dt1)
    put(data2coder, nonzero(f <- deadzone), dt2)
    return data2coder, mmin, mmax, step

#=====================================================
```

```
def idzq(dt, mmin, mmax, step):
    """ iuq(): quantização zona morta inversa
    Entrada:
        dt: dados quantizados
        mmin: valor mínimo do sinal quantizado
        mmax: valor máximo do sinal quantizado
        step: passo de quantização.
    Saída:
        f: sinal dequantizado. """
    try: dt = ravel(dt)
    except: pass
    f = zeros(len(dt))
    aux1 = compress( (where(dt > max(dt)/2, 0, dt) ), dt)
    aux2 = compress(dt > max(dt)/2, dt)
    aux2 -= max(dt)/2
    data1 = iuq(aux1, mmin[0], mmax[0], step[0])
    data2 = iuq(aux2, mmin[1], mmax[1], step[1])
    put(f, nonzero(where(dt > max(dt)/2, 0, dt)), data1)
    put(f, nonzero(dt > max(dt)/2), data2)
    return f

#=====================================================
```

APÊNDICE B3

CÓDIGO FONTE ALGORITMO EZW

```
#=======================================================
def ezw_encoder(*args):
    """ ezw_encoder(): codificador EZW
    Entrada:
        args[0]: imagem transformada
        args[1]: nivel de refinamento. Quanto maior, mais grosseiro o
            refinamento. Múltiplos de 2.
    Saída:
        bit_stream: fluxo de bits. """

    global image, threshold, lista_sub, fifo
    global bit_stream, bitpos, buffer
    global ZERO, ONE, POS, NEG, ZTR, IZ

    # parâmetros iniciais
    ZERO = 0 # binário 0
    ONE = 1  # binário 1
    POS = 2  # binário 00
    NEG = 3  # binário 01
    ZTR = 4  # binário 10
    IZ = 5   # binário 11
    image = args[0]
    bit_stream = []
    lista_sub = []
    fifo = []
    bitpos = 8
    buffer = 0
    if ( len(image.shape)==1 ):
        print 'squared image required'
        return
    maxx = MLab.max(MLab.max(abs(image)))
    threshold = 1 << int( floor(log2(maxx)) )

    # header
    ezw_size_encoder( int(log2(len(image))) )
    ezw_size_encoder( int(log2(threshold)) )

    # coder
    while ( threshold > args[1] ):
```

```
        coder_dominant_pass()
        coder_subordinate_pass( threshold >> 1 )
        threshold >>= 1

    # remaining bits
    if ( bitpos ):
        buffer >>= bitpos
        bit_stream.append(buffer)

    return bit_stream

#=======================================================

def coder_dominant_pass():
    """ Executa um passo dominante. Códigos do passo dominante são
        enviados para o vetor de saída e a lista subordinada é atualizada. """
    global image, threshold, lista_sub, fifo
    global ZERO, ONE, POS, NEG, ZTR, IZ

    x,y = 0,0
    code = process_element(image[y,x],x,y)
    output_code(code)

    x,y = 1,0
    code = process_element(image[y,x],x,y)
    fifo.append([x,y,code])

    x,y = 0,1
    code = process_element(image[y,x],x,y)
    fifo.append([x,y,code])

    x,y = 1,1
    code = process_element(image[y,x],x,y)
    fifo.append([x,y,code])

    x, y, code = fifo.pop(0)
    output_code(code)

    while ( fifo ):
        if ( code != ZTR ):
            min_x = x << 1
            max_x = min_x+2
            min_y = y << 1
            max_y = min_y+2
```

```
            if (max_x <= image.shape[0] and max_y <= image.shape[1]):
                for y in range(min_y,max_y):
                    for x in range(min_x,max_x):
                        code = process_element(image[y,x],x,y)
                        fifo.append([x,y,code])
        x, y, code = fifo.pop(0)
        output_code(code)
    return

#=============================================================

def coder_subordinate_pass(threshold):
    """ Executa um passo subordinado """
    global image, lista_sub, fifo
    global ZERO, ONE, POS, NEG, ZTR, IZ

    if ( threshold > 0 ):
        for i in range(len(lista_sub)):
            d = int( floor(lista_sub[i]) )
            if ( d & threshold ): output_code(ONE)
            else: output_code(ZERO)
    return

#=============================================================

def process_element(temp,x,y):
    """ Verifica qual o tipo de símbolo, coloca na lista subordinada
        se NEG ou POS e zera o elemento. """
    global image, threshold, lista_sub, fifo
    global ZERO, ONE, POS, NEG, ZTR, IZ

