UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

APROXIMAÇÕES DA DCT DE COMPRIMENTO 16 COM BAIXA COMPLEXIDADE ARITMÉTICA PARA COMPRESSÃO DE IMAGENS

TRABALHO DE GRADUAÇÃO

Thiago Lopes Trugillo da Silveira

Santa Maria, RS, Brasil

2014

APROXIMAÇÕES DA DCT DE COMPRIMENTO 16 COM BAIXA COMPLEXIDADE ARITMÉTICA PARA COMPRESSÃO DE IMAGENS

Thiago Lopes Trugillo da Silveira

Trabalho de Graduação apresentado ao Bacharelado em Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de

Bacharel em Ciência da Computação

Orientadora: Profª. Dr. Alice de Jesus Kozakevicius

Co-orientador: Prof. Dr. Fábio Mariano Bayer

Trabalho de Graduação Nº 363
Santa Maria, RS, Brasil

2014

Universidade Federal de Santa Maria
Centro de Tecnologia
Bacharelado em Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

APROXIMAÇÕES DA DCT DE COMPRIMENTO 16 COM BAIXA COMPLEXIDADE ARITMÉTICA PARA COMPRESSÃO DE IMAGENS

elaborado por
Thiago Lopes Trugillo da Silveira

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Fábio Mariano Bayer, Dr.
(Presidente/Co-orientador)

Everton Alceu Carara, Dr. (UFSM)

Leonardo Londero de Oliveira, Dr. (UFSM)

Santa Maria, 20 de Janeiro de 2014.

Ao avanço da Ciência...

AGRADECIMENTOS

A todos aqueles que estiveram comigo durante esses quatro anos de curso e que, de uma forma ou outra, tornaram possível a realização deste trabalho, meus sinceros agradecimentos.

Em especial, agradeço a todos da minha família que, mesmo que às vezes distante, sempre me apoiaram e confortaram. Agradeço enormemente a minha – sempre presente – namorada, Samara Dias Osorio, por estar comigo em todos os momentos, sejam eles de alegria ou angústia.

Agradeço também a todos meus professores pelos mais diversos ensinamentos, incentivos e conselhos. Em especial agradeço aqueles com quem tive o prazer de conviver mais de perto: Professora Dr^{a}. Marcia Pasin, Professora Dr^{a}. Iara Augustin e Professor Dr. Cesar Pozzer. Além destes, agradeço aos, não somente orientadores mas também amigos, Professor Dr. Fábio Bayer e Professora Dr^{a}. Alice Kozakevicius que se fizeram presentes durante meu crescimento profissional e pessoal. Agradeço também aos Professores Dr. Leonardo Londero e Dr. Everton Carara por terem aceito fazer parte da banca avaliadora deste trabalho. Um singular agradecimento ao Professor Dr. Osmar Marchi por ter colaborado e avaliado este trabalho durante a sessão de apresentação de andamento.

Um especial agradecimento a todos os meus colegas de curso com quem, muitas vezes, estive mais do que com minha família. Em especial, agradeço aos amigos Tháygoro Minuzzi, Alberto Kummer, Emmanuel Katende e Otávio Rodrigues por estarem todos estes anos compartilhando ideias, ensinamentos, histórias e gargalhadas.

Agradeço aos orgãos de fomento à pesquisa CNPq, RNP e FIT-UFSM por, durante esses quatro anos de curso, ter auxiliado financeiramente no desenvolvimento de diversos projetos e trabalhos.

Um enorme agradecimento a todos aqueles que não foram diretamente citados mas que contribuíram muito para que eu chegasse até aqui.

“The people who are crazy enough to think
they can change the world are the ones who do.“
— Apple Inc.

RESUMO

Trabalho de Graduação
Bacharelado em Ciência da Computação
Universidade Federal de Santa Maria

APROXIMAÇÕES DA DCT DE COMPRIMENTO 16 COM BAIXA COMPLEXIDADE ARITMÉTICA PARA COMPRESSÃO DE IMAGENS

AUTOR: THIAGO LOPES TRUGILLO DA SILVEIRA
ORIENTADORA: ALICE DE JESUS KOZAKEVICIUS
CO-ORIENTADOR: FÁBIO MARIANO BAYER
Local da Defesa e Data: Santa Maria, 20 de Janeiro de 2014.

Compressão de imagens é uma técnica que visa a redução do espaço de armazenamento de uma imagem em memória secundária. Exemplo de cenário onde esta técnica pode ser aplicada é a transmissão de imagens sobre uma rede. Neste processo, uma imagem comprimida pode ser transpassada por esta rede – ou barramento – em um menor período de tempo se comparada com a mesma imagem não comprimida. Complementando este cenário, pode-se imaginar que os dispositivos de captura de imagens têm baixo poder de armazenamento e processamento. Assim, é importante que estes dispositivos possam comprimir e enviar as imagens através da rede ou barramento com menor custo computacional possível e, consequentemente, em um menor tempo. Usualmente, transformadas discretas – especialmente as de núcleo trigonométrico – são utilizadas em compressão de imagens. A conhecida transformada discreta do cosseno (DCT) é utilizada em importantes padrões de compressão como o JPEG e MPEG-1. Entretanto, nos casos em que a capacidade de processamento é restrita, aproximações de baixo custo computacional desta transformada apresentam-se como opções interessantes. A literatura conta com diversas aproximações da DCT – especialmente de comprimento 8 pelo fato deste ser o comprimento utilizado nos padrões usuais. Entretanto, outros padrões de compressão, como o recente HEVC, utilizam não somente a DCT de comprimento 8, mas também a transformada de comprimentos 4, 16 e 32. Neste sentido, este trabalho faz uma revisão das transformadas de comprimento 16 presentes na literatura até o momento, assim como propõe uma nova transformada de mesmo comprimento. O algoritmo rápido da transformada proposta é livre de multiplicações e possui a mais baixa complexidade arquivada na literatura. Para avaliação da transformada proposta, em contraste com as já existentes, métricas de qualidade de imagens, similaridade com a DCT exata e ganho de codificação são implementadas. Consequentemente, é mostrado que a transformada proposta apresenta bons resultados, baixo custo computacional e potencial para implementações eficientes em *software* e *hardware*.

Palavras-chave: Algoritmos rápidos. Compressão de imagens. DCT. Transformadas aproximadas.

ABSTRACT

Undergraduate Final Work
Graduate Program in Computer Science
Federal University of Santa Maria

16-POINT DCT APPROXIMATIONS WITH LOW ARITHMETIC COMPLEXITY FOR IMAGE COMPRESSION

AUTHOR: THIAGO LOPES TRUGILLO DA SILVEIRA
ADVISOR: ALICE DE JESUS KOZAKEVICIUS
COADVISOR: FÁBIO MARIANO BAYER
Defense Place and Date: Santa Maria, January 20th, 2014.

Image compression is a technique that aims an image storage space reduction in secondary memory. One cenary where this technique can be used is the image transmission in a network. In this process, a compressed image can be transmitted through this network – or data bus – in less time if compared with the same image with no compression. Complementing this cenary, one can imagine that image capture devices have low processing and storage power. Thus, it is important that these devices can compress and, after, transmit images through a network or data bus with the lowest possible computational cost and, consequently, in a shorter time. Usualy, discrete transforms – especially those with trigonometric core – are used in image compression. The well known discrete cosine transform (DCT) is used in important compression standards as JPEG and MPEG-1. However, when the processing capacit is restrict, low computational cost approximations of this transform are presented as interesting options. There exist many DCT approximations in literature – especially 8-point transforms because this block lenght is applied in usual standards. Indeed, other compression standards, as the recent HEVC, use not only the 8-point DCT but also the 4, 16 and 32-point DCT. In this context, this work does a review of all 16-point DCT approximations found in literature so far, as well presents a new transform of same block lenght. The proposed transform's fast algorithm is multiplication free and have the lowest computational cost archived in the literature. To evaluate the proposed transform, in contrast to those found in literature, image quality, DCT similarity and coding gain metrics are implemented. Consequently, it is shown that the proposed transform presents good results, low computational cost and has potential to be efficiently implemented in software and hardware.

Keywords: Fast algorithms, Image compression, DCT, Approximated transforms.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE APÊNDICES

LISTA DE ABREVIATURAS E SIGLAS

BAS	*autores Bouguezel, Ahmad e Swamy*
DCT	*discrete cosine transform*
HEVC	*High Efficiency Video Coding*
IDCT	*inverse discrete cosine transform*
JPEG	*Joint Photographic Experts Group*
KLT	*Karhunen-Loève transform*
MPEG	*Moving Picture Experts Group*
MSE	*mean square error ou erro quadrático médio*
MSSIM	*structural similarity index*
PSNR	*peak signal-to-noise ratio*
SDCT	*signed DCT*
WHT	*Walsh-Hadamard transform*

LISTA DE SÍMBOLOS

$\circ$	Multiplicação de matrizes elemento-a-elemento
$\oslash$	Divisão de matrizes elemento-a-elemento
$\mathrm{round}(\,\cdot\,)$	Função de arredondamento para inteiro
d_2	Métrica de distorção da DCT
$\|\,\cdot\,\|$	Norma euclideana
$\mathrm{diag}(\,\cdot\,)$	Matriz diagonal
$\exp(\,\cdot\,)$	Função exponencial
$\in$	Métrica *Total Error Energy*
$\mathrm{tr}(\,\cdot\,)$	Função traço
C_g	Métrica *Transform Coding Gain*
η	Métrica *Transform Efficiency*
μ_x	Média aritmética dos elementos de x
μ_y	Média aritmética dos elementos de y
σ_x	Desvio padrão dos elementos de x
σ_y	Desvio padrão dos elementos de y
σ_{xy}	Covariância entre x e y

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização e motivação

Muitas vezes, em aplicações de tempo real onde há uma grande quantidade de imagens capturadas em curto período de tempo, o espaço em memória secundária torna-se um recurso escasso e deve ser bem gerenciado. Técnicas de análise de imagens, onde há a detecção e exclusão de imagens menos relevantes à aplicação, podem ser utilizadas visando o aumento do espaço de armazenamento. Frequentemente este tipo de técnica possui um alto custo computacional envolvido e acaba onerando o tempo de execução da aplicação. Além disso, há casos em que a seleção de imagens não é capaz de liberar espaço de armazenamento considerável. Nestes casos, técnicas de compressão de imagens podem ser empregadas visando diminuir o espaço em memória secundária requerido para o armazenamento de dados. A compressão de imagens é uma técnica que visa reduzir a redundância de dados correlacionados de forma que a perda de informações seja mínima ou, ao menos, aceitável para determinada aplicação (FLINT, 2012). Para este processo, frequentemente, transformadas discretas são utilizadas.

No processamento digital de sinais, transformadas discretas desempenham um papel de destaque (BAYER; CINTRA, 2010). Dentre estas transformadas, as com núcleo trigonométrico possuem maior aplicação, como a transformada discreta de Fourier (BRIGGS; HENSON, 1995), a transformada discreta de Hartley (BRACEWELL, 1986), a transformada discreta do seno (BRITANAK; YIP; RAO, 2007) e a transformada discreta do cosseno (DCT) (AHMED; NATARAJAN; RAO, 1974; BRITANAK; YIP; RAO, 2007). Especialmente a DCT tem larga aplicação em compressão de imagens, sendo utilizada em padrões de compressão de imagens como JPEG (PENNEBAKER; MITCHELL, 1992) e de vídeo como MPEG-1 (ROMA; SOUSA, 2007). Em parte, o amplo interesse pela DCT pode ser explicado por esta ser uma importante aproximação da transformada de Karhunen-Loève (KLT) (AHMED; NATARAJAN; RAO, 1974). A KLT tem a distinção de ser ótima em compactação de energia quando o sinal é modelado por meio de um processo markoviano de primeira ordem altamente correlacionado (RAO; YIP, 1990).

As transformadas discretas têm uma conveniente representação matricial que pode ser estendida para o processamento de imagens (CINTRA; BAYER, 2011). Uma imagem pode ser vista como uma matriz de tamanho $N \times N$. Desta forma, a aplicação de uma transformada $\mathbf{T}$ em

uma imagem $\mathbf{A}$ resulta em uma imagem $\mathbf{B}$. Da mesma maneira, a aplicação da transformada inversa ($\mathbf{T}^{-1}$) à imagem $\mathbf{B}$ resulta novamente na imagem original $\mathbf{A}$. Em termos matriciais, as aplicações das transformadas direta e inversa podem ser representadas, respectivamente, por $\mathbf{B} = \mathbf{T} \cdot \mathbf{A} \cdot \mathbf{T}^{-1}$ e $\mathbf{A} = \mathbf{T}^{-1} \cdot \mathbf{B} \cdot \mathbf{T}$. Para mensurar a complexidade computacional envolvida na aplicação dessas transformadas, muitos fatores podem ser considerados. Contudo, o número de operações aritméticas envolvidas é frequentemente utilizado como uma medida de complexidade do algoritmo (BRIGGS; HENSON, 1995). Observa-se que, se uma imagem tem tamanho $N \times N$, a matriz de transformação a ser aplicada deve ser, também, de ordem N. Se calculada de maneira direta, tal método numérico exige operações multiplicativas na ordem de $O(N^3)$ (ROBINSON, 2005). Esse custo computacional pode comprometer aplicações em tempo real e em equipamentos portáteis de baixo poder de processamento (BRITANAK; YIP; RAO, 2007), pois a complexidade computacional do algoritmo influi diretamente nas propriedades físicas do circuito integrado que o implementa, como área e consumo (LIN; LEE, 2007).

Com intuito de diminuir a complexidade aritmética, especialmente operações multiplicativas, pode-se subdividir a imagem de entrada $\mathbf{A}$ em blocos disjuntos de tamanho $k \times k$ e, dessa forma, aplicar a transformada de comprimento k sobre tais blocos. O fato de subdividir a imagem $\mathbf{A}$ em tais blocos não acarreta em perda de informações. O processo inverso, ou seja, a reunião destes blocos, resulta na mesma imagem $\mathbf{A}$. A transformada empregada no padrão JPEG, por exemplo, é a DCT de comprimento $k = 8$. Além desta abordagem, busca-se diminuir o custo aritmético através da fatoração das matrizes das trasformadas (FEIG; WINOGRAD, 1992; BRITANAK; YIP; RAO, 2007), de aproximações zonais (PAO; SUN, 1998; LECUIRE; MAKKAOUI; MOUREAUX, 2012; KOUADRIA et al., 2013), da estimação via momentos (LIU; LIU; WANG, 2005) e, mais recentemente, de métodos aproximados de cálculo da DCT com complexidade multiplicativa nula (BAYER; CINTRA, 2010; CINTRA; BAYER, 2011; BAYER; CINTRA, 2012; BOUGUEZEL; AHMAD; SWAMY, 2013; BAYER et al., 2013; CINTRA; BAYER; TABLADA, 2014) .

No que diz respeito a transformadas aproximadas, a comunidade de processamento de sinais tem dedicado grandes esforços. Apesar de não calcular exatamente a DCT, tais aproximadas são capazes de prover boas estimativas com reduzido custo computacional. Neste sentido, proeminentes técnicas para a DCT de comprimento 8 são a SDCT (HAWEEL, 2001), a aproximação *level* 1 (LENGWEHASATIT; ORTEGA, 2004), a *round-off DCT* (CINTRA; BAYER, 2011), a *round-off DCT* modificada (BAYER; CINTRA, 2012), a DCT aproximada

para aplicação de rádio frequência (POTLURI et al., 2012), e uma série de algoritmos propostos por Bouguezel-Ahmad-Swamy (BAS) (BOUGUEZEL; AHMAD; SWAMY, 2008, 2009, 2010, 2011). Pode-se citar ainda (CINTRA; BAYER; TABLADA, 2014) no qual é apresentado um método para derivar novas transformadas aproximadas baseado em funções inteiras e (BAYER et al., 2013) que propõem uma aproximação para a DCT de comprimento quatro. Para aproximações da DCT de comprimento 16 poucos trabalhos têm sido encontrados na literatura. Algumas alternativas podem ser verificadas em (BAYER et al., 2012), (BOUGUEZEL; AHMAD; SWAMY, 2010) e (BOUGUEZEL; AHMAD; SWAMY, 2013).

Contudo, especialmente com o desenvolvimento do padrão *High Efficiency Video Coding* (HEVC) (POURAZAD et al., 2012), a atenção tem se voltado para o desenvolvimento de aproximações da DCT com comprimentos k diferentes de 8. O HEVC utiliza a DCT com tamanhos de blocos $k = 4,\ 8,\ 16$ e 32. Esses blocos maiores podem acomodar melhoramentos em termos de compactação de energia do sinal, além de outras vantagens visuais (EDIRISURIYA et al., 2012). Neste sentido, este trabalho propõe uma nova aproximação para a DCT de comprimento 16 e compara-a com outras aproximações de mesma comprimento presentes na literatura.

1.2 Objetivos

1.2.1 Objetivo geral

Propor uma nova aproximação de complexidade multiplicativa nula para a DCT de comprimento 16 e compará-la com as aproximações da DCT de mesmo comprimento presentes na literatura.

1.2.2 Objetivos específicos

Para alcançar o objetivo geral proposto neste trabalho, pode-se destacar os seguintes objetivos específicos:

- atualização do estado da arte em transformadas aproximadas de complexidade multiplicativa nula;
- estudo da relação entre as transformadas de comprimento 8 e 16;

- implementação computacional do método de compressão de imagens baseado em transformadas aproximadas para a DCT de comprimento 16.
- estudo de diferentes algoritmos rápidos para fatoração em matrizes esparsas da matriz da DCT;
- estudo, proposta e implementação de uma nova aproximação para a DCT de comprimento 16;
- estudo e implementação computacional de direfentes métricas de qualidade de imagem e similaridade com a DCT exata.