    # identifica o elemento
    if ( abs(temp) >= threshold ):
        if (temp >= 0): code = POS
        else: code = NEG
    else:
        if ( zerotree(temp,x,y) ): code = ZTR
        else: code = IZ

    # coloca na lista subordinada se NEG ou POS e zera o elemento
    if ( code == POS or code == NEG ):
        lista_sub.append( abs(temp) )
        image[y,x] = 0
    return code
```

```
#=======================================================


def zerotree(temp,x,y):
    """ retorna 1 se a arvore descendente é uma zerotree. """
    global image, threshold, lista_sub, fifo
    stop = 0
    if ( x==0 and y==0 ): stop = 1
    else:
        min_x = x << 1
        min_y = y << 1
        max_x = (x+1) << 1
        max_y = (y+1) << 1
        if (min_x == image.shape[0] or min_y == image.shape[1]): return 1
        while (max_y <= image.shape[0] and max_x <= image.shape[1]):
            for i in range(min_y, max_y):
                for j in range(min_x, max_x):
                    temp = abs(image[i,j])
                    if ( temp >= threshold ):
                        stop = 1
                        break
                if ( stop ): break
            if ( stop ): break
            min_x <<= 1
            max_x <<= 1
            min_y <<= 1
            max_y <<= 1
    if ( stop ): return 0
    return 1

#=======================================================

def output_code(code):
    """ codificação dos símbolos. """
    global ZERO, ONE, POS, NEG, ZTR, IZ
    if (code == ZERO):
        output_bit(0)
    elif (code == ONE):
        output_bit(1)
    elif (code == POS):
        output_bit(0)
        output_bit(0)
    elif (code == NEG):
```

```
        output_bit(0)
        output_bit(1)
    elif (code == ZTR):
        output_bit(1)
        output_bit(0)
    elif (code == IZ):
        output_bit(1)
        output_bit(1)
    else: print 'Failure'
    return

#=========================================================

def ezw_size_encoder(N):
    """ codificação de um inteiro N. """
    if ( N <= 31):
        for i in range(5):
            output_bit( N & 1 )
            N >>= 1
    else: print 'Failure'
    return

#=========================================================

def ezw_decoder(*args):
    """ ezw_decoder(): decodificador EZW
    Entrada:
        args[0]: sinal a ser decodificado
        args[1]: nivel de refinamento. Múltiplo de 2. Idêntico ao codificador.
    Saída:
        image: imagem decodificada. """
    global image, threshold, lista_sub, fifo
    global bit_stream, bitpos, buffer
    global ZERO, ONE, POS, NEG, ZTR, IZ

    try: bit_stream = args[0].tolist()
    except: bit_stream = args[0]
    buffer = bit_stream.pop(0)
    ZERO = 0 # binario 0
    ONE = 1  # binario 1
    POS = 2  # binario 00
    NEG = 3  # binario 01
    ZTR = 4  # binario 10
    IZ = 5   # binário 11
```

```
    bitpos = 8
    lista_sub = []
    fifo = []
    m = 1 << ezw_size_decoder()
    threshold = 1 << ezw_size_decoder()
    image = zeros((m,m))

    # decoder
    while ( threshold > args[1] ):
        decoder_dominant_pass()
        decoder_subordinate_pass( threshold >> 1 )
        threshold >>= 1

    return image

#=========================================================

def decoder_dominant_pass():
    """ Executa um passo dominante inverso """
    global image, threshold, x, y, lista_sub, fifo
    global ZERO, ONE, POS, NEG, ZTR, IZ, input1, input2

    x,y = 0,0
    code = get_input_element()

    x,y = 1,0
    code = get_input_element()
    fifo.append([x,y,code])

    x,y = 0,1
    code = get_input_element()
    fifo.append([x,y,code])

    x,y = 1,1
    code = get_input_element()
    fifo.append([x,y,code])

    x,y,code = fifo.pop(0)

    while ( fifo ):
        if ( code != ZTR ):
            min_x = x << 1
            max_x = min_x+2
            min_y = y << 1
```

```
            max_y = min_y+2
            m,n = image.shape
            if ( max_x <= m and max_y <= n ):
                for y in range(min_y,max_y):
                    for x in range(min_x,max_x):
                        code = get_input_element()
                        fifo.append([x,y,code])
        x,y,code = fifo.pop(0)
    return

#=======================================================

def decoder_subordinate_pass(threshold):
    """ Executa um passo subordinado inverso. """
    global image, lista_sub
    global ZERO, ONE, POS, NEG, ZTR, IZ

    if ( threshold > 0 ):
        for i in range(len(lista_sub)):
            x,y = lista_sub[i]
            temp = image[y,x]
            if ( get_input_code(1) ):
                if ( temp < 0 ): image[y,x] = temp - threshold
                else: image[y,x] = temp + threshold
    return

#=======================================================

def get_input_element():
    """ Identifica os elementos significantes, POS ou NEG,
        e os coloca na imagem recuperada. """
    global image, threshold, x, y, lista_sub
    global ZERO, ONE, POS, NEG, ZTR, IZ

    code = get_input_code(0)
    if ( code == POS or code == NEG ):
        if ( code == POS ): image[y,x] = threshold
        else: image[y,x] = -threshold
        lista_sub.append([x,y])
    return code

#=======================================================

def get_input_code(flag):
```

```
    """ Identifica o símbolo no fluxo de bits de entrada. A flag sinaliza
        qual o tipo de símbolo, dominante ou subordinado. """
    global ZERO, ONE, POS, NEG, ZTR, IZ

    if ( input_bit() ):
        if ( flag ): return ONE
        else:
            if ( input_bit() ): return IZ
            else: return ZTR
    else:
        if ( flag ): return ZERO
        else:
            if ( input_bit() ): return NEG
            else: return POS

#=========================================================

def ezw_size_decoder():
    """ decodificador de inteiros """
    N = 0
    for i in range(5):
        if ( input_bit() ): N += 2**i
    return N

#=========================================================
```

APÊNDICE B4

CÓDIGO FONTE
CÓDIGO DE CORRIDA

```
#=====================================================
def rle(*args):
    """ rle(): código de corrida de zeros
    Entrada:
        um parâmetro: decodificador. Ex: rle(data)
        dois parâmetros: codificador. Ex: rle(data,0)
    Saída: output: sinal codificado ou decodificado. Depende da entrada. """

    input = args[0]
    if ( len(args) == 2 ): #  ===== compressor =====
        index = 0
        count = 0
        i = 0
        output = zeros(len(input))
        while ( i < len(input) ):
            if ( input[i] == 0 ):
                count += 1
                output[index+1] = count
                if ( count == 255 ):
                    count = 0
                    index += 2
            else:
                if ( count > 0 ):
                    index += 2
                    count = 0
                output[index] = input[i]
                index += 1
            i += 1
        if ( not count ): return output[:index]
        else: return output[:index+2]
    else: #   ===== descompressor =====
        i = 0
        output = []
            while ( i < len(input) ):
              if ( input[i] == 0 ):
                    count = input[i+1]
                    output += zeros(count)
                    i += 2
              else:
```

```
            output += [ input[i] ]
            i += 1
    return output
```

APÊNDICE B5

CÓDIGO FONTE
CÓDIGO DE HUFFMAN

```
#=======================================================
def huffman(*args):
    """ huffman(): código de Huffman
        Ver: huff_encoder(), huff_decoder()
    Entrada:
        um argumento: decodificador
        dois argumentos: codificador
    Saída: sinal codificado ou decodificado. Depende da entrada."""
    global bit_stream, bitpos, buffer
    bitpos = 8
    if ( len(args) == 2 ):
        buffer = 0
        x = array(args[0])
        bit_stream = []
        huff_encoder(x)
        if ( bitpos ):
            buffer >>= bitpos
            bit_stream.append(buffer)
        bit_rate = ( len(bit_stream)*8. ) / 65536
        return bit_stream, bit_rate

    else:
        bit_stream = args[0]
        buffer = bit_stream.pop(0)
        x = huff_decoder()
        return x

#=======================================================

def huff_encoder(*args):
    """ huff_encoder(): determina as palavras de Huffman e codifica o sinal,
        ou executa a divisão recursiva. Um argumento, codifica o sinal.
        Quatro argumentos, perfaz a divisão recursiva.
        Ver: statistics(), output_bit(), huff_size_encoder(),
            hufflen_encoder, huffcode()."""