1.3 Estrutura do texto

O restante deste trabalho está organizado da seguinte forma. A Seção 2.1 apresenta a definição matemática da DCT e sua representação matricial. O padrão de compressão de imagens JPEG é abordado na Seção 2.2. A Seção 2.3 introduz e motiva o estudo sobre algoritmos rápidos. A Seção 2.4 exibe aproximações da DCT com baixo custo computacional encontradas na literatura e a Seção 2.5 descreve rapidamente métodos para comparação de performance destas aproximações. O Capítulo 3 apresenta a metodologia aplicada neste trabalho. O Capítulo 4 mostra através das Seções 4.1, 4.2 e 4.3, respectivamente, a transformada e algoritmo rápido propostos, a avaliação de performance da transformada e sua aplicação em compressão de imagens. Por fim, as conclusões obtidas neste trabalho são apresentadas no Capítulo 5.

2 REVISÃO BIBLIOGRÁFICA

2.1 Transformada Discreta do Cosseno

A DCT é uma ferramenta essencial em processamento digital de sinais e vem sendo extensivamente aplicada na área de codificação de imagens (SALOMON, 2000). A DCT apresenta grande descorrelação do sinal no domínio da transformada, ou seja, ela compacta a maior quantidade de energia do sinal em poucos coeficientes. Dessa forma, o desempenho da DCT fica próximo ao da ótima KLT (RAO; YIP, 1990; BRITANAK; YIP; RAO, 2006). A DCT tem a seguinte formulação matemática. Considere dois vetores N-dimensionais, $\mathbf{v}$ e $\mathbf{V}$, definidos sobre o conjunto dos números reais. De acordo com (SALOMON, 2000, p.290), a DCT relaciona $\mathbf{v}$ e $\mathbf{V}$ da seguinte maneira:

$$V_k = \alpha_k \sum_{n=0}^{N-1} \cos\left(\frac{\pi k(2n+1)}{2N}\right) v_n, \text{ para } k = 0, 1, \ldots, N-1, \tag{2.1}$$

$$v_n = \sum_{k=0}^{N-1} \alpha_k \cos\left(\frac{\pi k(2n+1)}{2N}\right) V_k, \text{ para } n = 0, 1, \ldots, N-1, \tag{2.2}$$

sendo

$$\alpha_k = \begin{cases} 1/\sqrt{N}, & \text{se } k = 0 \\ \sqrt{2/N}, & \text{se } k \neq 0 \end{cases} .$$

As Equações 2.1 e 2.2 representam, respectivamente, as transformações direta e inversa (BAYER; CINTRA, 2010). Além disso, a DCT apresenta uma conveniente representação matricial que pelo fato de apresentar simetrias, diferentemente da KLT, proporciona que eficientes implementações em *software* e *hardware* sejam exploradas. Independentemente da ordem N da matriz de transformação da DCT ($\mathbf{C_N}$), seus (i, j)-ésimos elementos podem ser obtidos por

$$c_{i,j} = \alpha_{i-1} \cos\left(\frac{\pi(i-1)(2(j-1)+1))}{2N}\right), \text{ para } i, j = 1, 2, \ldots N.$$

Além de a matriz de transformação da DCT ser definida para qualquer ordem N, outra importante característica é que esta matriz é ortogonal. Dessa forma, as transformações direta e inversa, definidas em (2.1) e (2.2), podem ser representadas matricialmente, por

$$\mathbf{V} = \mathbf{C_N} \cdot \mathbf{v} \quad \text{e} \tag{2.3}$$

$$\mathbf{v} = \mathbf{C_N^\top} \cdot \mathbf{V}. \tag{2.4}$$

A estrutura matemática da DCT pode ser estendida para o processamento de imagens (BAYER; CINTRA, 2010), pois pode-se interpretar uma imagem $\mathbf{A}$ como um vetor bidimensional de tamanho $N \times N$ em que cada elemento é denotado $a_{m,n}$. A aplicação da DCT bidimensional na imagem $\mathbf{A}$ fornece uma imagem $\mathbf{B}$, cujos elementos são dados por (SALOMON, 2000, p.293):

$$b_{k,l} = \alpha_k \alpha_l \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} \cos\left(\frac{\pi k(2m+1)}{2N}\right) \cos\left(\frac{\pi l(2n+1)}{2N}\right) a_{m,n}, \text{ para } k, l = 1, 2, \ldots N-1.$$

De maneira análoga ao caso unidimensional, a transformação inversa para o caso bidimensional é dada por

$$a_{m,n} = \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} \alpha_k \alpha_l \cos\left(\frac{\pi k(2m+1)}{2N}\right) \cos\left(\frac{\pi l(2n+1)}{2N}\right) b_{k,l}, \text{ para } m, n = 1, 2, \ldots N-1.$$

Na representação matricial, as transformações direta e inversa são dadas por

$$\mathbf{B} = \mathbf{C_N} \cdot \mathbf{A} \cdot \mathbf{C_N^\top} \quad \text{e} \tag{2.5}$$

$$\mathbf{A} = \mathbf{C_N^\top} \cdot \mathbf{B} \cdot \mathbf{C_N}, \tag{2.6}$$

respectivamente.

Contudo, em alguns casos, precisa-se aplicar a DCT com um custo de processamento inferior ao que esta oferece. O custo para a aplicação da DCT de comprimento 8 sobre um vetor unidimensional, como na Equação 2.3, dada a multiplicação matricial, é de 64 multiplicações e 56 adições. Devido ao custo das multiplicações ser maior, se comparado com de adições e deslocamento de bits (BAYER; CINTRA, 2010), são propostas, em diversas pesquisas, algoritmos rápidos e aproximações cuja complexidade multiplicativa é nula ou, ao menos, inferior a da DCT exata.

2.2 Compressão JPEG

Em processamento de imagens, padrões de compressão, como JPEG (WALLACE, 1992), utilizam a aplicação de uma transformada discreta sobre a imagem a ser compactada (PENNEBAKER; MITCHELL, 1992; SALOMON, 2000). Como dito anteriormente, uma imagem pode ser vista como um matriz de ordem $N \times N$. Na representação de uma imagem em escala de cinza com 8 bits, por exemplo, tem-se uma matriz composta por valores inteiros que variam no intervalo $[0, 255]$. No padrão de compressão JPEG é empregada a DCT de comprimento $k = 8$.

Assim, antes da compressão, deve ser feita a quebra da imagem em blocos disjuntos de tamanho 8×8 e, sobre cada um destes, aplica-se a DCT. Após a aplicação da transformada direta, os coeficientes dos blocos estão no domínio das transformadas. A maior parte da energia do sinal fica concentrada nos primeiros coeficientes espectrais – seguindo a sequência *zigzag* ilustrada na Figura 2.1 – de cada bloco de imagem (SALOMON, 2000). Estes blocos são submetidos à etapa de quantização, na qual somente os coeficientes de maior energia são retidos enquanto que os demais são anulados (PENNEBAKER; MITCHELL, 1992). Nestes blocos quantizados é aplicada a transformada discreta do cosseno inversa (IDCT) e, após a recomposição de todos os blocos, tem-se a imagem comprimida similar à original. Tal similaridade vai depender do nível de compressão utilizado na etapa de quantização.

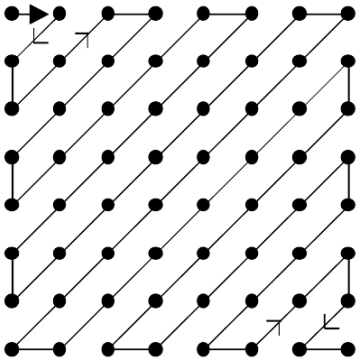


Figura 2.1: Sequência *zigzag* para uma matriz de 64 coeficientes (Adaptada de (INTERNATIONAL TELECOMMUNICATION UNION, 1993)).

A seguir, é exemplificado o procedimento de compressão JPEG, seguindo os seguintes passos: aplicação da transformada direta, quantização e aplicação da transformada inversa sobre o primeiro bloco 8×8 da imagem *boat* (THE USC-SIPI IMAGE DATABASE, 2011) em escala de cinza de 8 *bits*. A matriz $\mathbf{A_8}$ contém os elementos deste bloco:

$$\mathbf{A_8} = \begin{bmatrix} 127 & 123 & 125 & 120 & 126 & 123 & 127 & 128 \\ 142 & 135 & 144 & 143 & 140 & 145 & 142 & 140 \\ 128 & 126 & 128 & 122 & 125 & 125 & 122 & 129 \\ 132 & 144 & 144 & 139 & 140 & 149 & 140 & 142 \\ 128 & 124 & 128 & 126 & 127 & 120 & 128 & 129 \\ 133 & 142 & 141 & 141 & 143 & 140 & 146 & 138 \\ 124 & 127 & 128 & 129 & 121 & 128 & 129 & 128 \\ 134 & 143 & 140 & 139 & 136 & 140 & 138 & 141 \end{bmatrix}.$$

O processo de aplicação da transformada direta sobre o bloco $\mathbf{A_8}$ é dada por

$$\mathbf{B_8} = (\mathbf{C_8} \cdot \mathbf{A_8} \cdot \mathbf{C_8}^{\top}),$$

em que $\mathbf{C_8}$ é matriz de transformação da DCT. O processo de transformação neste ponto é dito sem perda de dados – ou seja, se a transformada inversa for aplicada, teremos como resultado a

mesma matriz $\mathbf{A_8}$ (exceto por eventuais erros de arredondamento) (SALOMON, 2000). Neste exemplo, a matriz resultante é

$$\mathbf{B_8} \simeq \begin{bmatrix} 1065{,}50 & -5{,}04 & 0{,}17 & -5{,}18 & -5{,}50 & -0{,}84 & 2{,}37 & 0{,}86 \\ -9{,}77 & 0{,}63 & 2{,}27 & 4{,}65 & 4{,}25 & 0{,}46 & 5{,}86 & 4{,}78 \\ -3{,}82 & 0{,}39 & 2{,}68 & 0{,}99 & 0{,}98 & 2{,}16 & 0{,}56 & -1{,}30 \\ -11{,}65 & -1{,}99 & -0{,}07 & 0{,}96 & 2{,}14 & 4{,}49 & -6{,}03 & 5{,}68 \\ -3{,}00 & -0{,}62 & 2{,}17 & -1{,}23 & 0{,}00 & -4{,}36 & -0{,}63 & 0{,}84 \\ -17{,}75 & -1{,}75 & 1{,}06 & -0{,}24 & -4{,}60 & -3{,}74 & 0{,}66 & -3{,}24 \\ -4{,}64 & 1{,}03 & 1{,}56 & 0{,}11 & 0{,}41 & -5{,}43 & -3{,}68 & 3{,}27 \\ -53{,}65 & 4{,}56 & 8{,}17 & -1{,}71 & 7{,}99 & 2{,}22 & 0{,}78 & 3{,}66 \end{bmatrix}.$$

Percebe-se que a matriz de coeficientes $\mathbf{B_8}$ concentra a maior parte da energia do sinal em seus primeiros coeficientes espectrais de acordo com a sequência *zigzag* (SALOMON, 2000). Os demais coeficientes são muito próximos de zero e podem ser descartados, ou anulados, com pouca perda de qualidade do sinal reconstruído.

Na etapa de quantização, cada elemento da matriz $\mathbf{B_8}$ é dividido pelo elemento de mesma posição de uma matriz de quantização $\mathbf{Q_8}$. Há diversas matrizes de quantização, paramétricas ou não, sugeridas no padrão JPEG (SALOMON, 2000). Utilizaremos, neste exemplo, a matriz de quantização paramétrica dada por

$$\mathbf{Q}_{i,j} = 1 + (i + j) \cdot R, \tag{2.7}$$

onde, neste caso, i e j são inteiros entre 0 e 7, inclusive. O parâmetro R é informado pelo usuário e é diretamente proporcional ao nível de compressão da imagem. Este parâmetro pode assumir qualquer valor inteiro positivo. Neste exemplo, optou-se por utilizar $R = 10$ e, dessa forma, tem-se que

$$\mathbf{Q_8} = \begin{bmatrix} 1 & 11 & 21 & 31 & 41 & 51 & 61 & 71 \\ 11 & 21 & 31 & 41 & 51 & 61 & 71 & 81 \\ 21 & 31 & 41 & 51 & 61 & 71 & 81 & 91 \\ 31 & 41 & 51 & 61 & 71 & 81 & 91 & 101 \\ 41 & 51 & 61 & 71 & 81 & 91 & 101 & 111 \\ 51 & 61 & 71 & 81 & 91 & 101 & 111 & 121 \\ 61 & 71 & 81 & 91 & 101 & 111 & 121 & 131 \\ 71 & 81 & 91 & 101 & 111 & 121 & 131 & 141 \end{bmatrix}.$$

Dada a matriz $\mathbf{Q_8}$, pode-se aplicar a quantização sobre o bloco $\mathbf{B_8}$ por meio da seguinte operação (BHASKARAN; KONSTANTINIDES, 1997, pág. 82):

$$\mathbf{B}'_{\mathbf{8}} = \mathbf{Q_8} \circ \mathrm{round}(\mathbf{B_8} \oslash \mathbf{Q_8}),$$

em que $\circ$ e $\oslash$ são, respectivamente, a multiplicação e a divisão elemento a elemento (SEBER, 2008) e $\mathrm{round}(\,\cdot\,)$ é a função de arredondamento para inteiro. Neste ponto ocorre, de fato, a compressão, visto que valores próximos de zero são anulados. O bloco quantizado $\mathbf{B}'_{\mathbf{8}}$, neste exemplo, é

$$\mathbf{B}'_{\mathbf{8}} = \begin{bmatrix} 1065 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -71 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

O bloco quantizado $\mathbf{B}'_{\mathbf{8}}$ tem apenas 3 de 64 coeficientes não nulos, o que provoca uma compressão de aproximadamente 95%. Para obtermos o bloco $\mathbf{A_8}$ comprimido (que nomeamos $\mathbf{A}'_{\mathbf{8}}$), devemos aplicar a transformada inversa sobre o bloco $\mathbf{B}'_{\mathbf{8}}$. Este processo é dado por

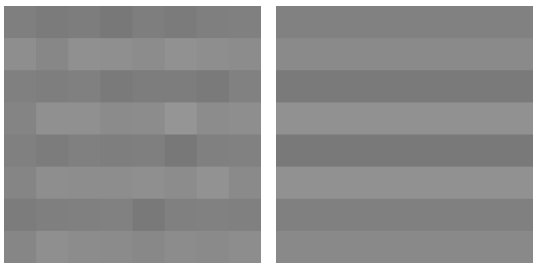
$$\mathbf{A}'_{\mathbf{8}} = \mathrm{round}({\mathbf{C_8}}^{\top} \cdot \mathbf{B}'_{\mathbf{8}} \cdot \mathbf{C_8}).$$

A função $\mathrm{round}(\,\cdot\,)$ se faz necessária pois a matriz $\mathbf{A}'_{\mathbf{8}}$ deve ter apenas valores inteiros para que possa ser representada em uma imagem em escala de cinza de 8 *bits*. Para este exemplo, tem-se

$$\mathbf{A}'_{\mathbf{8}} = \begin{bmatrix} 129 & 129 & 129 & 129 & 129 & 129 & 129 & 129 \\ 138 & 138 & 138 & 138 & 138 & 138 & 138 & 138 \\ 122 & 122 & 122 & 122 & 122 & 122 & 122 & 122 \\ 145 & 145 & 145 & 145 & 145 & 145 & 145 & 145 \\ 121 & 121 & 121 & 121 & 121 & 121 & 121 & 121 \\ 145 & 145 & 145 & 145 & 145 & 145 & 145 & 145 \\ 128 & 128 & 128 & 128 & 128 & 128 & 128 & 128 \\ 137 & 137 & 137 & 137 & 137 & 137 & 137 & 137 \end{bmatrix}.$$

Note que as matrizes $\mathbf{A_8}$ e $\mathbf{A}'_{\mathbf{8}}$ são bastante semelhantes apesar da alta taxa de compressão (aproximadamente 95%) aplicada.

O processo descrito deve ser aplicado a todos os blocos $\mathbf{A_8}$ da imagem e, após, deve-se recompor a mesma, realocando cada bloco na sua posição correspondente na imagem original. A imagem resultante é dita comprimida ou compactada com perdas. As matrizes $\mathbf{A_8}$ e $\mathbf{A}'_{\mathbf{8}}$ utilizadas neste exemplo são apresentadas na forma de imagem em escala de cinza nas Figuras 2.2a e 2.2b, respectivamente. As imagens original e comprimida pelo processo descrito são apresentadas nas Figuras 2.3a e 2.3b. É importante destacar que este processo de compressão tipo JPEG é válido para outras ordens de bloco como $4, 16$ e 32 e que outras matrizes de quantização que não a mostrada na Equação 2.7 podem ser usadas.



(a) (b)

Figura 2.2: Blocos 8×8 (a) original e (b) comprimido da imagem *boat* em escala de cinza de 8 bits.

2.3 Algoritmos rápidos

Algoritmos rápidos visam reduzir o custo computacional – número de operações aritméticas – exigidos na aplicação de uma transformada discreta. Como visto na Seção 2.1, para

(a) (b)

Figura 2.3: Imagem *boat* em escala de cinza de 8 bits (a) original e (b) comprimida.

aplicar a DCT de comprimento 8 sobre um vetor unidimensional sem o uso de nenhum algoritmo rápido, são necessárias 64 multiplicações e 56 adições. Em contrapartida, com o uso do algoritmo rápido de Chen (CHEN; SMITH; FRALICK, 1977), por exemplo, tem-se apenas 16 multiplicações e 26 adições.