    if ( len(args)==1 ):
        x = args[0]
        bits, HL, L = statistics(x)
```

```
    if ( len(args) == 4 ): x, bits, L, HL = args
    if ( L > 50 ):
        xm = MLab.median(x)
        x1 = zeros(L)
        x2 = zeros(L)
        x2[0] = x[0]
        i1, i2 = 0, 1
        for i in range(1,L):
            if ( x[i-1] <= xm ):
                x1[i1] = x[i]
                i1 += 1
            else:
                x2[i2] = x[i]
                i2 += 1
        x1,x2 = x1[:i1], x2[:i2]
        bits1, HL1, L1 = statistics(x1)
        bits2, HL2, L2 = statistics(x2)
    else: bits1, bits2 = bits, bits

    if ( ( bits1 + bits2 ) < bits ):
        output_bit(1)
        bits1 = huff_encoder( x1, bits1, L1, HL1 )
        bits1 = huff_encoder( x2, bits2, L2, HL2 )
    else:
        output_bit(0)
        huff_size_encoder(L)
        hufflen_encoder(HL)
        HK = huffcode(HL)
        for i in range(L):
            n = x[i]
            for k in range(HL[n]): output_bit( HK[n,k] )
    return bits

#=====================================================

def statistics(x):
    """ statistics(): estatística dos dados sendo codificados. Consistem
        no histograma do sinal, comprimento das palavras código, e contagem
        do numero de bits necessários para codificar o sinal.
        Ver: hufftable(), hufflen(), histogram. """
    L = len(x)
    counter = histogram(x)
    HL = hufflen(counter)
    bit_counter = hufftable(HL)
```

```
    bit_counter += 6
    if ( L >= 16 ):   bit_counter += 4
    if ( L >= 256 ):  bit_counter += 4
    if ( L >= 4096 ): bit_counter += 4
    bits =  bit_counter + sum(HL*counter)
    return bits, HL, L

#=======================================================

def huff_decoder():
    """ huff_decoder(): decodifica o sinal recursivamente codificado.
        Ver: input_bit(), huff_size_decoder(), hufflen_decoder(), hufftree().
    Entrada: fluxo de bits
    Saída: sinal decodificado. """
    if ( input_bit() ):
        x1 = huff_decoder()
        x2 = huff_decoder()
        L = len(x1) + len(x2)
        aux = concatenate(( x1,x2 ))
        xm = MLab.median( aux )
        x = zeros(L)
        x[0] = x2[0]
        i1, i2 = 0, 1
        for i in range(1,L):
            if ( x[i-1] <= xm ):
                x[i] = x1[i1]
                i1 += 1
            else:
                x[i] = x2[i2]
                i2 += 1
    else:
        L = huff_size_decoder()
        x = zeros(L)
        HL = hufflen_decoder()
        ht = hufftree(HL)
        n, pos, root = 0,0,0
        while ( n < L ):
            if ( input_bit() ): pos = 1*ht[pos,2]
            else: pos = 1*ht[pos,1]
            if ( ht[pos,0] ):
                x[n] = ht[pos,1] - 1
                n += 1
                pos = root
    return x
```

```
#=======================================================

def hufftable(HL):
    """ Executa contagem dos bits necessários na codificação das
        palavras código:
        6-18 bits para o comprimento do vetor de palavras código
        5 bits para dar o comprimento do primeiro símbolo
        Então, para cada um dos próximos símbolos, um código para dizer
        seu comprimento:
          '0'  - mesmo comprimento que o símbolo anterior: bit_counter += 1
          '10' - comprimento incrementou em 1: bit_counter += 2
          '1100' - comprimento reduziu em 1: bit_counter += 4
          '1101' - comprimento incrementou em 2: bit_counter += 4
          '111xxxxx' - atribui o valor do símbolo a xxxxx: bit_counter += 8. """

    L = len(HL)
    seq_length_bits = 6  # 6 bits se comprimento da seq for menor que 16
    if ( L >= 16 ):   seq_length_bits += 4
    if ( L >= 256 ):  seq_length_bits += 4
    if ( L >= 4096 ): seq_length_bits += 4
    bit_counter = seq_length_bits + 5  # + 5 bits p/ a primeira palavra código
    for i in range(1,L):
        if   ( HL[i] == HL[i-1] ):     bit_counter += 1
        elif ( HL[i] == (HL[i-1]+1) ): bit_counter += 2
        elif ( HL[i] == (HL[i-1]-1) ): bit_counter += 4
        elif ( HL[i] == (HL[i-1]+2) ): bit_counter += 4
        else: bit_counter += 8
    return bit_counter

#=======================================================

def hufflen_encoder(HL):
    """ codifica as palavras código de acordo com tabela 4.3.
        Ver:  huff_size_encoder(), output_bit(). """
    L = len(HL)
    huff_size_encoder(L)
    temp = 1*HL[0]
    for i in range(5):
        output_bit( temp & 1 )
        temp >>= 1
    for j in range(1,L):
        if ( HL[j] == HL[j-1] ):
            output_bit(0)
```

```
        elif ( HL[j] == (HL[j-1]+1) ):
            output_bit(1)
            output_bit(0)
        elif ( HL[j] == (HL[j-1]-1) ):
            output_bit(1)
            output_bit(1)
            output_bit(0)
            output_bit(0)
        elif ( HL[j] == (HL[j-1]+2) ):
            output_bit(1)
            output_bit(1)
            output_bit(0)
            output_bit(1)
        else:
            output_bit(1)
            output_bit(1)
            output_bit(1)
            temp = 1*HL[j]
            for i in range(5):
                output_bit( temp & 1 )
                temp >>= 1
    return

#=======================================================

def hufflen_decoder():
    """ Decodifica as palavras código.
        Ver: huff_size_decoder(), input_bit(). """
    L = huff_size_decoder()
    HL = zeros(L)
    for i in range(5):
        if ( input_bit() ): HL[0] += 2**i
    for j in range(1,L):
        if ( input_bit() ):
            if ( input_bit() ):
                if ( input_bit() ):
                    for i in range(5):
                        if ( input_bit() ): HL[j] += 2**i
                else:
                    if ( input_bit() ): HL[j] = HL[j-1] + 2
                    else: HL[j] = HL[j-1] - 1
            else: HL[j] = HL[j-1] + 1
        else: HL[j] = HL[j-1]
    return HL
```

```
#=======================================================

def huff_size_encoder(N):
    """ Codificação de inteiros """
    if ( N < 16 ):
        output_bit(1)
        output_bit(0)
        output_bit(0)
        k = 4
    elif ( N < 256 ):
        output_bit(1)
        output_bit(0)
        output_bit(1)
        k = 8
    elif ( N < 4096 ):
        output_bit(1)
        output_bit(1)
        output_bit(0)
        k = 12
    elif ( N < 65536 ):
        output_bit(1)
        output_bit(1)
        output_bit(1)
        k = 16
    else:
        output_bit(0)
        ezw_size_encoder( int(log2(N)) )
        return
    temp = 1*N
    for i in range(k):
        output_bit( temp & 1 )
        temp >>= 1
    return

#=======================================================

def huff_size_decoder():
    """ decodificador de inteiros """
    N = 0
    if ( input_bit() ):
        if ( input_bit() ):
            if ( input_bit() ): k = 16
            else: k = 12
```

```
        else:
            if ( input_bit() ): k = 8
            else: k = 4
    else:
        N = 1 << ezw_size_decoder()
        return N
    for i in range(k):
        if ( input_bit() ): N += 2**i
    return N

#=====================================================

def hufflen(x):
    """ Determina o comprimento das palavras código. Caminha na arvore
        de Huffman, da fonte mais alta para a mais baixa, para determinar
        cada comprimento. """

    hl = zeros(len(x))
    add = nonzero(ravel(x))
    n = len(add)
    h = zeros(len(add))
    for i in range(len(add)): h[i] = x[add[i]]
    hl_temp = zeros(n)
    c = zeros(2*n-1)
    c[:n] = h
    top = arange(n)
    pos = argsort(-h,0)
    last = n-1
    next = n
    while ( last > 0 ):
        c[next] = c[pos[last]] + c[pos[last-1]]
        addr = nonzero( top == pos[last] )
        for i in range(len(addr)):
            hl_temp[addr[i]] += 1
            top[addr[i]] = next
        addr = nonzero( top == pos[last-1] )
        for i in range(len(addr)):
            hl_temp[addr[i]] += 1
            top[addr[i]] = next
        last -= 1
        pos[last] = next
        next += 1
        count = last-1
        while ( (count >= 0) and (c[pos[count+1]] >= c[pos[count]]) ):
```

```
            temp = 1*pos[count]
            pos[count] = pos[count+1]
            pos[count+1] = temp
            count -= 1
    for i in range(len(add)): hl[add[i]] = hl_temp[i]
    return hl
#=====================================================

def huffcode(hl):
    """ Determina o código de Huffman para cada comprimento.
    Entrada:
        hl: comprimento das palavras código.
    Saída:
        huffcode: código de Huffman. """
    N = len(hl)
    L = max(hl)
    huffcode = zeros((N,L))
    addr = argsort(hl)
    hls = sort(hl)
    code = zeros(L)
    for i in range(N):
        if ( hls[i] > 0 ):
            huffcode[addr[i]] = code
            k = hls[i] - 1
            while ( k >= 0 ):
                code[k] += 1
                if ( code[k] == 2):
                    code[k] = 0
                    k -= 1
                else: break
    return huffcode