Segundo (BLAHUT, 2010), qualquer algoritmo deve, dado um conjunto de entradas, retornar uma saída. O que é de interesse de um algoritmo rápido é o quão eficiente será o processamento destas entradas. Para que o algoritmo seja eficiente, deve-se resolver determinada tarefa com o mínimo de cálculos necessários. Um exemplo simples dado em (BLAHUT, 2010) é o cálculo da seguinte expressão

$$A = ac + ad + bc + bd. \tag{2.8}$$

O cálculo de A, como foi apresentado na Equação 2.8, requer três adições e quatro multiplicações. Uma maneira mais eficiente de obter-se o mesmo resultado é dada por

$$A = a(c + d) + b(c + d), \tag{2.9}$$

na qual são requeridas três adições e duas multiplicações. Ainda, se considerarmos a, b, c e d valores conhecidos, podemos calcular A por

$$A = (a + b)(c + d), \tag{2.10}$$

com apenas duas adições e uma multiplicação. Pode-se ver que as Equações 2.8, 2.9 e 2.10 resultam no mesmo valor A e que pode-se, até certo limite, rearranjar adições e multiplicações de forma a reduzir o custo envolvido neste cálculo.

No que tange aplicações em processamento de sinais, os algoritmos rápidos se utilizam muito do reaproveitamento de cálculos. Tal processo pode ser comparado com técnicas de programação dinâmica onde, à medida que cálculos são realizados, armazenam-se os resultados em tabelas ou matrizes. Quando necessário, uma simples busca sobre os valores desejados é feita. Uma bibliografia completa a respeito de algoritmos rápidos para o processamento de sinais pode ser encontrada em (BLAHUT, 2010).

Muitos algoritmos rápidos para o cálculo da DCT já foram propostos desde a década de 70, por exemplo: os algoritmos de Chen (CHEN; SMITH; FRALICK, 1977), de Arai (ARAI; AGUI; NAKAJIMA, 1988), de Wang (WANG, 1984), de Lee (LEE, 1984), de Loeffler (LOEFFLER; LIGTENBERG; MOSCHYTZ, 1989) e de Feig e Winograd (FEIG; WINOGRAD, 1992). Tais algoritmos propõe cálculos computacionalmente eficientes para a DCT, mas ainda assim exigem multiplicações não triviais. Entretanto, diversas aplicações em processamento de imagens precisam aplicar a transformada em um menor tempo. Para atingir tal objetivo, têm-se concentrado esforços para o cálculo aproximado da DCT, principalmente focados na proposição de algoritmos livres de multiplicação. Uma breve contextualização sobre tais aproximações da DCT é feita na seção seguinte.

2.4 Transformadas aproximadas

As pesquisas para proposição de algoritmos rápidos para a DCT exata já são bastante maduras e apresentam algoritmos que alcançam o limite inferior teórico de complexidade computacional ou muito próximo disso. É improvável que novos algoritmos mais rápidos do que os já existentes possam ser desenvolvidos. Neste sentido, a comunidade científica têm voltado seus esforços para o desenvolvimento de aproximações da DCT. Estes algoritmos não calculam exatamente a DCT, mas são bons estimadores espectrais e possuem complexidade computacional inferior aos algoritmos exatos.

Neste sentido, e, principalmente focados em implementações eficientes em *hardware*, diversas aproximações da DCT têm sido propostas. Em especial, aproximações para a transformada de comprimento $k = 8$ tomaram destaque pela possível aplicabilidade em padrões conhecidos como JPEG e MPEG-1. Aproximações matematicamente simples, como a SDCT (HAWEEL, 2001), alavancaram pesquisas sobre métodos para aproximar transformadas através de arredondamento por inteiros, fatoração em matrizes esparsas, entre outras técnicas. Como resultado deste tipo de pesquisa, pode-se citar a aproximação *level* 1 (LENGWEHASATIT; OR-

TEGA, 2004), as aproximações (CINTRA; BAYER, 2011), (BAYER; CINTRA, 2012), (POTLURI et al., 2012), (POTLURI et al., 2014), (CINTRA; BAYER; TABLADA, 2014) e a série de aproximações propostas por BAS (BOUGUEZEL; AHMAD; SWAMY, 2008, 2009, 2010, 2011, 2013). Além das aproximações para a transformada de comprimento 8, nos últimos anos, verificou-se o interesse por transformadas aproximadas de outros comprimentos, como transformadas de comprimento $k = 16$ em (BAYER et al., 2012), (BOUGUEZEL; AHMAD; SWAMY, 2010) e (BOUGUEZEL; AHMAD; SWAMY, 2013). O desenvolvimento dessas transformadas pode ser motivado, em grande parte, pela possível aplicabilidade no recente *codec* HEVC (POURAZAD et al., 2012), que utiliza transformadas de comprimentos 4, 8, 16 e 32.

As pesquisas sobre aproximações da DCT de comprimento 16, da mesma forma que as de comprimento 8, podem ter sido impulsionadas pela aplicação da DCT destes comprimentos em *codecs* de vídeo, como H.264/MPEG-4 (LUTHRA; SULLIVAN; WIEGAND, 2003) e HEVC. Dentre as transformadas de comprimento 16 de complexidade multiplicativa nula, pode-se citar a transformada de Walsh-Hadamard (WHT) (SALOMON, 2011), as aproximações propostas por BAS (BOUGUEZEL; AHMAD; SWAMY, 2010) e (BOUGUEZEL; AHMAD; SWAMY, 2013) e a aproximação proposta por Bayer *et al* (BAYER et al., 2012). Estas quatro aproximações da DCT são utilizadas para comparação com a transformada proposta neste trabalho.

A seguir, são mostradas explicitamente as matrizes de transformação das aproximações WHT, BAS-2010, BAS-2013 e Bayer-2012, respectivamente.

$$\hat{\mathbf{C}}_{\text{WHT}} = \frac{1}{4}\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1\\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1\\
1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1\\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1\\
1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1\\
1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1\\
1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1\\
1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1\\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1\\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1\\
1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1\\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1\\
1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1\\
1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & 1 & -1 & 1 & -1\\
1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 & -1 & 1 & 1 & 1 & -1 & -1\\
1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1
\end{bmatrix},$$

$$\hat{\mathbf{C}}_{\text{BAS-2010}} = \mathbf{D_0} \cdot \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1\\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1\\
2 & 1 & -1 & -2 & -2 & -1 & 1 & 2 & 2 & 1 & -1 & -2 & -2 & -1 & 1 & 2\\
2 & 2 & 1 & 1 & -1 & -1 & -2 & -2 & 2 & 2 & 1 & 1 & -1 & -1 & -2 & -2\\
2 & 1 & -1 & -2 & 2 & 1 & -1 & -2 & -2 & -1 & 1 & 2 & -2 & -1 & 1 & 2\\
2 & -2 & -1 & 1 & -1 & 1 & 2 & -2 & 2 & -2 & -1 & 1 & -1 & 1 & 2 & -2\\
1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1\\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1\\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1\\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1\\
1 & -1 & 2 & -2 & 2 & -2 & 1 & -1 & 1 & -1 & 2 & -2 & 2 & -2 & 1 & -1\\
1 & 1 & -2 & -2 & 2 & 2 & -1 & -1 & 1 & 1 & -2 & -2 & 2 & 2 & -1 & -1\\
1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1\\
1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & 1 & -1 & 1 & -1\\
1 & -2 & 2 & -1 & 1 & -2 & 2 & -1 & -1 & 2 & -2 & 1 & -1 & 2 & -2 & 1\\
1 & -2 & 2 & -1 & -1 & 2 & -2 & 1 & 1 & -2 & 2 & -1 & -1 & 2 & -2 & 1
\end{bmatrix},$$

$$\hat{\mathbf{C}}_{\text{BAS-2013}} = \frac{1}{4}\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 \\
1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \\
1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\
1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\
1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 \\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\
1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 \\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 \\
1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 \\
1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \\
1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 & 1 & -1 & 1 & -1 & 1 & -1 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1
\end{bmatrix} \text{ e}$$

$$\hat{\mathbf{C}}_{\text{Bayer-2012}} = \mathbf{D_1} \cdot \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & -1 & -1 & 0 & -1 & -1 & -1 & -1 & -1 \\
1 & 1 & 1 & 0 & 0 & -1 & -1 & -1 & -1 & -1 & -1 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & 0 & -1 & -1 & -1 \\
1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\
1 & 1 & -1 & -1 & -1 & 1 & 1 & 0 & 0 & -1 & -1 & 1 & 1 & 1 & -1 & -1 \\
1 & 0 & -1 & -1 & 1 & 1 & 0 & -1 & -1 & 0 & 1 & 1 & -1 & -1 & 0 & 1 \\
1 & 0 & -1 & 1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & -1 & 1 & 0 & -1 \\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\
1 & -1 & -1 & 1 & -1 & -1 & 0 & 1 & -1 & 0 & 1 & 1 & -1 & 1 & 1 & -1 \\
1 & -1 & 0 & 1 & -1 & 0 & 1 & -1 & -1 & 1 & 0 & -1 & 1 & 0 & -1 & 1 \\
0 & -1 & 1 & 1 & -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & -1 & 1 & 0 \\
1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\
1 & -1 & 1 & -1 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 1 & -1 & 1 & -1 \\
0 & -1 & 1 & -1 & 1 & -1 & 1 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 0 \\
1 & -1 & 0 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & 0 & 1 & -1
\end{bmatrix},$$

sendo que

$$\mathbf{D_0} = \text{diag}\left(\frac{1}{4}, \frac{1}{4}, \frac{1}{\sqrt{40}}, \frac{1}{\sqrt{40}}, \frac{1}{\sqrt{40}}, \frac{1}{\sqrt{40}}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{\sqrt{40}}, \frac{1}{\sqrt{40}}, \frac{1}{4}, \frac{1}{4}, \frac{1}{\sqrt{40}}, \frac{1}{\sqrt{40}}\right) \text{ e}$$
$$\mathbf{D_1} = \text{diag}\left(\frac{1}{4}, \frac{1}{\sqrt{14}}, \frac{1}{2\sqrt{3}}, \frac{1}{\sqrt{14}}, \frac{1}{4}, \frac{1}{\sqrt{14}}, \frac{1}{2\sqrt{3}}, \frac{1}{\sqrt{14}}, \frac{1}{4}, \frac{1}{\sqrt{14}}, \frac{1}{2\sqrt{3}}, \frac{1}{\sqrt{14}}, \frac{1}{4}, \frac{1}{\sqrt{14}}, \frac{1}{2\sqrt{3}}, \frac{1}{\sqrt{14}}\right).$$

Como todas essas matrizes de transformação são ortogonais, assim como a DCT exata, podemos definir a aplicação das transformações direta e inversa de forma similar às Equações 2.5 e 2.6, respectivamente. Dessa forma, a aplicação das transformações direta e inversa, respectivamente, das aproximações da DCT são dadas por

$$\mathbf{B}_k = \hat{\mathbf{C}}_k^\top \cdot \mathbf{A}_k \cdot \hat{\mathbf{C}}_k \text{ e} \tag{2.11}$$

$$\mathbf{A}_k = \hat{\mathbf{C}}_k \cdot \mathbf{B}_k \cdot \hat{\mathbf{C}}_k^\top, \tag{2.12}$$

sendo $k = 16$ a ordem das matrizes, $\mathbf{A}_k$ é um bloco da imagem original, $\mathbf{B}_k$ é um bloco no domínio da transformada e $\hat{\mathbf{C}}_k$ é uma das transformadas citadas acima. Além destas transformadas aproximadas da DCT, poderíamos destacar outras aproximações como a clássica SDCT que, assim como a DCT, tem sua formulação genérica para qualquer comprimento k. Contudo, a SDCT é não-ortogonal e sua inversa possui alto custo aritmético. É importante salientar que podemos considerar as matrizes diagonais de ajuste, $\mathbf{D_0}$ e $\mathbf{D_1}$, apenas na etapa de quantização, de modo que estas não influenciam no custo associado à transformada em questão (CINTRA; BAYER, 2011; BOUGUEZEL; AHMAD; SWAMY, 2011).

2.5 Medidas de avaliação e qualidade de imagem

Geralmente, as aproximações da DCT são propostas visando um menor custo computacional e, ao mesmo tempo, uma maior qualidade em relação às já existentes na literatura. Entretanto, raramente é possível que uma mesma transformada atinja estas duas metas de forma simultânea e, dessa forma, cabe ao engenheiro ou desenvolvedor, avaliar qual a mais adequada para sua aplicação. Por este motivo, torna-se importante a definição de medidas de desempenho das transformadas aproximadas. Essas medidas dizem respeito à complexidade computacional da transformada, à similaridade com a DCT exata, ao ganho de codificação e, em compressão de imagens, à qualidade das imagens comprimidas.

2.5.1 Complexidade de transformadas

As complexidades computacionais das transformadas são tradicionalmente medidas pela quantidade de operações aritméticas envolvidas em sua aplicação em *hardware* (BRIGGS; HENSON, 1995). O número de multiplicações é tomado como medida principal visto que esta é a operação de maior custo, seguido de adições e deslocamentos de *bits* (BAYER; CINTRA, 2010). Entretanto, algumas multiplicações não exigem um algoritmo completo de multiplicação como, por exemplo, multiplicações por zero, por uma unidade ou troca de sinal (multiplicação por -1). A multiplicação destes valores é trivial e, de fato, não ocorre em *hardware*. Ademais, multiplicações por potências de dois podem ser implementadas de forma simples em aritmética de base dois por meio de deslocamento de *bits*. Estas operações são denominadas multiplicações triviais e, classicamente, na avaliação da complexidade multiplicativa de algoritmos numéricos, não são contabilizadas (BRIGGS; HENSON, 1995).

Desta forma, busca-se, ao propor uma nova aproximação de uma transformada, reduzir ao máximo a complexidade multiplicativa ao mesmo tempo que tenta-se manter sua similaridade à DCT exata. Neste trabalho, serão abordadas aproximações da DCT cuja complexidade multiplicativa é nula e, assim, avalia-se a complexidade aritmética através do número de operações de deslocamentos de *bits* e de adições (CINTRA; BAYER, 2011; SOUZA, 2011).

2.5.2 Medidas de similaridade com a DCT e ganho de codificação

Deseja-se que as aproximações da DCT tenham suas matrizes de transformação próximas da matriz da DCT exata, ao mesmo tempo que as operações aritméticas associadas a elas

sejam mínimas. Desta forma, é importante mensurar o quão próxima a transformada proposta está da DCT. Isto pode ser calculado através de algumas medidas de similaridade como distorção da DCT (FONG; CHAM, 2012), *Total Error Energy* (CINTRA; BAYER, 2011) e erro quadrático médio (MSE) (BRITANAK; YIP; RAO, 2007). A habilidade de descorrelação do sinal no domínio das transformadas pode ser medida através métricas como *Transform Coding Gain* (JAYANT; NOLL, 1984) e *Transform Efficiency* (TAKALA; NIKARA, 2001). As expressões das métricas recém citadas são dadas a seguir:

2.5.2.1 Distorção da DCT

Uma forma de medir a diferença entre uma transformada aproximada e a DCT é através da distorção da DCT. Nesta medida, a transformada cuja distorção é mínima é aquela que tem maior similaridade com a DCT exata (FONG; CHAM, 2012). Pode-se calcular a distorção de uma transformada aproximada $\hat{\mathbf{C}}$ de comprimento N por meio da seguinte medida:

$$d_2(\hat{\mathbf{C}}) = 1 - \frac{1}{N} \| \operatorname{diag}\left(\mathbf{C} \cdot \hat{\mathbf{C}}^{\top}\right) \|^2, \tag{2.13}$$

em que $\mathbf{C}$ é a matriz de transformação da DCT exata e $\| \cdot \|$ é a norma euclideana.

2.5.2.2 *Total Error Energy*

A figura de mérito *Total Error Energy* mensura a proximidade espectral entre uma aproximação da DCT ($\hat{\mathbf{C}}$) e a DCT exata ($\mathbf{C}$) (CINTRA; BAYER, 2011) e é dada por

$$\in (\hat{\mathbf{C}}) = \int_0^{\pi} D_m\left(\omega; \hat{\mathbf{C}}\right) d\omega,$$

em que

$$D_m\left(\omega; \hat{\mathbf{C}}\right) \triangleq \left| H_m\left(\omega; \mathbf{C}\right) - H_m\left(\omega; \hat{\mathbf{C}}\right) \right|^2 \text{ e } m = 0, 1, ..., N-1,$$

e $H_m(\omega; \mathbf{C})$ e $H_m(\omega; \hat{\mathbf{C}})$ são as funções de transferência da DCT exata e da aproximação, respectivamente. A função de transferência de uma transformada $\mathbf{T}$ é dada por

$$H_m\left(\omega; \mathbf{T}\right) = \sum_{n=0}^{N-1} t_{m,n} \exp\left(-jn\omega\right), m = 0, 1, ..., N-1,$$

em que N é ordem da matriz, $j = \sqrt{-1}$, $\omega \in [0, \pi]$ e $t_{m,n}$ é o $(m+1, n+1)$-ésimo elemento de $\mathbf{T}$.

Contudo, via teorema de Parseval (JAIN, 1989, pág. 17), pode-se calcular o *Total Error Energy* diretamente por (TABLADA; BAYER; CINTRA, 2013)

$$\in (\hat{\mathbf{C}}) = \pi \cdot \left\| \mathbf{C} - \hat{\mathbf{C}} \right\|^2 . \tag{2.14}$$

2.5.2.3 Erro quadrático médio

O erro quadrático médio ou *mean square error* (MSE) entre a DCT ($\mathbf{C}$) e uma transformada aproximada $\hat{\mathbf{C}}$ é dado por

$$\mathrm{MSE}(\hat{\mathbf{C}}) = \frac{1}{N} \cdot \mathrm{tr}\left((\mathbf{C} - \hat{\mathbf{C}}) \cdot \mathbf{R_x} \cdot (\mathbf{C} - \hat{\mathbf{C}})^\top \right), \tag{2.15}$$

em que $\mathbf{R_x}$ é a matriz de covariância dada por

$$\mathbf{R_{x(i,j)}} = \rho^{|i-j|} \text{ com } \rho = 0,95 \tag{2.16}$$

e $\mathrm{tr}(\,\cdot\,)$ é a função traço – dada pela soma dos elementos da diagonal principal da matriz argumento. Para que haja maior correspondência entre $\hat{\mathbf{C}}$ e $\mathbf{C}$, o valor obtido por MSE deve ser o menor possível (BRITANAK; YIP; RAO, 2007).