#=====================================================

def hufftree(hl):
    """ Constrói uma arvore de decodificação. A primeira coluna contem
    um código, 0 ou 1, dizendo se o símbolo é uma folha ou nó. A segunda
    e terceira colunas contem endereços para outras posições da tabela quando
    o símbolo na primeira coluna é uma folha, ou contém o valor do
    símbolo, quando é um nó. Ver: huffcode().
    Entrada:
        hl: comprimento das palavras de Huffman
    Saída:
        ht: arvore de Huffman."""
```

```
    hk = huffcode(hl)
    N = len(hl)
    ht = zeros((2*N,3))
    root = 0
    next = 1
    for n in range(N):
        if ( hl[n] > 0 ):
            pos = root
            for k in range(hl[n]):
                if ( (ht[pos,0] == 0) & (ht[pos,1] == 0) ):
                    ht[pos,1] = next
                    ht[pos,2] = next+1
                    next += 2
                if ( hk[n,k] ): pos = 1*ht[pos,2]
                else: pos = 1*ht[pos,1]
            ht[pos,0] = 1
            ht[pos,1] = n+1
    return ht

#=======================================================
```

APÊNDICE B6

CÓDIGO FONTE
CÓDIGO ARITMÉTICO

```
#=======================================================
def arith(*args):
    """ arith(): código aritmético.
        Ver: arith_encoder(), arith_decoder(), output_bit(), input_bit()
    Entrada:
        um parâmetro: decodificador
        dois parâmetros: codificador
    Saída: sinal codificado ou decodificado. """

    global bit_stream, bitpos, buffer
    global limite_inferior, limite_superior, intervalo, bits_criticos
    global contador_alto, contador_baixo, escala, codigo

    bitpos = 8
    limite_inferior = 0
    limite_superior = 0xffff
    bits_criticos = 0
    codigo = 0

    if ( len(args) == 2 ):
        buffer = 0
        x = array(args[0])
        bit_stream = []
        arith_encoder(x)

        # flush arithmetic coder
        output_bit( limite_inferior & 0x4000 )
        bits_criticos += 1
        while ( bits_criticos > 0 ):
            output_bit( not( limite_inferior & 0x4000 ) )
            bits_criticos -= 1
        if ( bitpos ):
            buffer >>= bitpos
            bit_stream.append(buffer)
        bit_rate = ( len(bit_stream)*8. ) / 65536
        return bit_stream, bit_rate

    else:
        try: bit_stream = args[0].tolist()
```

```
        except: bit_stream = args[0]
        bit_stream += [0,0,0]
        buffer = bit_stream.pop(0)
        for i in range(20):
            codigo <<= 1
            codigo += input_bit()
        x = arith_decoder()
        return x

#=========================================================

def arith_encoder(x):
    """ arith_encoder(): codificador aritmético ou divisor recursivo.
        Ver: arith_statistics(), arith_size_encoder(), output_a_bit(),
        encode_symbol().
    Entrada:
        x: sinal a ser codificado ou dividido. A regra de divisão leva em
        consideração a entropia dos sinais.
    Saída: fluxo de bits. """

    global limite_inferior, limite_superior, intervalo, bits_criticos
    global contador_alto, contador_baixo, escala, codigo

    L = len(x)
    split = 0
    if ( L > 50 ):
        entropy, L, temp = arith_statistics(x)
        xm = MLab.median(x)
        x1 = zeros(L)
        x2 = zeros(L)
        x2[0] = x[0]
        i1, i2 = 0, 1
        for i in range(1,L):
            if ( x[i-1] <= xm ):
                x1[i1] = x[i]
                i1 += 1
            else:
                x2[i2] = x[i]
                i2 += 1
        x1,x2 = x1[:i1], x2[:i2]
        entropy1, L1 = arith_statistics(x1)[:2]
        entropy2, L2 = arith_statistics(x2)[:2]
        if (( L*entropy - L1*entropy1 - L2*entropy2 ) > ( 5*temp )):
            split = 1
```

```
    if ( split ):
        output_a_bit(1)
        bits1 = arith_encoder(x1)
        bits2 = arith_encoder(x2)
        bits = bits1 + bits2 + 1
    else:
        output_a_bit(0)
        arith_size_encoder(L)
        M0 = min(x)
        if ( M0 == 0 ): output_a_bit(0)
        else:
            output_a_bit(1)
            arith_size_encoder(M0)
        M = max(x)
        arith_size_encoder(M)