2.5.2.4 *Transform Coding Gain*

Esta medida, segundo (FONG; CHAM, 2012), é um método simples para medir o desempenho de codificação de uma transformada $\mathbf{T}$ de comprimento N. Tomando r_{ij} sendo o (i,j)-ésimo elemento de $(\mathbf{T} \cdot \mathbf{R_x} \cdot \mathbf{T}^\top)$, em que $\mathbf{R_x}$ é definido na Equação 2.16, pode-se calcular o *Transform Coding Gain* através de

$$C_g(\mathbf{T}) = 10 \log_{10} \frac{\frac{1}{N} \sum_{i=0}^{N-1} r_{ii}}{\left(\prod_{i=0}^{N-1} \left(r_{ii} \cdot \sqrt{\sum_{j=0}^{N-1} r_{ij}^2} \right) \right)^{\frac{1}{N}}}. \tag{2.17}$$

2.5.2.5 *Transform Efficiency*

Segundo (BRITANAK; YIP; RAO, 2007), uma medida alternativa para determinar o ganho de codificação de uma transformada $\mathbf{T}$ de comprimento N é a *Transform Efficiency*, definida como

$$\eta(\mathbf{T}) = \frac{\sum_{i=0}^{N-1} |r_{ii}|}{\sum_{i=0}^{N-1} \sum_{j=0}^{N-1} |r_{ij}|} 100, \tag{2.18}$$

em que r_{ij} é dado na Seção 2.5.2.4. A medida *Transform Efficiency*, assim como *Transform Coding Gain*, indica a habilidade de descorrelacionar os coeficientes após a aplicação de uma transformada. A KLT converte o sinal de forma completamente descorrelata e apresenta o valor máximo, $\eta = 100$. A DCT de comprimento $k = 8$, por outro lado, apresenta $\eta = 93,9911$ (BRITANAK; YIP; RAO, 2007).

2.5.3 Medidas de qualidade de imagens

No processamento digital de imagens, diversas medidas são consideradas para determinar, quantitativamente, a qualidade de uma imagem comprimida, dada a imagem original como referência. O erro quadrático médio (MSE) entre duas imagens (BRITANAK; YIP; RAO, 2007) é uma figura de mérito amplamente aplicada como medida de qualidade de uma imagem comprimida ou ruidosa. Entretanto, esta medida pode apresentar resultados ruins se operações simples, como a translação da imagem em um *pixel*, forem feitas. A Relação Pico Sinal-Ruído (PSNR) é sugerida por BAS (BOUGUEZEL; AHMAD; SWAMY, 2008) como figura de mérito para avaliação quantitativa da qualidade de uma imagem comprimida e, de fato, vem sendo aplicada em diversos trabalhos. As formulações matemáticas do MSE e do PSNR são, respectivamente,

$$\mathrm{MSE} = \frac{\sum_{i=1}^{N}\sum_{j=1}^{N}(\mathbf{A_{i,j}} - \mathbf{B_{i,j}})^2}{N^2} \quad \text{e} \tag{2.19}$$

$$\mathrm{PSNR} = 10 \cdot \log_{10}\left(\frac{\mathrm{MAX}^2}{\mathrm{MSE}}\right), \tag{2.20}$$

sendo $\mathbf{A}$ a imagem original, $\mathbf{B}$ a imagem reconstituída através de uma transformada e N é a dimensão de ambas. A variável MAX é o valor máximo possível para cada um dos elementos de $\mathbf{A}$ e $\mathbf{B}$. Em se tratando de imagens em escala de cinza com 8 bits, $\mathrm{MAX} = 255$, pois os valores, nesta representação, são inteiros que variam no intervalo [0,255].

Uma proposta alternativa para avaliação da qualidade de imagens, dada uma imagem de referência, é o índice de similaridade estrutural (MSSIM) (WANG et al., 2004) que, diferentemente do MSE e PSNR, considera o funcionamento da visão humana para determinar sua formalização matemática (WANG et al., 2004). Considerando ainda $\mathbf{A}$ e $\mathbf{B}$ as matrizes mencionadas anteriormente, pode-se calcular o MSSIM por

$$\mathrm{MSSIM}(\mathbf{A}, \mathbf{B}) = \frac{1}{M}\sum_{j=1}^{M}\mathrm{SSIM}(a_j, b_j), \tag{2.21}$$

considerando a_j e b_j, respectivamente, os j-ésimos blocos de $\mathbf{A}$ e $\mathbf{B}$ e M, o total de blocos. O cálculo de cada bloco, ou seja, o SSIM é dado por

$$\mathrm{SSIM}(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)},$$

sendo x e y blocos de $\mathbf{A}$ e $\mathbf{B}$, respectivamente. $C_1 = 6,5025$ e $C_2 = 58,5225$ (WANG et al., 2004), μ_x é a média aritmética de x (Equação 2.22), μ_y é a média aritmética de y (Equação 2.23), σ_x é o desvio padrão de x (Equação 2.24), σ_y é o desvio padrão de y (Equação 2.25) e σ_{xy} é a covariância entre x e y (Equação 2.26). Tais medidas estatísticas têm as formulações dadas, respectivamente, por

$$\mu_x = \frac{1}{N}\sum_{i=1}^{N} x_i, \tag{2.22}$$

$$\mu_y = \frac{1}{N}\sum_{i=1}^{N} y_i, \tag{2.23}$$

$$\sigma_x = \left(\frac{1}{N-1}\sum_{i=1}^{N} (x_i - \mu_x)^2\right)^{\frac{1}{2}}, \tag{2.24}$$

$$\sigma_y = \left(\frac{1}{N-1}\sum_{i=1}^{N} (y_i - \mu_y)^2\right)^{\frac{1}{2}} \text{ e} \tag{2.25}$$

$$\sigma_{xy} = \frac{1}{N-1}\sum_{i=1}^{N} (x_i - \mu_x)(y_i - \mu_y), \tag{2.26}$$

com N sendo o número de elementos de x e y.

3 METODOLOGIA

Neste trabalho, os métodos abordados para propor uma nova transformação aproximada de comprimento 16 são: a aproximação por inteiros, exploração de algoritmos rápidos clássicos para a DCT e das relações entre a DCT de comprimento 8 e 16 como em (BAYER et al., 2012).

A aproximação por inteiros é um dos artifícios utilizados para propor aproximações para a DCT com baixo custo computacional (CINTRA; BAYER; TABLADA, 2014) como visto na Seção 2.4. Este artifício pode ser encontrado na literatura, primeiramente, em (HAWEEL, 2001) com a apresentação da transformada não-ortogonal SDCT. Além disso, outros trabalhos, como (CINTRA; BAYER, 2011), também exploram, de maneira simples, esta técnica para a proposição de uma nova transformada. A transformada apresentada em (CINTRA; BAYER, 2011), assim como em (BAYER; CINTRA, 2012; BOUGUEZEL; AHMAD; SWAMY, 2008, 2009, 2010; BAYER et al., 2012) , vale-se do fato de que uma matriz diagonal de ajuste pode ser inserida na etapa de quantização e, dessa forma, seu custo computacional pode ser desconsiderado em compressão de imagens do tipo JPEG e de vídeo como HEVC (POTLURI et al., 2014). Neste trabalho, assim como em (BOUGUEZEL; AHMAD; SWAMY, 2010) e (BAYER et al., 2012), apresenta-se não apenas a matriz de transformação da transformada proposta como também um algoritmo rápido construído através da fatoração em matrizes esparsas para o seu cálculo. Com base nestes métodos para proposição de aproximações da DCT e redução de custo computacional, é apresentada uma nova aproximação da DCT de comprimento 16 ortogonal e com complexidade multiplicativa nula.

Para avaliar o desempenho da transformada proposta, utilizaram-se imagens retiradas do Banco de Imagens Público (THE USC-SIPI IMAGE DATABASE, 2011) – importante referência internacional. Todas as imagens armazenadas neste banco encontram-se em tamanhos 256×256 e 512×512 e, portanto, vão ao encontro da condição abordada neste trabalho, ou seja, a dimensão das imagens ser $N \times N$. A compressão de imagens apresentada na Seção 4.3 é realizada de forma similar à do padrão JPEG introduzida na Seção 2.2. Na etapa de quantização aplicada neste trabalho, conservam-se os r primeiros coeficientes – de acordo com a sequência *zig-zag* – e anulam-se os demais, diferentemente do padrão JPEG que divide cada bloco quantizado, elemento a elemento, por uma matriz de quantização $\mathbf{Q}$. Esta alternativa à quantização do padrão JPEG é utilizada em diversos trabalhos como (CINTRA; BAYER, 2011; BAYER et al., 2012; BOUGUEZEL; AHMAD; SWAMY, 2010, 2011, 2013). O pseudo-algoritmo para

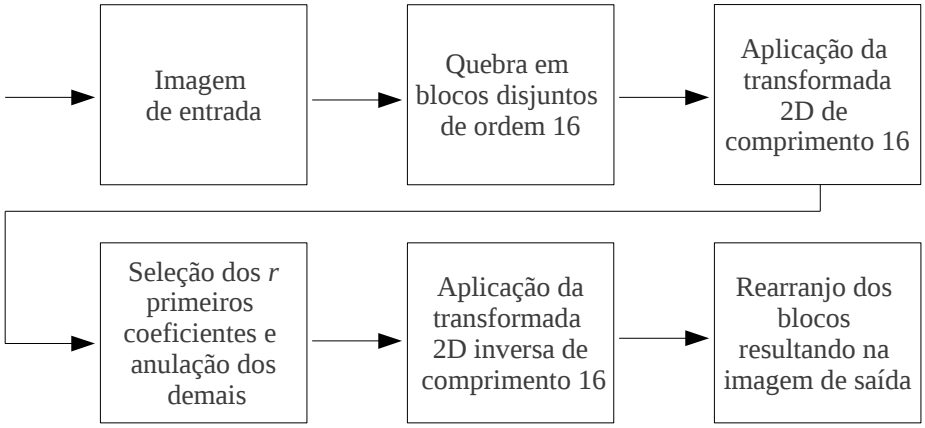


Figura 3.1: Pseudo-algoritmo para compressão de imagens.

o processo de compressão utilizado neste trabalho é ilustrado, sob a forma de diagrama, na Figura 3.1.

Os resultados comparativos entre a transformada proposta e as demais transformadas de comprimento 16 estudadas neste trabalho foram obtidos através da implementação das diversas métricas de similaridade com a DCT exata (Equações 2.13, 2.14 e 2.15) e ganho de codificação (Equações 2.17 e 2.18). As métricas de qualidade de imagens (Equações 2.19, 2.20 e 2.21) foram implementas visando a avaliação das transformadas em compressão de imagens do tipo JPEG. Além disso, assim como em outros trabalhos, o número de operações aritméticas para o cálculo da transformada é tomado como medida de eficiência computacional.

As formas de avaliação, assim como a manipulação das imagens, foram implementadas utilizando a linguagem Java. Para geração de gráficos, optou-se por utilizar a linguagem R (R Core Team, 2013). A Figura 3.2 ilustra um diagrama de classes simplificado onde são mostrados os principais métodos públicos das mais relevantes classes Java implementadas. Os códigos fonte necessários para reproduzir os resultados apresentados neste trabalho estão disponíveis no Apêndice A.

A classe responsável pela abstração de matrizes, *Matrix*, apresenta métodos básicos como operações matriciais, função traço e transposição. Para o processo de leitura e escrita de imagens, foram implementadas as classes *ImageReader* e *ImageWriter* respectivamente. As imagens utilizadas nesta aplicação estão no formato PNM em ASCII. A classe *Viewer*, como alternativa à *ImageWriter*, permite a visualização de uma imagem em tempo de execução, de forma que não haja a necessidade da escrita em memória secundária. Para auxiliar a manipulação de matrizes, foi implementada a classe *MatrixHandler* que oferece métodos como

split e *merge* que têm, respectivamente, a função de quebrar uma matriz em blocos de ordem k e reunir tais blocos. Além disso, os pacotes *br.ufsm.lacesm.transform.metrics.quality* e *br.ufsm.lacesm.transform.metrics.transforms* oferecem uma série de classes que implementam as métricas de qualidade de imagens, as métricas de similaridade com a DCT e ganho de codificação dadas na Seção 2.5.

As classes que implementam as transformadas presentes na literatura avaliadas neste trabalho, assim como a transformada proposta, estão no pacote *br.ufsm.lacesm.transform.sixteen*. Estas classes implementam, sobre as características comum a todas as transformadas (classe *Transform*), suas particularidades. Devido à estruturação das classes construídas, é possível adicionar novas transformadas ao conjunto já existente sem que haja necessidade de alterar o que está pronto.

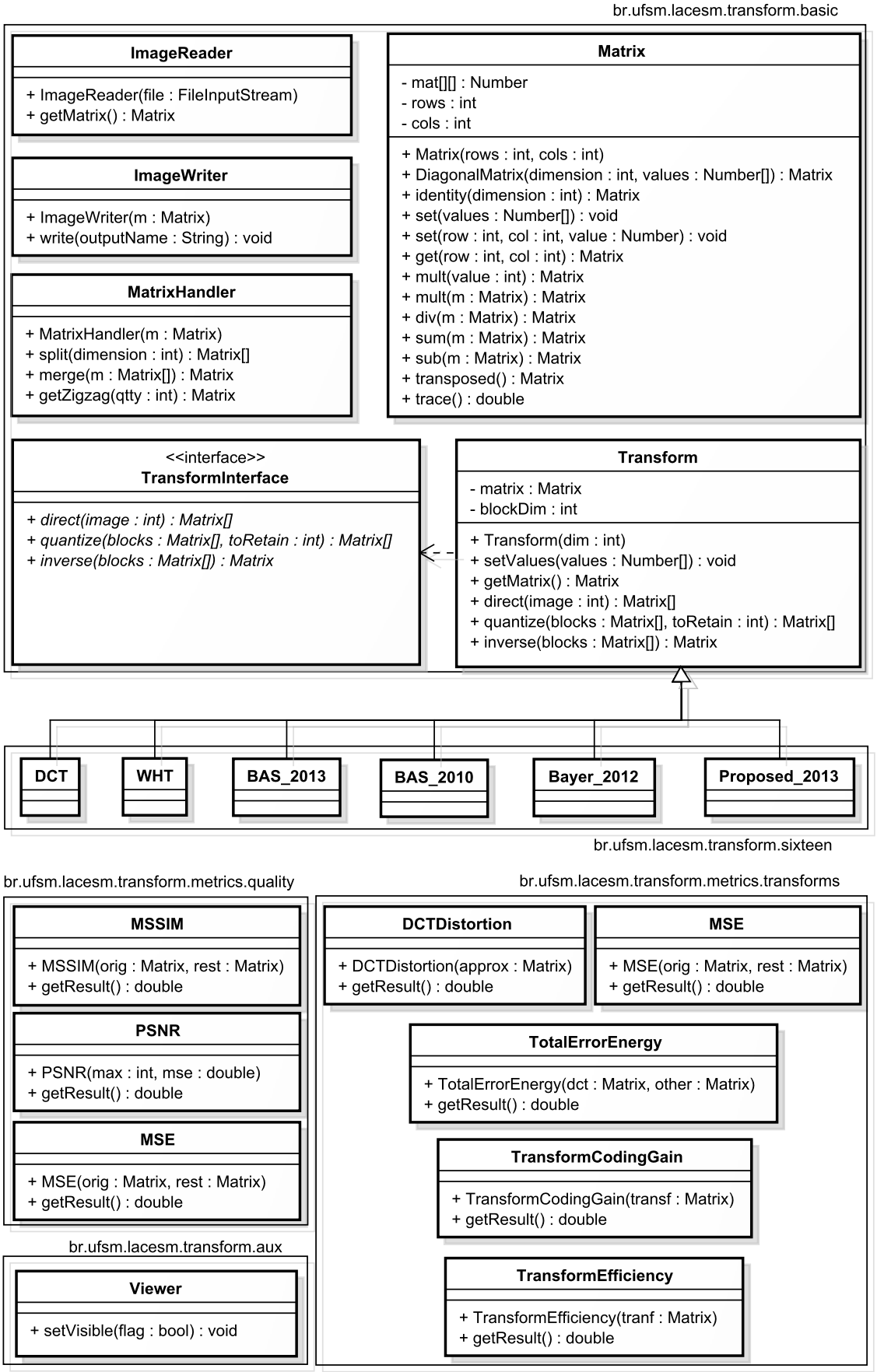


Figura 3.2: Diagrama de classes do conjunto de ferramentas implementadas para avaliação da transformada proposta.

4 RESULTADOS E DISCUSSÕES

4.1 Transformada proposta

Muitos algoritmos rápidos para a DCT de comprimento N têm estruturas que permitem derivar algoritmos rápidos para a DCT de comprimento $2N$ (BRITANAK; YIP; RAO, 2007; RAO; YIP, 1990; YIP; RAO, 1988). Baseado na (i) característica de decimação na frequência, (ii) na aproximação da DCT apresentada em (CINTRA; BAYER, 2011) e (iii) em algumas permutações e multiplicações de linhas por -1, obtém-se a matriz de transformação

$$\mathbf{T} = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
1 & 1 & 1 & 0 & 0 & -1 & -1 & -1 & -1 & -1 & -1 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 \\
1 & 0 & 0 & -1 & -1 & 0 & 0 & 1 & 1 & 0 & 0 & -1 & -1 & 0 & 0 & 1 \\
1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 \\
1 & 0 & -1 & -1 & 1 & 1 & 0 & -1 & -1 & 0 & 1 & 1 & -1 & -1 & 0 & 1 \\
0 & 0 & -1 & 1 & 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 & -1 & 1 & 0 & 0 \\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\
1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & -1 & 1 & 1 & -1 \\
1 & -1 & 0 & 1 & -1 & 0 & 1 & -1 & -1 & 1 & 0 & -1 & 1 & 0 & -1 & 1 \\
0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 \\
0 & -1 & 1 & 0 & 0 & 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & 1 & -1 & 0 \\
1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 \\
0 & -1 & 1 & -1 & 1 & -1 & 1 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 0 \\
1 & -1 & 0 & 0 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & 0 & 0 & 1 & -1
\end{bmatrix}.$$

Considerando o método de ortogonalização dado por (CINTRA; DIMITROV, 2010; CINTRA; BAYER, 2011)

$$\mathbf{S} = \sqrt{(\mathbf{T} \cdot \mathbf{T}^\top)^{-1}},$$

tem-se a aproximação da DCT ortogonal proposta escrita como

$$\hat{\mathbf{C}} = \mathbf{S} \cdot \mathbf{T},$$

em que

$$\mathbf{S} = \mathrm{diag}\left(\frac{1}{4}, \frac{1}{4}, \frac{1}{\sqrt{12}}, \frac{1}{\sqrt{8}}, \frac{1}{\sqrt{8}}, \frac{1}{4}, \frac{1}{\sqrt{12}}, \frac{1}{\sqrt{12}}, \frac{1}{4}, \frac{1}{\sqrt{12}}, \frac{1}{\sqrt{12}}, \frac{1}{\sqrt{8}}, \frac{1}{\sqrt{8}}, \frac{1}{\sqrt{12}}, \frac{1}{\sqrt{12}}, \frac{1}{\sqrt{12}}\right).$$

A transformada proposta neste trabalho, assim como em (BOUGUEZEL; AHMAD; SWAMY, 2010) e (BAYER et al., 2012) pode ser vista como um produto entre uma matriz diagonal $\mathbf{S}$ e uma matriz $\mathbf{T}$ composta por valores -1, 0 e 1. Como mencionado no Caítulo 3, em padrões de compressão como JPEG e HEVC, a matriz diagonal $\mathbf{S}$ pode ser calculada junto à matriz de quantização de modo que o custo computacional associado a esta não seja contabilizado.

Visando implementações eficientes principalmente em *hardware*, pode-se utilizar de artifícios como a fatoração em matrizes esparsas (BAYER; CINTRA, 2012). Dessa forma, a

complexidade computacional associada à transformada é dada pela soma das complexidades computacionais das matrizes que compõe sua fatoração. A aplicação da transformada sem algoritmo rápido exige 176 adições (ou subtrações) e nenhuma multiplicação, uma vez que multiplicações por 1 e -1, não são contabilizadas. Entretanto, pode-se reduzir a complexidade computacional de $\mathbf{T}$ utilizando o algoritmo rápido dado por

$$\mathbf{T} = \mathbf{P_2} \cdot \mathbf{M_4} \cdot \mathbf{M_3} \cdot \mathbf{M_2} \cdot \mathbf{P_1} \cdot \mathbf{M_1}, \tag{4.1}$$

sendo

$$\mathbf{P}_2 = \begin{bmatrix}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1
\end{bmatrix},$$

$$\mathbf{M}_4 = \begin{bmatrix}
1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&-1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1
\end{bmatrix},$$

$$\mathbf{M}_3 = \begin{bmatrix}
1&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&-1&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
1&0&0&-1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&1&1&0&0&0&0&0&0&0&0\\
0&0&0&0&-1&-1&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&-1&1&-1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&-1&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&-1&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&-1&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&-1\\
0&0&0&0&0&0&0&0&0&0&0&0&1&-1&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&-1&1
\end{bmatrix},$$

$$\mathbf{M}_2 = \begin{bmatrix}
1&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&-1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&-1&0&0&0&0&0&0&0&0&0&0\\
0&1&0&0&0&0&-1&0&0&0&0&0&0&0&0&0\\
1&0&0&0&0&0&0&-1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&-1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&-1&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&-1&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&-1
\end{bmatrix},$$

$$\mathbf{P}_1 = \begin{bmatrix}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0
\end{bmatrix} \text{ e}$$

$$\mathbf{M}_1 = \begin{bmatrix}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&1&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&-1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&-1&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&-1&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&-1&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&-1&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&-1&0&0\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&-1&0\\
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&-1
\end{bmatrix}.$$

Pode-se notar que todas as matrizes que compõe a fatoração de $\mathbf{T}$ possuem complexidade multiplicativa nula. Dessa forma, soma-se o número de adições de cada uma destas matrizes para obter o custo artimético associado ao algoritmo rápido da transformada proposta. A Tabela 4.1 apresenta explicitamente o custo de cada matriz que, multiplicadas, compõe $\mathbf{T}$. Através da fatoração dada na Equação 4.1 calcula-se $\mathbf{T}$ com apenas 60 adições.

Tabela 4.1: Complexidades aditiva e multiplicativa das matrizes esparsas que compõe $\mathbf{T}$

Matriz	Multiplicações	Adições	Deslocamentos de *bits*
$\mathbf{P_2}$	0	0	0
$\mathbf{M_4}$	0	4	0
$\mathbf{M_3}$	0	24	0
$\mathbf{M_2}$	0	16	0
$\mathbf{P_1}$	0	0	0
$\mathbf{M_1}$	0	16	0
Total	0	60	0

Ao fatorar $\mathbf{T}$ em uma sequência de matrizes esparsas, tem-se associado um algoritmo rápido na forma de diagrama de fluxo e sinal que pode ser traduzido para uma implementação da transformada em *hardware* como em (POTLURI et al., 2012, 2014). A Figura 4.1 apresenta o grafo de fluxo e sinal deste algoritmo rápido em que flexas sólidas representam multiplicações por 1, enquanto as flexas tracejadas representam multiplicações por -1. Além disso, quando há o encontro de duas ou mais flexas, soma-se seus valores. Em implementações em *hardware*, pode-se utilizar *adders* para somar os sinais. *Adders* são circuitos digitais lógicos que realizam

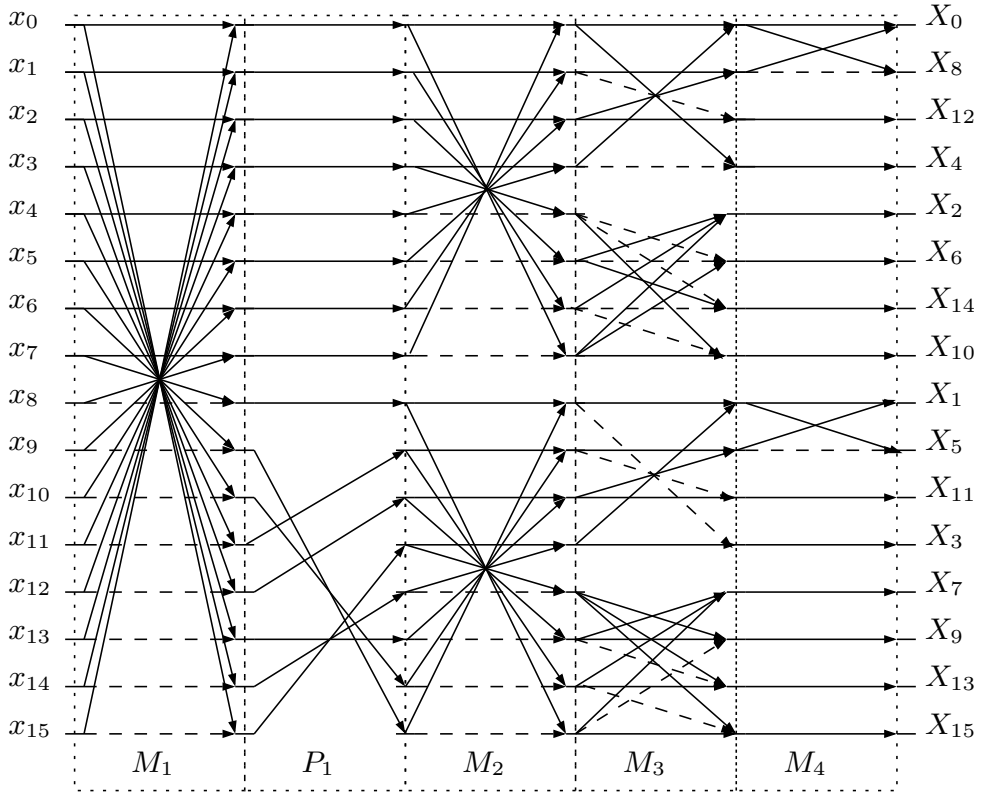


Figura 4.1: Diagrama de fluxo e sinal do algoritmo rápido da transformada proposta para implementação em *hardware*.

a soma de dois sinais de entrada e resultam em um sinal de saída. Caso se queira somar dois sinais, utilizam-se *adders* simples e, caso o número de sinais a serem somados seja maior do que dois, consideram-se *adders* cascateados. Neste caso, se três sinais precisam ser somados, por exemplo, somam-se os dois primeiros sinais em um *adder* simples e o resultado desta adição é então somado com o terceiro sinal em outro *adder* simples.

4.2 Avaliação de performance

Através da fatoração da matriz de transformação $\mathbf{T}$ em matrizes esparsas (Equação 4.1), é possível diminuir o custo computacional envolvido na aplicação de uma transformada (BAYER et al., 2012). Neste presente trabalho são apresentadas, em comparação com a transformada proposta, apenas aproximações da DCT de comprimento $k = 16$ com complexidade multiplicativa nula. Dessa forma, define-se a complexidade de uma transformada através do número de adições e deslocamentos de bits. A Tabela 4.2 apresenta as complexidades computacionais das aproximações da DCT estudadas neste trabalho, assim como da DCT exata. A complexidade computacional para a DCT exata foi calculada através do algortimo rápido de Chen (CHEN;

SMITH; FRALICK, 1977).

Tabela 4.2: Comparação da complexidade computacional da transformada proposta com as demais avaliadas neste trabalho

Tranformada	Multiplicações	Adições	Deslocamentos de *bits*	Total
DCT	44	74	0	118
WHT	0	64	0	64
BAS-2010	0	64	8	72
BAS-2013	0	64	0	64
Bayer-2012	0	72	0	72
Proposta	0	60	0	60

A Tabela 4.2 mostra que a transformada proposta neste trabalho apresenta o menor número de adições dentre as transformadas comparadas. Além disso, nenhum deslocamento de *bit* é necessário. Como resultado disso, pode-se concluir que a aproximação proposta neste trabalho apresenta a menor complexidade aritmética e, consequentemente, a menor complexidade computacional dentre as aproximações comparadas. Até o momento, desconhece-se a existência de uma transformada aproximada da DCT com menor custo computacional que a transformada proposta.

Diversas métricas, como apresentado na Seção 2.5, são encontradas na literatura com o intuito de comparar as performances das transformadas aproximadas da DCT. Algumas destas métricas representam, numericamente, o quão distante uma aproximação está da DCT exata, como é o caso da distorção da DCT (Seção 2.5.2.1), *Total Error Energy* (Seção 2.5.2.2) e *MSE* (Seção 2.5.2.3). Valores mais baixos para estas métricas significam maior similaridade com a DCT exata sob algum aspecto. Métricas como *Transform Coding Gain* (Seção 2.5.2.4) e *Transform Efficiency* (Seção 2.5.2.5) indicam o quão descorrelatos são os coeficientes de um bloco de uma imagem após a aplicação de determinada transformada. Estes são importantes parâmetros a serem analisados quando se deseja escolher uma transformada para aplicação em compressão de imagens do tipo JPEG.

Em geral, busca-se encontrar transformadas aproximadas cujos valores obtidos pela aplicação de todas estas métricas sejam os mais próximos dos valores obtidos para a DCT exata. Os valores obtidos pela aplicação destas métricas às aproximações presentes na literatura assim como à transformada proposta estão apresentados na Tabela 4.3.

Deve-se destacar que a transformada proposta neste trabalho, além de possuir o menor custo computacional, compete muito bem com as demais transformadas analisadas em termos de medidas de desempenho. Em medidas de similaridade com a DCT exata, a transformada

Tabela 4.3: Comparação da performance da transformada proposta com as demais avaliadas neste trabalho

Transformada	d_2	$\in$	MSE	C_g	η
DCT	0	0	0	9,4555	88,4518
WHT	0,8783	92,5631	0,4284	8,1941	70,6465
BAS-2010	0,6666	64,7490	0,1866	8,5208	73,6345
BAS-2013	0,5108	54,6207	0,1320	8,1941	70,6465
Bayer-2012	0,1519	8,0806	0,0465	7,8401	65,2789
Proposta	0,3405	30,3230	0,0639	8,295	70,8315

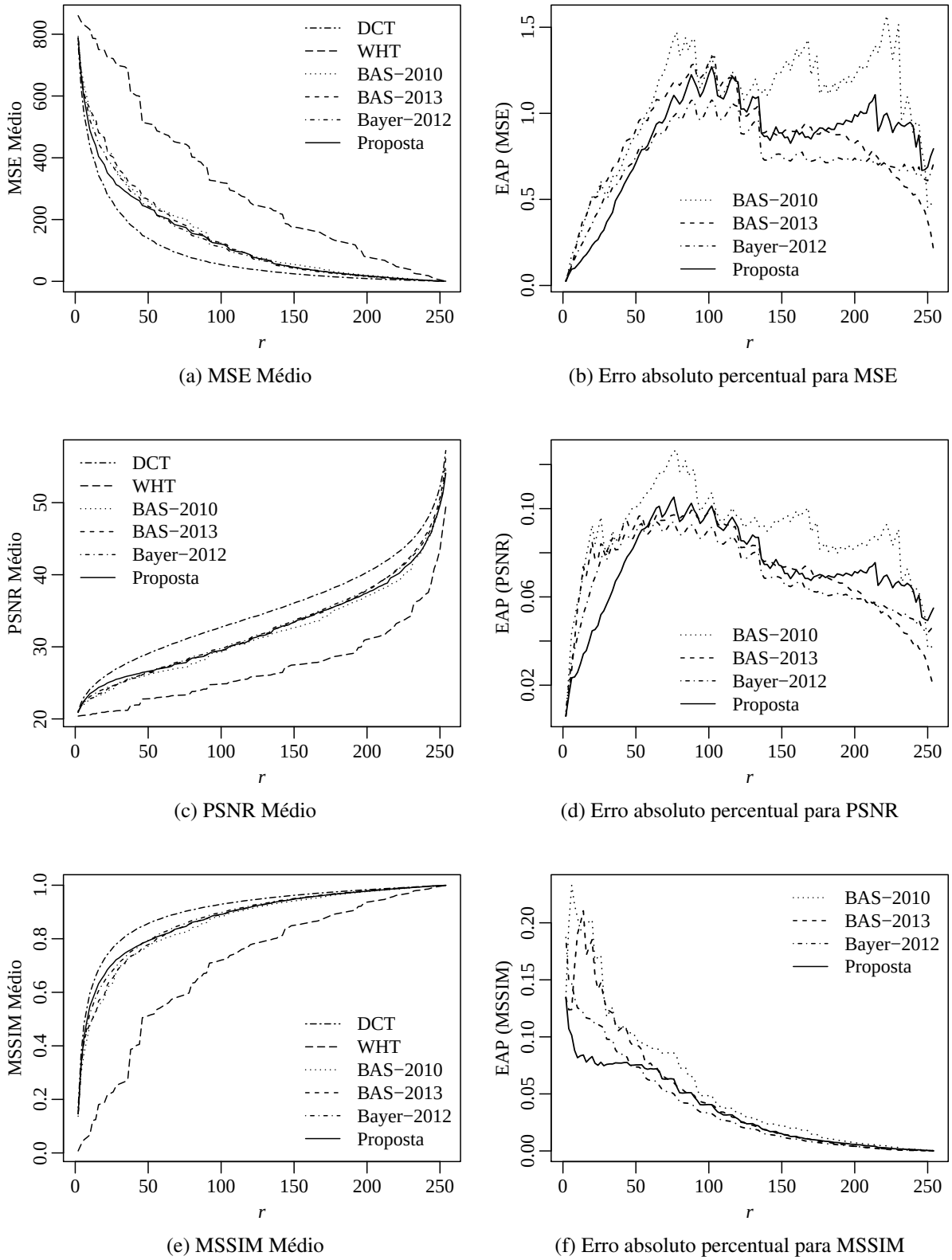
proposta apresenta resultados inferiores apenas aos de Bayer-2012 e, em medidas de ganho de codificação, apenas aos de BAS-2010. É importante salientar que a transformada proposta apresenta $16,6\%$ menos adições que Bayer-2012 e $6,25\%$ menos adições que BAS -2010, BAS - 2013 e a WHT, além de não apresentar nenhum deslocamento de *bit*.

4.3 Aplicação em compressão de imagens

Neste trabalho, seguiu-se a metodologia utilizada em trabalhos como (BOUGUEZEL; AHMAD; SWAMY, 2008, 2010; BAYER; CINTRA, 2012; CINTRA; BAYER, 2011; CINTRA; BAYER; TABLADA, 2014). A compressão das imagens, utilizando esta metodologia é feita através da seleção, seguindo a sequência *zig-zag*, dos r primeiros coeficientes. Os demais coeficientes são descartados. Neste caso, não há a divisão elemento a elemento dos blocos no domínio das transformadas por uma matriz de quantização, como mostrado no Capítulo 3.

Para realizar os experimentos comparativos entre as transformadas abordadas neste trabalho, assim como a transformada proposta, 45 imagens obtidas do Banco de Imagens Público foram utilizadas. Sobre estas imagens foram aplicadas as métricas MSE, PSNR e MSSIM para avaliar a eficiência das aproximações da DCT em compressão de imagens do tipo JPEG. As Figuras 4.2a, 4.2c e 4.2e ilustram os resultados médios – considerados mais robustos (BAYER et al., 2012; CINTRA; BAYER, 2011) – de tais métricas.