        # Fim do cabeçalho e inicio do codificador.
        # Aqui ocorre o modelamento estatístico dos dados.
        counter_table = ones(M+2)
        index_table = arange(M+1,-1,-1)
         for j in range(L):
            escala = counter_table[0]
            m = x[j]
            contador_alto = counter_table[m]
            contador_baixo = counter_table[m+1]
            # Se o teste a seguir passar, significa que um símbolo diferente
            # dos anteriores foi encontrado. É enviado, então, um código de
            # escape (contador_alto=1 e contador_baixo=0), e o símbolo
            # é codificado com a tabela index_table.
            if ( contador_alto == contador_baixo ):
                contador_alto = 1
                contador_baixo = 0
                encode_symbol()
                escala = index_table[0]
                contador_alto = index_table[m]
                contador_baixo = index_table[m+1]
                index_table[:m+1] -= 1    # atualize index_table
            encode_symbol()
            # atualize contador de símbolo.
            counter_table[:m+1] += 1
    bits = 8*len(bit_stream) - bitpos
    return bits

#=====================================================
```

```
def arith_statistics(x):
    """ Levanta as estatísticas do sinal, como histograma e entropia.
        Ver: histogram(), entropia(). """
    L = len(x)
    counter = histogram(x)
    entropy, temp = entropia(counter)
    return entropy, L, temp

#=====================================================

def arith_decoder():
    """ arith_decoder(): decodificador aritmético.
        Ver: arith_size_decoder(), get_input_bit(), remove_symbol(). """

    global limite_inferior, limite_superior, intervalo, bits_criticos
    global contador_alto, contador_baixo, escala, codigo

    if ( get_input_bit() ):
        x1 = arith_decoder()
        x2 = arith_decoder()
        L = len(x1) + len(x2)
        aux = concatenate( (x1,x2) )
        xm = MLab.median( aux )
        x = zeros(L)
        x[0] = x2[0]
        i1, i2 = 0, 1
        for i in range(1,L):
            if ( x[i-1] <= xm ):
                x[i] = x1[i1]
                i1 += 1
            else:
                x[i] = x2[i2]
                i2 += 1
    else:
        L = arith_size_decoder()
        if ( L > 1 ):
            x = zeros(L)
            if ( get_input_bit() ): M0 = arith_size_decoder()
            else: M0 = 0
            M = arith_size_decoder()

            # Inicio do decodificador.
            counter_table = ones(M+3)
```

```
            counter_table[-1] = 0
            index_table = arange(M+1,-2,-1)
            for j in range(L):
                escala = array( counter_table[0] ).astype(Float)
                intervalo = limite_superior - limite_inferior + 1
                count = floor((( codigo-limite_inferior+1 ) * escala - 1) / intervalo)
                m = 1
                while ( counter_table[m] > count ): m += 1
                contador_alto = counter_table[m-1]
                contador_baixo = counter_table[m]
                m -= 1
                remove_symbol()
                if ( m > M ):
                    # decodifique escape simbol a partir da index_table
                    escala = array( index_table[0] ).astype(Float)
                    intervalo = limite_superior - limite_inferior + 1
                    count = floor((( codigo-limite_inferior+1 ) * escala - 1) / intervalo)
                    m = 1
                    while ( index_table[m] > count ): m += 1
                    contador_alto = index_table[m-1]
                    contador_baixo = index_table[m]
                    m -= 1
                    remove_symbol()
                    index_table[:m+1] -= 1 # atualize index_table
                x[j] = m
                # atualize counter_table
                counter_table[:m+1] += 1
    return x

#=====================================================

def arith_size_encoder(N):
    """ Codificador de inteiros """
    if ( N < 16 ):
        output_a_bit(1)
        output_a_bit(0)
        output_a_bit(0)
        k = 4
    elif ( N < 256 ):
        output_a_bit(1)
        output_a_bit(0)
        output_a_bit(1)
        k = 8
    elif ( N < 4096 ):
```

```
        output_a_bit(1)
        output_a_bit(1)
        output_a_bit(0)
        k = 12
    elif ( N < 65536 ):
        output_a_bit(1)
        output_a_bit(1)
        output_a_bit(1)
        k = 16
    else:
        output_a_bit(0)
        N = int(log2(N))
        k = 5
    temp = 1*N
    for i in range(k):
        output_a_bit( temp & 1 )
        temp >>= 1
    return

#=========================================================

def arith_size_decoder():
    """ Decodificador de inteiros """
    N = 0
    if ( get_input_bit() ):
        if ( get_input_bit() ):
            if ( get_input_bit() ): k = 16
            else: k = 12
        else:
            if ( get_input_bit() ): k = 8
            else: k = 4
    else: k = 5
    for i in range(k):
        if ( get_input_bit() ): N += 2**i
    if ( k==5 ): return (1 << N)
    else: return N