O erro absoluto percentual (EAP) entre as imagens comprimidas com as transformadas aproximadas em relação àquelas comprimidas com a DCT exata é apresentado nas Figuras 4.2b, 4.2d e 4.2f para o MSE, PSNR e MSSIM respectivamente. Para a melhor apresentação dos resultados gráficos, optou-se por retirar a WHT visto que há uma discrepância muito grande entre ela e a DCT exata. Por meio da interpretação dos gráficos apresentados na Figura 4.2, pode se ver que, para altas taxas de compressão ($r \lesssim 50$), a transformada proposta apresenta os me-



(a) MSE Médio

(b) Erro absoluto percentual para MSE

(c) PSNR Médio

(d) Erro absoluto percentual para PSNR

(e) MSSIM Médio

(f) Erro absoluto percentual para MSSIM

Figura 4.2: Medidas de qualidade de imagem aplicadas sobre diferentes valores de coeficientes retidos r.

lhores resultados. Para ilustrações mais pontuais, as Figuras 4.3, 4.5 e 4.7 mostram a aplicação das transformadas abordadas neste trabalho sobre as imagens *Boat*, *Lenna* e *Baboo*. Nestas imagens, há a retenção de $6,25\%$, $12,5\%$ e 25% do total de coeficientes, respectivamente. As Figuras 4.4, 4.6 e 4.8 apresentam, de maneira gráfica, a diferença entre as imagens originais e as imagens comprimidas. Para melhor visualização, como sugerido em (SALOMON, 2000), foi somado o valor 128 à cada coeficiente (*pixel*) das imagens que apresentam as diferenças entre a imagem original e a imagem comprimida. Assim, a imagem diferença é dada por

$$\mathbf{D}_{ij} = \mathbf{A}_{ij} - \mathbf{B}_{ij} + 128, \text{ para } i, j = 1, 2, 3, ..., N,$$

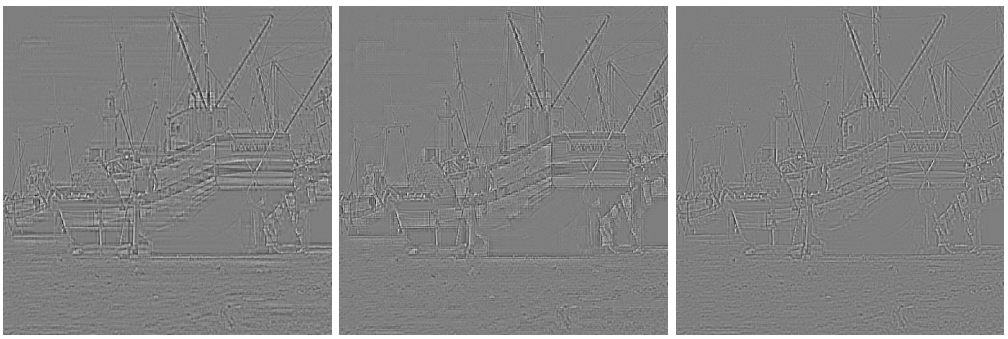
em que N é a largura e altura das imagens, $\mathbf{A}$ é a imagem original e $\mathbf{B}$ é a imagem comprimida. Nestas figuras, pode-se perceber por inspeção visual das imagens *Boat*, *Lenna* e *Baboo*, que a transformada proposta apresenta os melhores resultados para taxas de compressão acima de 80% ($r \lesssim 50$).

(a) DCT exata (PSNR=26.20dB) (b) WHT (PSNR=20.75dB) (c) BAS-2010 (PSNR=23.91dB)

(d) BAS-2013 (PSNR=23.96dB) (e) Bayer-2012 (PSNR=24.44dB) (f) Proposta (PSNR=25.04dB)

Figura 4.3: Imagem *Boat* reconstituída com retenção de 16 coeficientes.

(a) DCT exata (b) WHT (c) BAS-2010



(d) BAS-2013 (e) Bayer-2012 (f) Proposta

Figura 4.4: Diferença entre a imagem *Boat* original e a reconstituída com retenção de 16 coeficientes.

(a) DCT exata (PSNR=31.43dB) (b) WHT (PSNR=21.31dB) (c) BAS-2010 (PSNR=27.81dB)

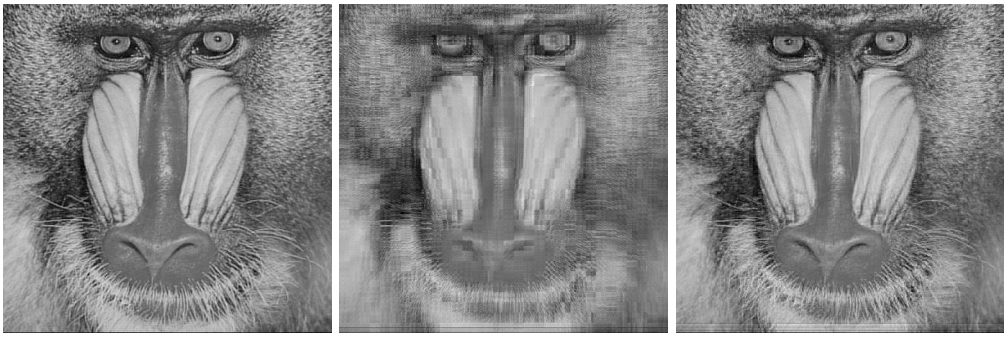
(d) BAS-2013 (PSNR=27.83dB) (e) Bayer-2012 (PSNR=27.42dB) (f) Proposta (PSNR=28.69dB)

Figura 4.5: Imagem *Lenna* reconstituída com retenção de 32 coeficientes.

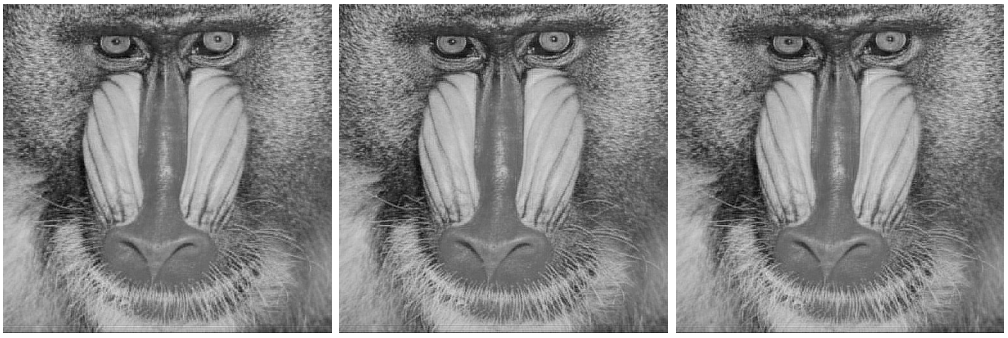
(a) DCT exata (b) WHT (c) BAS-2010

(d) BAS-2013 (e) Bayer-2012 (f) Proposta

Figura 4.6: Diferença entre a imagem *Lenna* original e a reconstituída com retenção de 32 coeficientes.

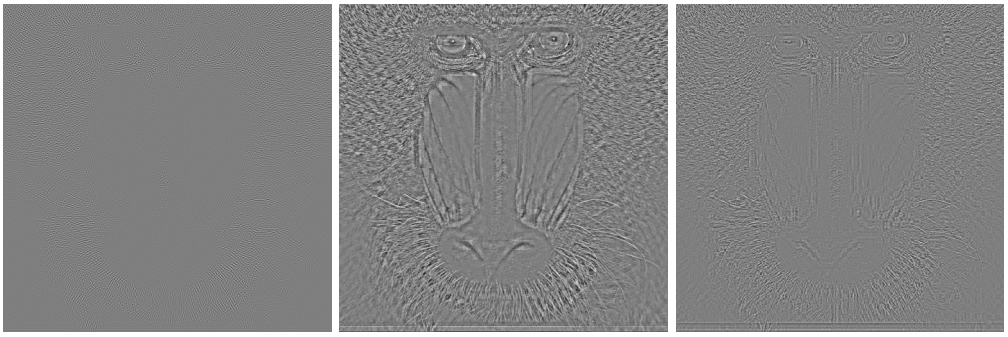


(a) DCT exata (PSNR=24.21dB) (b) WHT (PSNR=20.65dB) (c) BAS-2010 (PSNR=22.75dB)

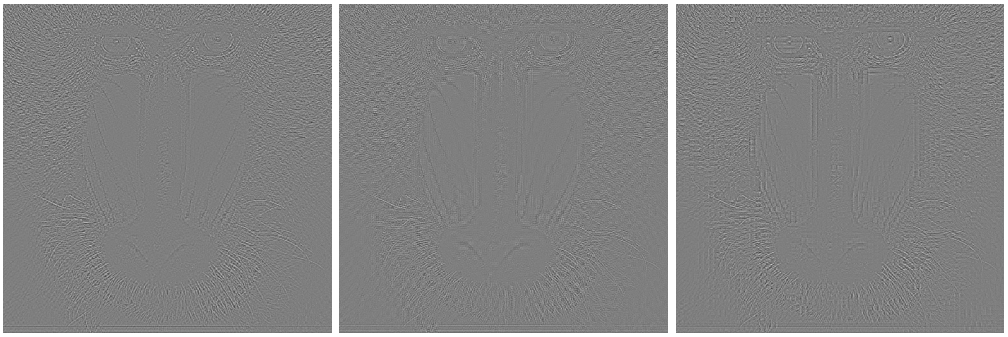


(d) BAS-2013 (PSNR=22.84dB) (e) Bayer-2012 (PSNR=23.19dB) (f) Proposta (PSNR=22.99dB)

Figura 4.7: Imagem *Baboo* reconstituída com retenção de 64 coeficientes.



(a) DCT exata (b) WHT (c) BAS-2010



(d) BAS-2013 (e) Bayer-2012 (f) Transformada proposta

Figura 4.8: Diferença entre a imagem *Baboo* original e a reconstituída com retenção de 64 coeficientes.

5 CONCLUSÕES

Este trabalho contextualizou a aplicação de transformadas discretas no processamento de sinais e enfatizou o uso da DCT em compressão de imagens do tipo JPEG. Uma breve introdução sobre o padrão de compressão de imagens JPEG e sobre a teoria de algoritmos rápidos foi apresentada. Além disso, uma revisão sobre as aproximações da DCT – especialmente de comprimento 16 – foi desenvolvida. Por fim, este trabalho apresentou uma nova transformada ortogonal de comprimento 16 de baixo custo computacional juntamente com seu algoritmo rápido. Medidas de performance foram consideradas, evidenciando o bom desempenho da transformada proposta em padrões de compressão de imagem e vídeo.

A transformada proposta possui o mais baixo custo computacional quando comparada com as demais aproximações para a DCT de mesmo comprimento encontradas na literatura e, além disso, compete muito bem em termos de similaridades com a DCT e ganho de codificação, como mostrado na Seção 4.2. A eficiência da transformada proposta em compressão de imagens do tipo JPEG é avaliada. Os melhores resultados para altas taxas de compressão, ou seja, compressões acima de 80%, são atingidos pela transformada proposta. Além disso, tal transformada apresenta resultados competitivos para taxas de compressão menores. Eficientes implementações em *hardware* e *software* podem ser feitas e aplicações em tempo real – principal razão pelo qual o HEVC foi proposto – como *streaming* de Internet, comunicação e videoconferência podem ser exploradas com o uso da transformada proposta.

REFERÊNCIAS

AHMED, N.; NATARAJAN, T.; RAO, K. R. Discrete Cosine Transform. **IEEE Transactions on Computers**, v.C-23, n.1, p.90–93, Jan. 1974.

ARAI, Y.; AGUI, T.; NAKAJIMA, M. A Fast DCT-SQ Scheme for Images. **Transactions of the IEICE**, v.E-71, n.11, p.1095–1097, Nov. 1988.

BAYER, F. M.; CINTRA, R. J. Image Compression via a Fast DCT Approximation. **IEEE Latin America Transactions**, v.8, n.6, p.708–713, Dec. 2010.

BAYER, F. M.; CINTRA, R. J. DCT-like transform for image compression requires 14 additions only. **Eletronics Letters**, v.48, 2012.

BAYER, F. M. et al. A Digital Hardware Fast Algorithm and FPGA-based Prototype for a Novel 16-point Approximate DCT for Image Compression Applications. **Measurement Science and Technology**, v.23, n.8, 2012.

BAYER, F. M. et al. Multiplierless approximate 4-point DCT VLSI architectures for transform block coding. **Eletronics Letters**, v.49, p.1532 – 1534, 2013.

BHASKARAN, V.; KONSTANTINIDES, K. **Image and Video Compression Standards**. Boston: Kluwer Academic Publishers, 1997.

BLAHUT, R. E. **Fast Algorithms for Signal Processing**, Cambridge University Press, 2010.

BOUGUEZEL, S.; AHMAD, M. O.; SWAMY, M. N. S. Low-complexity 8×8 transform for image compression. **Electronics Letters**, v.44, n.21, p.1249–1250, Sept. 2008.

BOUGUEZEL, S.; AHMAD, M. O.; SWAMY, M. N. S. A Low-Complexity Parametric Transform for Image Compression. **Proceedings of the 2011 IEEE International Symposium on Circuits and Systems**, 2011.

BOUGUEZEL, S.; AHMAD, M. O.; SWAMY, M. N. S. Binary Discrete Cosine and Hartley Transforms. **IEEE Transactions on Circuits and Systems I: Regular Papers**, v.60, n.4, p.989–1002, 2013.

BOUGUEZEL, S.; AHMAD, M.; SWAMY, M. A fast 8×8 transform for image compression. **2009 International Conference on Microelectronics (ICM)**, p.74–77, Dec. 2009.

BOUGUEZEL, S.; AHMAD, M.; SWAMY, M. A novel transform for image compression. **53rd IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)**, p.509–512, Aug. 2010.

BRACEWELL, R. N. The Hartley transform. **Oxford University Press**, New York, 1986.

BRIGGS, W. L.; HENSON, V. E. The DFT: an owner's manual for the discrete fourier transform. **SIAM**, 1995.

BRITANAK, V.; YIP, P.; RAO, K. R. **Discrete Cosine Transform Algorithms, Advantages, Applictions**, Academic Pres, 2006.

BRITANAK, V.; YIP, P.; RAO, K. R. **Discrete Cosine and Sine Transforms**, Academic Press, 2007.

CHEN, W. H.; SMITH, C.; FRALICK, S. A fast computational algorithm for the Discrete Cosine Transform. **IEEE Transactions on Communications**, v.25, n.9, p.1004–1009, Sept. 1977.

CINTRA, R. J.; BAYER, F. M. A DCT Approximation for Image Compression. **IEEE Signal Processing Letters**, v.18, n.10, p.579–582, Oct. 2011.

CINTRA, R. J.; BAYER, F. M.; TABLADA, C. J. Low-complexity 8-point DCT approximations based on integer functions. **Signal Processing**, 2014. In press.

CINTRA, R. J.; DIMITROV, V. S. The Arithmetic Cosine Transform: exact and approximate algorithms. **IEEE Transactions on Signal Processing**, v.58, n.6, p.3076–3085, June 2010.

EDIRISURIYA, A. et al. VLSI Architecture for 8-Point AI-based Arai DCT having Low Area-Time Complexity and Power at Improved Accuracy. **Journal of Low Power Electronics and Applications**, v.2, n.2, p.127–142, 2012.

FEIG, E.; WINOGRAD, S. Fast Algorithms for the Discrete Cosine Transform. **IEEE Transactions on Signal Processing**, v.40, n.9, p.2174–2193, 1992.

FLINT, A. C. Determining optimal medical image compression: psychometric and image distortion analysis. **BMC Medical Imaging**, v.12, 2012.

FONG, C.-K.; CHAM, W.-K. LLM Integer Cosine Transform and its fast algorithm. **IEEE Transactions on Circuits and Systems for Video Technology**, v.22, n.6, p.844–854, 2012.

HAWEEL, T. I. A new square wave transform based on the DCT. **Signal Processing**, v.82, p.2309–2319, 2001.

INTERNATIONAL TELECOMMUNICATION UNION. **ITU Recommendation T.81 Information technology - Digital compression and coding of continuous-tone still images - Requirements and guidelines**, ITU-T, 1993.

JAIN, A. K. **Fundamentals of digital image processing**. Upper Saddle River, N. J., USA: Prentice-Hall, Inc., 1989.

JAYANT, N. S.; NOLL, P. **Digital Coding of Wavetransforms**: principles and applications to speech and video, Prentice-Hall, 1984.

KOUADRIA, N. et al. Low complexity DCT for image compression in wireless visual sensor networks. **Eletronics Letters**, v.49, p.1531 – 1532, 2013.

LECUIRE, V.; MAKKAOUI, L.; MOUREAUX, J.-M. Fast zonal DCT for energy conservation in wireless image sensor networks. **Electronics Letters**, v.48, n.2, 2012.

LEE, B. G. A new algorithm for computing the discrete cosine transform. **IEEE Transactions on Acoustics, Speech and Signal Processing**, v.ASSP-32, p.1243–1245, Dec. 1984.

LENGWEHASATIT, K.; ORTEGA, A. Scalable variable complexity approximate forward DCT. **IEEE Transactions on Circuits and Systems for Video Technology**, v.14, n.11, p.1236–1248, Nov. 2004.

LIN, Y. W.; LEE, C. Y. Design of an FFT/IFFT processor for MIMO OFDM systems. **IEEE Transactions on Circuits and Systems-I: Regular Papers**, 2007.

LIU, J.; LIU, Y.; WANG, G. Fast dct-i, dct-iii, and dct-iv via moments. **EURASIP Journal on Applied Signal Processing**, p.1902–1909, 2005.

LOEFFLER, C.; LIGTENBERG, A.; MOSCHYTZ, G. Practical Fast 1D DCT Algorithms With 11 Multiplications. **Proceedings of the International Conference on Acoustics, Speech, and Signal Processing**, p.988–991, 1989.

LUTHRA, A.; SULLIVAN, G. J.; WIEGAND, T. Introduction to the special issue on the H.264/AVC video coding standard. **IEEE Transactions on Circuits and Systems for Video Technology**, v.13, n.7, p.557–559, July 2003.

PAO, I.-M.; SUN, M.-T. Approximation of calculations for forward discrete cosine transform. **IEEE Transactions on Circuits and Systems for Video Technology**, v.8, n.3, p.264–268, June 1998.

PENNEBAKER, W. B.; MITCHELL, J. L. **JPEG Still Image Data Compression Standard**. New York, NY: Van Nostrand Reinhold, 1992.

POTLURI, U. S. et al. Multiplier-free DCT approximations for RF multi-beam digital aperture-array space imaging and directional sensing. **Measurement Science and Technology**, v.23, n.11, p.114003, 2012.

POTLURI, U. S. et al. Improved 8-point Approximate DCT for Image and Video Compression Requiring Only 14 Additions. **IEEE Transactions on Circuits and Systems I**, 2014. In press.

POURAZAD, M. T. et al. HEVC: the new gold standard for video compression: how does HEVC compare with H.264/AVC? **IEEE Consumer Electronics Magazine**, v.1, n.3, p.36–46, July 2012.

R Core Team. **R**: a language and environment for statistical computing. Vienna, Austria: R Foundation for Statistical Computing, 2013.

RAO, K. R.; YIP, P. **Discrete Cosine Transform**: algorithms, advantages, applications. San Diego, CA: Academic Press, 1990.

ROBINSON, S. Toward an optimal algorithm for matrix multiplication. **SIAM News**, v.38, 2005.

ROMA, N.; SOUSA, L. Efficient hybrid DCT-domain algorithm for video spatial downscaling. **EURASIP Journal on Advances in Signal Processing**, New York, NY, v.2007, n.2, p.30–30, 2007.

SALOMON, D. **Data Compression**: the complete reference, Springer, 2000. v.2.

SALOMON, D. **The Computer Graphics Manual**, Springer, 2011. v.1.

SEBER, G. A. F. **A Matrix Handbook for Statisticians**, John Wiley and Sons, Inc, 2008.

SOUZA, G. J. da Silva Jr. e R. M. Campello de. Teoria da Complexidade Aditiva para Transformadas. **XXIX Simpósio Brasileiro de Telecomunicações**, 2011.

TABLADA, C.; BAYER, F. M.; CINTRA, R. J. Duas Aproximações para a DCT Baseadas na Fatoração de Chen. **XXXI Simpósio Brasileiro de Telecomunicações (sBrT2013)**, 2013.

TAKALA, J.; NIKARA, J. Unified pipeline architecture for discrete sine and cosine transforms of type IV. **Proceedings of the 3-rd International Conference on Information Communication and Signal Processing**, Singapore, 2001.

THE USC-SIPI Image Database. University of Southern California, Signal and Image Processing Institute, http://sipi.usc.edu/database/.

WALLACE, G. K. The JPEG still picture compression standard. **IEEE Transactions on Consumer Electronics**, v.38, n.1, 1992.

WANG, Z. Fast algorithms for the discrete W transform and for the discrete Fourier transform. **IEEE Transactions on Acoustics, Speech and Signal Processing**, v.ASSP-32, p.803–816, Aug. 1984.

WANG, Z. et al. Image quality assessment: from error visibility to structural similarity. **IEEE Transactions Image Processing**, v.13, n.4, p.600–612, Apr. 2004.

YIP, P.; RAO, K. R. The decimation-in-frequency algorithms for a family of discrete sine and cosine transform. **Circuits, Systems, and Signal Processing**, p.4–19, 1988.