#=========================================================

def output_a_bit( bit ):
    """ Codifica o código de escape e outros sinais. """

    global limite_inferior, limite_superior, intervalo, bits_criticos
    global contador_alto, contador_baixo, escala, codigo
```

```
    escala = 2
    if ( bit ):
        contador_alto = 1
        contador_baixo = 0
    else:
        contador_alto = 2
        contador_baixo = 1
    encode_symbol()
    return

#============================================================

def get_input_bit():
    """ Decodifica código de escape e outros sinais """

    global limite_inferior, limite_superior, intervalo, bits_criticos
    global contador_alto, contador_baixo, escala, codigo

    intervalo = limite_superior - limite_inferior + 1
    escala = 2
    count = floor((( codigo-limite_inferior+1 ) * escala - 1) / intervalo)
    if ( 1 > count ):
        bit = 1
        contador_alto = 1
        contador_baixo = 0
    else:
        bit = 0
        contador_alto = 2
        contador_baixo = 1
    remove_symbol()
    return bit

#============================================================

def encode_symbol():
    """ Esta rotina codifica um símbolo. O símbolo é passado como um
    contador alto, um contador baixo e uma escala, ao invés das
    probabilidades tradicionais. A codificação é em dois passos: primeiro,
    os valores dos limites superior e inferior são atualizados para levar
    em consideração a restrição de intervalo criada pelo novo símbolo.
    A seguir, os bits que casarem no MSB são enviados ao fluxo de saída e
    os limites são novamente estabilizados e a rotina continua. """
```

```
    global limite_inferior, limite_superior, intervalo, bits_criticos
    global contador_alto, contador_baixo, escala, codigo

    # estas três linhas reescalam os limites para o novo símbolo
    intervalo = array( limite_superior-limite_inferior+1 ).astype(Float)
    limite_superior = int( limite_inferior+floor(( intervalo*contador_alto / escala ) - 1) )
    limite_inferior = int( limite_inferior+floor( intervalo*contador_baixo / escala ))
    TRUE = 1
    # esta rotina codifica todos os bits até que os limites estejam novamente
    # estabilizados.
    while ( TRUE ):
        # se este teste passar, os MSB casam e podem ser enviados ao fluxo de
        # saida
        if ( (limite_superior & 0x8000) == (limite_inferior & 0x8000) ):
            output_bit( limite_superior & 0x8000 )
            while ( bits_criticos > 0 ):
                output_bit( not(limite_superior & 0x8000) )
                bits_criticos -= 1
        # se este teste passar, os limites estão se aproximada de uma situação
        # critica, porque os MSB não conferem, e os segundos dígitos estão
        # próximos. Desloque o segundo MSB.
        elif ( (limite_inferior & 0x4000) and not(limite_superior & 0x4000) ):
            bits_criticos += 1
            limite_superior |= 0x4000
            limite_inferior &= 0x3fff
        else: break
        limite_superior <<= 1
        limite_superior += 1
        limite_inferior <<= 1
        limite_superior &= 0xffff
        limite_inferior &= 0xffff
    return

#=============================================================

def remove_symbol():
    """ remove o símbolo do fluxo de bits. """

    global limite_inferior, limite_superior, intervalo, bits_criticos
    global contador_alto, contador_baixo, escala, codigo

    # primeiro o intervalo é atualizado para considerar a remoção do símbolo
    intervalo = array( limite_superior-limite_inferior+1 ).astype(Float)
    limite_superior = int(limite_inferior+floor((( intervalo*contador_alto ) / escala ) - 1))
```

```
    limite_inferior = int(limite_inferior+floor(( intervalo*contador_baixo ) / escala))
    TRUE = 1
    # a seguir, todos os bits possíveis são retirados do fluxo.
    while ( TRUE ):
        # se os MSB casarem, os bits serão deslocados.
        if ( (limite_superior & 0x8000) == (limite_inferior & 0x8000) ): pass
        # se a situação critica estiver presente, desloque o segundo MSB.
        elif ( (limite_inferior & 0x4000) and not (limite_superior & 0x4000) ):
            codigo ^= 0x4000
            limite_superior |= 0x4000
            limite_inferior &= 0x3fff
        else: break
        # caso contrário, desloque os limites
        limite_superior <<= 1
        limite_superior += 1
        limite_inferior <<= 1
        codigo <<= 1
        codigo += input_bit()
        limite_superior &= 0xffff
        limite_inferior &= 0xffff
        codigo &= 0xffff
    return

#=========================================================
#=========================================================
#=========================================================
#=========================================================
```