APÊNDICES

APÊNDICE A – CÓDIGOS FONTE

```
/* Arquivo Viewer.java */
package br.ufsm.lacesm.transform.aux;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.Dimension;

import javax.swing.JLabel;
import javax.swing.JFrame;

import br.ufsm.lacesm.transform.basic.Matrix;

@SuppressWarnings("serial")

/* Classe reponsável pela implementação
de uma maneira alternativa de vizualização de uma
imagem em tempo de execução */
public class Viewer extends JFrame{

   private myCanvas canvas;

   /* Construtor. O parâmetros "title" e "m" são,
    respectivamente, o título da janela que irá
    ser criada e a imagem que será desenhada */
   public Viewer(String title, Matrix m){
     super(title);
     canvas = new myCanvas(m);

     Dimension dimension = new
         Dimension(m.getNumberOfColumns(),m.getNumberOfRows());

     add(canvas);
     setSize(dimension);
     setPreferredSize(dimension);
     setMinimumSize(dimension);
     setMaximumSize(dimension);

     setVisible(true);

  }

  /* Classe privada para auxiliar a classe
   Viewer */
  private class myCanvas extends JLabel{

    private Matrix matrix;

 /* Construtor. O parâmetro "image" é
 a imagem que será desenhada */
    public myCanvas(Matrix image){
      super();
      matrix = image;
    }
```

```
    /* Método público que desenha em
    um objeto Graphics a imagem recebida no
    construtor */
    public void paint(Graphics g){
      super.paint(g);
      for(int i = 0 ; i < matrix.getNumberOfRows(); i++){
        for(int j = 0; j < matrix.getNumberOfColumns(); j++){
          int component = matrix.get(i,j).intValue();
            if(component < 0)
              component = 0;
            if(component > 255)
              component = 255;
            g.setColor(new Color(component,component,component));
            g.drawRect(j, i, 1, 1);
        }
      }
    }
  }
}

/* Arquivo ImageReader.java */
package br.ufsm.lacesm.transform.basic;

import java.io.IOException;
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.util.Scanner;

import br.ufsm.lacesm.transform.exceptions.*;

/* Classe reponsável pela implementação
 da leitura de uma imagem no formato PNM em ASCII
 do tipo P2 (escala de cinza) */
public class ImageReader{

  private static String GRAYSCALE = "P2";

  private int maxValue;
  private String type;
  private Matrix content;

  /* Construtor. O parêmetro "file" éum objeto FileInputStream
  que representa o arquivo que contém a imagem deve ser lida */
  public ImageReader(FileInputStream file) throws IOException{
    String line;
    Integer index = 0, rowIndex = 0, columnIndex = 0;
    BufferedReader reader = new BufferedReader(new InputStreamReader(file));
    if(reader.ready()){

      try{
        verifyType(reader.readLine());
      }catch(UnsupportedFormatException e){
        System.out.println(e.getMessage());
      }

      line = reader.readLine();
```

```
    verifyDimensions(reader.readLine());

    maxValue = Integer.parseInt(reader.readLine());

    do{
      line = reader.readLine();
      if(line==null)
        break;
      Scanner scan = new Scanner(line);
      while(scan.hasNext()){
        Integer value = scan.nextInt();
        processContent(rowIndex,columnIndex,value);
        columnIndex++;
        if(columnIndex == content.getNumberOfColumns()){
          columnIndex = 0;
          rowIndex++;
        }
        index++;
      }
    }while(line!=null);
  }
  reader.close();
}

/* Método público utilizado para retornar a imagem
lida como um objeto da classe Matrix */
public Matrix getMatrix(){ return content; }

/* Método privado utilizado para verificar as dimensões da imagem lida.
O parâmetro string éo trecho do arquivo que contém as dimensões */
private void verifyDimensions(String string){
  Scanner scan = new Scanner(string);
  Integer width = scan.nextInt();
  Integer height = scan.nextInt();
  content = new Matrix(width, height);
}

/* Método privado utilizado para verificar o tipo da imagem lida.
O parâmetro string éo trecho do arquivo que contém o tipo */
private void verifyType(String string) throws UnsupportedFormatException{
  if(string.compareTo(GRAYSCALE)==0)
    type = GRAYSCALE;
  else throw new UnsupportedFormatException(string);
}

/* Método privado responsável pelo processamento dos dados (pixels) da
    imagem. Os parâmetros "rowIndex", "columnIndex" e "line" são,
    respectivamente, os índices da linha, coluna e o valor de um pixel. */
private void processContent(int rowIndex, int columnIndex, Integer line){
  if(type.compareTo(GRAYSCALE)==0){
    processGrayscale(rowIndex, columnIndex, (line));
  }
}

/* Método privado responsável pelo armazenamento da
imagem em memória primária. */
private void processGrayscale(int rowIndex, int columnIndex, int value){
  content.set(rowIndex, columnIndex, value);
```

```
  }

}

/* Arquivo ImageWriter.java */
package br.ufsm.lacesm.transform.basic;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

/* Classe reponsável pela implementação da gravação
de uma imagem de um objeto Matrix em um arquivo PNM
em ASCII no formato P2 */
public class ImageWriter {

  private static String GRAYSCALE = "P2";
  private static String MAXVALUES = "255";

  private Matrix data;

  /* Construtor. O parametro "data" é
  a matriz que contém a imagem a ser gravada */
  public ImageWriter(Matrix data){
    this.data = data;
  }

  /* Método público responsável pela gravação da imagem
  em um arquivo. O parametro "filename" deve indicar
  o diretório e nome do arquivo que será criado */
  public void write(String filename) throws IOException{
    BufferedWriter writer = new BufferedWriter(
       new FileWriter(new File(filename)));

    writer.write(GRAYSCALE + "\n");
    writer.flush();
    writer.write(data.getNumberOfRows() + " " + data.getNumberOfColumns() +
        "\n");
    writer.flush();
    writer.write(MAXVALUES + "\n");
    writer.flush();

    for(int i = 0; i < data.getNumberOfRows(); i++){
      for(int j = 0; j < data.getNumberOfColumns(); j++){
        writer.write(truncate(data.get(i,j).intValue())+"");
        writer.flush();
        if(j < data.getNumberOfColumns()){
          writer.write(" ");
          writer.flush();
        }
      }
      writer.write("\n");
      writer.flush();
    }

    writer.close();
  }
```

```
  /* Método privado usado para truncar valores
  que não podem ser representados em uma imagem em
  escala de cinza de 8 bits */
  private int truncate(int value){
    if(value > 255)
      return 255;
    if(value < 0)
      return 0;
    return value;
  }
}

/* Arquivo Matrix.java */
package br.ufsm.lacesm.transform.basic;

import br.ufsm.lacesm.transform.exceptions.*;

/* Classe responsável pela implementação da
 abstração de matrizes */
public class Matrix{
  protected Number[][] matrix;
  private int rows, columns;


  /* Construtor. O parâmetro "dimension" representa a dimensão
  de uma matriz que contém apenas valores "value" */
  public static Matrix constantMatrix(int dimension, Number value){
    Matrix i = new Matrix(dimension,dimension);

    for(int k = 0 ; k < dimension; k++)
      for(int l = 0; l < dimension; l++)
         i.set(k,l,value);
    return i;
  }

  /* Método público responsável pela criação de uma matriz
  de dimensão "dimension" cuja diagonal principal é
  composta pelos valores "diagonalCoef" e o demais elementos são nulos */
  public static Matrix DiagonalMatrix(int dimension, Number[] diagonalCoef)
  throws DifferDimensionException{

    if(dimension != diagonalCoef.length)
      throw new
      DifferDimensionException("The diagonal matrix is with incorrect number
          of coeficients");

    Matrix diag = new Matrix(dimension, dimension);

    for(int i = 0; i < diag.getNumberOfRows(); i++){
      for(int j = 0; j < diag.getNumberOfColumns(); j++){
        if(i!=j)
          diag.set(i,j,0.0);
        else
          diag.set(i,j,diagonalCoef[i]);
      }
    }
```

```
  return diag;
}

/* Construtor. O parâmetro "dimension" indica
a ordem da matriz identidade que será criada */
public static Matrix identity(int dimension){
  Matrix i = new Matrix(dimension,dimension);

  for(int k = 0 ; k < dimension; k++)
    for(int l = 0; l < dimension; l++){
      if( k==l )
        i.set(k,l,1.0);
      else
        i.set(k,l,0.0);
    }


  return i;
}

/* Construtor de copia. O parâmetro "other" é
a matriz que será copiada */
public Matrix(Matrix other){
  rows = other.getNumberOfColumns();
  columns = other.getNumberOfColumns();
  matrix = (Number[][])new Number[rows][columns];
  for(int i = 0; i < rows; i++)
    for(int j = 0; j < columns; j++)
      set(i,j,other.get(i,j));
}

/* Construtor. Os parâmetros "rows" e "columns"
são as dimensões da matriz que será criada */
public Matrix(int rows, int columns){
  this.rows = rows;
  this.columns = columns;
  matrix = (Number[][])new Number[rows][columns];
}

/* Método público que retorna o ("row","column")-ésimo
elemento da matriz */
public Number get(int row, int column){
  return matrix[row][column];
}

/* Método público que atribui o valor "value" ao
("row","column")-ésimo elemento da matriz */
public void set(int row, int column, Number value){
  matrix[row][column]=value;
}

/* Método público que preenche a matriz com
 valores "values" */
public void set(Number[] values)
   throws OutOfBoundsMatrixException{
  if(values.length != rows*columns) throw new
     OutOfBoundsMatrixException();
  int row = 0, column = 0;
```

```
  for(int index = 0; index < values.length; index++){
    set(row,column,values[index]);
    column++;
    if(column == columns){
      row++;
      column = 0;
    }
  }
}

/* Método público que implementa a multiplicação
de uma matriz por escalar */
public Matrix mult(Number value){
  Matrix newest = new Matrix(getNumberOfRows(), getNumberOfColumns());
  for(int i = 0; i < getNumberOfRows(); i++)
    for(int j = 0; j < getNumberOfColumns(); j++)
      newest.set(i,j,value.doubleValue()*
      get(i,j).doubleValue());
  return newest;
}

/* Método público que implementa a divisão elemento a elemento de
   matrizes */
public Matrix div(Matrix other)
   throws DifferDimensionException{
  Matrix newest;
  if((other.getNumberOfColumns() == getNumberOfColumns())
     &&(other.getNumberOfRows() == getNumberOfRows())){
    newest = new Matrix(getNumberOfRows(),getNumberOfColumns());
    for(int i = 0; i < getNumberOfRows(); i++){
      for(int j = 0; j < getNumberOfColumns(); j++){
        newest.set(i, j, get(i,j).doubleValue() /
            other.get(i,j).doubleValue());
      }
    }
  }
  else{
    throw new DifferDimensionException();
  }
  return newest;
}

/* Método público que implementa a soma de duas matrizes */
public Matrix sum(Matrix other)
   throws DifferDimensionException{
  Matrix newest;
  if((other.getNumberOfColumns() == getNumberOfColumns())
     &&(other.getNumberOfRows() == getNumberOfRows())){
    newest = new Matrix(getNumberOfRows(),getNumberOfColumns());
    for(int i = 0; i < getNumberOfRows(); i++){
      for(int j = 0; j < getNumberOfColumns(); j++){
        newest.set(i, j, (get(i,j).doubleValue() +
            other.get(i,j).doubleValue()));
      }
    }
  }
  else{
    throw new DifferDimensionException();
```

```
  }
  return newest;
}

/* Método público que implementa a subtração de duas
 matrizes */
public Matrix sub(Matrix other)
   throws DifferDimensionException{
  Matrix newest;
  if((other.getNumberOfColumns() == getNumberOfColumns())
     &&(other.getNumberOfRows() == getNumberOfRows())){
    newest = new Matrix(getNumberOfRows(),getNumberOfColumns());
    for(int i = 0; i < getNumberOfRows(); i++){
      for(int j = 0; j < getNumberOfColumns(); j++){
        newest.set(i, j, (get(i,j).doubleValue() -
            other.get(i,j).doubleValue()));
      }
    }
  }
  else{
    throw new DifferDimensionException();
  }
  return newest;
}

/* Método público que implementa a multiplicação de duas matrizes */
public Matrix mult(Matrix other)
   throws DifferDimensionException{
  Matrix newest;
  if(other.getNumberOfRows() == getNumberOfColumns()){
    newest = new Matrix(getNumberOfRows(),other.getNumberOfColumns());
    for(int i = 0; i < newest.getNumberOfRows(); i++){
      for(int j = 0; j < newest.getNumberOfColumns(); j++){
        Double value = 0.0;
        for(int k = 0; k < getNumberOfColumns(); k++){
           value = value.doubleValue() +
           get(i,k).doubleValue() * other.get(k,j).doubleValue();
        }
        newest.set(i, j, value);
      }
    }
  }else
    throw new DifferDimensionException();
  return newest;
}

/* Método público que implementa a transposição
de matrizes */
public Matrix transposed(){
  Matrix trans = new Matrix(getNumberOfColumns(), getNumberOfRows());
  for(int i = 0; i < getNumberOfRows(); i++)
    for(int j = 0; j < getNumberOfColumns(); j++)
      trans.set(j,i,get(i,j));
  return trans;
}

/* Método público que retorna o número
de linhas da matriz */
```

```
public int getNumberOfRows(){
  return rows;
}

/* Método público que retorna o número
de colunas da matriz */
public int getNumberOfColumns(){
  return columns;
}

/* Método público que imprime a matriz
no terminal */
public void print(){
  for(int i = 0; i < getNumberOfRows(); i++){
    for(int j = 0; j < getNumberOfColumns(); j++){
      System.out.print(get(i,j).doubleValue() + " ");
    }
    System.out.println("");
  }
  System.out.println("");
}

/* Método público que implementa a função traço */
public static double trace(Matrix mat) {
  double trace = 0.0;
  for(int i = 0; i < mat.getNumberOfRows(); i++){
    for(int j = 0; j < mat.getNumberOfColumns(); j++){
      if(i == j)
        trace += mat.get(i,j).doubleValue();
    }
  }
  return trace;
}

/* Método público que atribui valores "values" a
"row"-ésima linha da matriz */
public void setRow(int row, Number[] values){
  if(values.length != getNumberOfColumns())
    System.out.println("The number of columns differ");
  for(int i = 0; i < getNumberOfColumns(); i++){
    set(row, i, values[i]);
  }
}

/* Método público que obtem a "row"-ésima linha
 da matriz */
public Number[] getRow(int row){
  Number[] ret = new Number[getNumberOfColumns()];
  for(int i = 0; i < ret.length; i++){
    ret[i] = get(row, i);
  }
  return ret;
}

/* Método público que verifica se duas
matrizes são iguais */
public boolean isEqual(Matrix m){
  if(m.getNumberOfColumns() != getNumberOfColumns() ||
```

```
      m.getNumberOfRows() != getNumberOfRows())
     return false;
    for(int i = 0; i < m.getNumberOfRows(); i++){
     for(int j = 0; j < m.getNumberOfColumns(); j++){
      if(get(i,j).doubleValue() != m.get(i,j).doubleValue())
       return false;
     }
    }
    return true;
  }

  /* Método público que soma um escalar a
  todos os elementos de uma matriz */
  public Matrix sum(int value) {
    Matrix result = this.clone();
    for(int i = 0; i < getNumberOfRows(); i++){
     for(int j = 0; j < getNumberOfColumns(); j++){
      result.set(i,j,result.get(i,j).doubleValue() + value);
     }
    }
    return result;
  }
}

/* Arquivo MatrixHandler.java */
package br.ufsm.lacesm.transform.basic;

/* Classe responsável pela implementação de funções de manipulação de
   matrizes */
public class MatrixHandler {

  private Matrix matrix;

  /* Construtor. O parâmetro "matrix" é
   a matriz que será manipulada */
  public MatrixHandler(Matrix matrix){
    this.matrix = matrix;
  }

  /* Método público que retorna os "numbOfCoef"
  primeiros valores de uma matriz seguindo a sequência zig-zag */
  public Matrix getZigzag(int numbOfCoef){
    Matrix zigzag = new
        Matrix(matrix.getNumberOfRows(),matrix.getNumberOfColumns());

    Integer i = 0, j = 0, index = 0;
    Number[] list = doTheSequence(numbOfCoef);

    while(i < matrix.getNumberOfColumns()){
     if(index < numbOfCoef)
      zigzag.set(i,j,list[index]);
     else
      zigzag.set(i,j,new Double(0));
     if(i==matrix.getNumberOfColumns()-1){
      i = j+1; j = matrix.getNumberOfColumns()-1;
     }
     else if(j==0){
      j = i+1; i = 0;
```

```
    }
    else{
      i++; j--;
    }
    index++;
  }

  return zigzag;
}

/* Método público que atribui "values" aos "values.length" primeiros
    elementos de uma matriz e atribui zero aos demais */
public void setAndCompleteWithZeros(Number[] values){
  int row = 0, column = 0;
  for(int index = 0; index <
      matrix.getNumberOfColumns()*matrix.getNumberOfRows(); index++){
    if(index < values.length)
      matrix.set(row,column,values[index]);
    else
      matrix.set(row,column,0.0);
    column++;
    if(column == matrix.getNumberOfColumns()){
      row++;
      column = 0;
    }
  }
}

/* Método público que retorna a matriz - seja
ela modificada ou não */
public Matrix getMatrix(){ return matrix; }

/* Método público responsável pela divisão de uma
 matriz em blocos de tamanho "dimension" */
public Matrix[] split(int dimension){
  int parts = (matrix.getNumberOfColumns()*matrix.getNumberOfRows())/
      (dimension*dimension);

  Matrix[] matrixes = new Matrix[parts];

  for(int i = 0; i < parts; i++)
    matrixes[i] = new Matrix(dimension,dimension);

  int ir = 0, ic = 0, it = 0;

  while(it != parts){
    for(int i = 0; i < dimension; i++){
      for(int j = 0; j < dimension; j++){
        matrixes[it].set(i,j,matrix.get(i+ir,j+ic));
      }
    }
    if(ic+dimension < matrix.getNumberOfColumns())
      ic+=dimension;
    else{
      ic=0;
      if(ir+dimension < matrix.getNumberOfRows())
        ir+=dimension;
    }
```

```
    it++;
  }
  return matrixes;
}

/* Método público responsável pelo rearranjo do vetor de
 matrizes em uma só */
public Matrix merge(Matrix[] vector){
  int iterator = 0, rowIterator = 0, columnIterator = 0;
  int dimension = vector[0].getNumberOfColumns();
  while(iterator < vector.length){
    for(int i = 0; i < dimension; i++){
      for(int j = 0; j < dimension; j++){
        Number value = vector[iterator].get(i,j);
        matrix.set(i + rowIterator, j + columnIterator, value);
      }
    }
    iterator++;
    columnIterator += dimension;
    if(columnIterator == matrix.getNumberOfColumns()){
      rowIterator += dimension;
      columnIterator = 0;
    }
  }
  return matrix;
}

/* Método privado que auxilia na obtenção dos primeiros
"numbOfCoef" elementos - seguindo a sequencia zig-zag -
de uma matriz */
private Number[] doTheSequence(int numbOfCoef){
  Number[] list = new Number[numbOfCoef];
  Integer i = 0, j = 0, index = 0;

  while(i < matrix.getNumberOfColumns()){
    if(index < numbOfCoef)
      list[index] = matrix.get(i,j);
    else
      break;
    if(i==matrix.getNumberOfColumns()-1){
      i = j+1; j = matrix.getNumberOfColumns()-1;
    }
    else if(j==0){
      j = i+1; i = 0;
    }
    else{
      i++; j--;
    }
    index++;
  }

  return list;
}

/* Método público que transforma uma matriz de ordem m x n
em um vetor de tamanho m.n */
public static Number[] toArray(Matrix matrix){
  int index = 0;
```

```
    Number[] result = new Number[matrix.getNumberOfColumns()*
                          matrix.getNumberOfRows()];

    for(int i = 0; i < matrix.getNumberOfRows(); i++)
      for(int j = 0; j < matrix.getNumberOfColumns(); j++){
        result[index] = matrix.get(i,j);
        index++;
      }

    return result;
  }

}


/* Arquivo Transform.java */
package br.ufsm.lacesm.transform.basic;

import br.ufsm.lacesm.transform.exceptions.DifferDimensionException;
import br.ufsm.lacesm.transform.exceptions.OutOfBoundsMatrixException;

/* Classe abstrata responsável pela implementação
da interface das transformadas */
public abstract class Transform implements TransformInterface{
  protected Matrix matrix;
  protected Integer blockDimension;

  /* Construtor. O parâmetro informado éutilizado
  para criar uma transformada cuja matriz de transformação
  é de ordem "dimension" */
  protected Transform(int dimension) {
    blockDimension = dimension;
    matrix = new Matrix(dimension,dimension);
  }

  /* Método protegido responsável pela atribuição de
  "values" a uma matriz de transformação */
  protected void setValues(Number[] values){
    try{
      matrix.set(values);
    }catch(OutOfBoundsMatrixException e){
      System.out.println(e.getMessage());
    }
  }

  /* Método público que retorna a matriz de
  transformação de uma transformada */
  public Matrix getMatrix(){
    return matrix;
  }

  /* Método público que implementa a transformação direta
   padrão de uma transformada ortogonal */
  public Matrix[] direct(Matrix image){
    Matrix[] rblocks = new MatrixHandler(image).split(blockDimension);
    for(int i = 0; i < rblocks.length; i++)
      try {
        rblocks[i] = (getMatrix().mult(rblocks[i])).mult(getMatrix()
            .transposed());
```

```
    } catch (DifferDimensionException e) {
      e.printStackTrace();
    }
   return rblocks;
  }

  /* Método público responsável pela implementação da
   quantização utilizada neste trabalho */
  public Matrix[] quantize(Matrix[] blocks, Integer retainedCoef) {
    for(int i = 0; i < blocks.length; i++){
      blocks[i] = new MatrixHandler(blocks[i]).getZigzag(
         retainedCoef);
    }
    return blocks;
  }

  /* Método público responsável pela implementação da
   transformação inversa padrão de uma transformada ortogonal */
  public Matrix inverse(Matrix[] blocks) {
    try {
      for(int i = 0; i < blocks.length; i++){
        blocks[i] = (getMatrix().transposed().
           mult(blocks[i])).mult(getMatrix());
      }
    } catch (DifferDimensionException e) {
      e.printStackTrace();
    }
    return new MatrixHandler(new Matrix(512,512)).merge(blocks);
  }
}

/* Arquivo TransformInterface.java */
package br.ufsm.lacesm.transform.basic;

/* Interface responsável pela declaração dos métodos
que todas transformadas precisam obrigatoriamente implementar */
public interface TransformInterface {
  public Matrix[] direct(Matrix image);
  public Matrix[] quantize(Matrix[] blocks, Integer retainedCoef);
  public Matrix inverse(Matrix[] blocks);
}

/* Arquivo DifferDimensionException.java */
package br.ufsm.lacesm.transform.exceptions;

/* Classe responsável pela implementação da excessão que
deve ser lançada quando duas matrizes que precisaam ter
mesma dimensão, nao têm */
public class DifferDimensionException extends Exception{

  private static final long serialVersionUID = 1L;

  /* Construtor padrão */
  public DifferDimensionException(){
    super("The matrix dimensions are different");
  }
```

```
  /* Construtor. O parâmetro "message" é
  uma mensagem customizada pelo programador */
  public DifferDimensionException(String message){
    super(message);
  }
}

/* Arquivo OutOfBoundsMatrixException.java */
package br.ufsm.lacesm.transform.exceptions;

/* Classe responsável pela implementação da excessão que
deve ser lançada quando a leitura dos elementos de uma
matriz ultrapassa seus limites (número de linhas x número de colunas) */
public class OutOfBoundsMatrixException extends Exception{

  private static final long serialVersionUID = 1L;

  /* Construtor padrão */
  public OutOfBoundsMatrixException(){
    super("Some data is out of bounds");
  }

  /* Construtor. O parâmetro "message" é
  uma mensagem customizada pelo programador */
  public OutOfBoundsMatrixException(String message){
    super(message);
  }
}

/* Arquivo OutofBoundsZigZagException.java */
package br.ufsm.lacesm.transform.exceptions;

/* Classe responsável pela implementação da excessão que deve ser
lançada quando o número de elementos desejados em uma sequência zigzag
é maior que o número total de elementos de uma matriz */
public class OutOfBoundsZigZagException extends Exception{

  private static final long serialVersionUID = 1L;

  /* Construtor padrão */
  public OutOfBoundsZigZagException(){
    super("You required too much numbers");
  }

  /* Construtor. O parâmetro "message" é
  uma mensagem customizada pelo programador */
  public OutOfBoundsZigZagException(String message){
    super(message);
  }
}

/* Arquivo UnsupportedFormatException.java */
package br.ufsm.lacesm.transform.exceptions;

/* Classe responsável pela implementação da excessão que
deve ser lançada quando a imagem que se deseja ler
não écompatível com o formato PNM ASCII do tipo P2 */
```

```
public class UnsupportedFormatException extends Exception{

  private static final long serialVersionUID = 1L;

  /* Construtor padrão */
  public UnsupportedFormatException(){
    super("The PNM format isn't supported");
  }

  /* Construtor. O parâmetro "message" é
  uma mensagem customizada pelo programador */
  public UnsupportedFormatException(String message){
    super("The PNM format '"+ message + "' isn't supported");
  }
}


/* Arquivo MSE.java */
package br.ufsm.lacesm.transform.metrics.quality;

import br.ufsm.lacesm.transform.basic.Matrix;

/* Classe responsável pela implementação da métrica de qualidade de
   imagens, dada uma imagem de referencia, MSE */
public class MSE {

  private Double result;

  /* Construtor que recebe as imagens original "original"
  e reconstruída "restored" */
  public MSE(Matrix original, Matrix restored){
    Double mse = 0.0;
    for(int i = 0; i < original.getNumberOfRows(); i++){
      for(int j = 0; j < original.getNumberOfColumns(); j++){
        mse += Math.pow((original.get(i,j).doubleValue() -
            restored.get(i,j).doubleValue()),2.0);
      }
    }
    result = (mse/((double)(original.getNumberOfColumns())
    *(double)(original.getNumberOfRows())));
  }

  /* Método público que retorna o resultado desta métrica */
  public Double getResult(){ return result; }
}


/* Arquivo MSSIM.java */
package br.ufsm.lacesm.transform.metrics.quality;

import br.ufsm.lacesm.transform.basic.Matrix;
import br.ufsm.lacesm.transform.basic.MatrixHandler;
import br.ufsm.lacesm.transform.exceptions.DifferDimensionException;

/* Classe responsável pela implementação da métrica de qualidade de
   imagens, dada uma imagem de referencia, MSSIM */
public class MSSIM {

  Double result;
```

```
private static final Double L = 255.0;

private static final double K1 = 0.0001;
private static final Double C1 = Math.pow(K1 * L, 2.0);

private static final Double K2 = 0.0003;
private static final Double C2 = Math.pow(K2 * L, 2.0);

 /* Construtor que recebe as imagens original "original"
e reconstruída "reconstructed" e inicializa o cálculo do MSSIM */
public MSSIM(Matrix original, Matrix reconstructed){
  result = 0.0;
  try{
    Matrix[] x = new MatrixHandler(original).split(16);
    Matrix[] y = new MatrixHandler(reconstructed).split(16);

    for(int i = 0; i < x.length; i++){
      result += SSIM(x[i],y[i]);
    }

    result = result/(double)(x.length);

  }catch(DifferDimensionException e){
    e.printStackTrace();
  }
}

/* Método privado que calcula o SSIM */
private Double SSIM(Matrix X, Matrix Y) throws DifferDimensionException{
  return (
       (2.0 * mean(X) * mean(Y) + C1) *
       (2.0 * cov(X,Y) + C2)
     )
     /
     (
        (Math.pow(mean(X),2.0) + Math.pow(mean(Y),2.0) + C1) *
        (Math.pow(squareRootOfVariance(X),2.0)+
         Math.pow(squareRootOfVariance(Y), 2.0)+
         C2)
      );
}

/* Método público que calcula a média do elementos
 de uma matriz */
private double mean(Matrix matrix){
  Double result = 0.0;
  Number[] image = MatrixHandler.toArray(matrix);
  for(int i = 0; i < image.length; i++){
    result += image[i].doubleValue();
  }

  return result/(double)image.length;
}

/* Método privado que calcula a raiz quadrada da
variância dos elementos de uma matriz */
private double squareRootOfVariance(Matrix matrix){
```

```
    Double result = 0.0;
    Double avg = mean(matrix);
    Number[] image = MatrixHandler.toArray(matrix);
    for(int i = 0; i < image.length; i++){
      result += Math.pow((image[i].doubleValue() - avg), 2.0);
    }

    return Math.sqrt(result/((double)image.length - 1.0));
  }

   /* Método privado que calcula a covariância entre
   as imagens original "original" e reconstruída "reconstructed" */
  private double cov(Matrix original, Matrix reconstructed)
     throws DifferDimensionException{
    Double result = 0.0;
    Double avgOriginal = mean(original);
    Double avgReconst = mean(reconstructed);
    Number[] imageOrig = MatrixHandler.toArray(original);
    Number[] imageReconst = MatrixHandler.toArray(reconstructed);
    if(imageOrig.length != imageReconst.length)
      throw new DifferDimensionException();
    for(int i = 0; i < imageOrig.length; i++){
      result += (imageOrig[i].doubleValue() - avgOriginal)*
         (imageReconst[i].doubleValue()- avgReconst);
    }

    return result/((double)imageOrig.length - 1.0);
  }

  /* Método público que retorna o resultado desta métrica */
  public double getResult(){ return result; }

}


/* Arquivo PSNR.java */
package br.ufsm.lacesm.transform.metrics.quality;

/* Classe responsável pela implementação da métrica de qualidade de
    imagens, dada uma imagem de referencia, PSNR */
public class PSNR {

  private Double result;

   /* Construtor que recebe o valor máximo "maxValue" que um pixel
  pode ter e o resultado do MSE para fazer o cálculo do PSNR*/
  public PSNR(Integer maxValue, Double MSE){
    result = 10.0*Math.log10(new Double(maxValue*maxValue) / MSE);
  }

  /* Método público que retorna o resultado desta métrica */
  public Double getResult(){ return result; }

}


/* Arquivo DCTDistortion.java */
package br.ufsm.lacesm.transform.metrics.transforms;
```

```
import br.ufsm.lacesm.transform.basic.Matrix;
import br.ufsm.lacesm.transform.exceptions.DifferDimensionException;
import br.ufsm.lacesm.transform.sixteen.DCT_II;

/* Classe responsável pela implementação da métrica distorcao da DCT */
public class DCTDistortion {
  double result;

  /* Construtor que recebe a matriz de transformação
  "approx" para calcular a distorção da DCT */
  public DCTDistortion(Matrix approx){
    result = 0.0;
    Integer matrixOrder = approx.getNumberOfRows();
    try {
      Matrix other = new DCT_II(matrixOrder).getMatrix()
      .mult(approx.transposed());
      result = 1.0 - (1.0/(double)(matrixOrder)) *
          frobeniusPow2(diagonalMatrix(other));
    } catch (DifferDimensionException e) {
      System.out.println(e.getMessage());
    }

  }

  /* Método privado que, dada uma matriz "other", retorna
  apenas os elementos de sua diagonal */
  private Matrix diagonalMatrix(Matrix other) {
    Matrix diag = new Matrix(other.getNumberOfRows(),
        other.getNumberOfColumns());
    for(int i = 0; i < other.getNumberOfRows(); i++){
      for(int j = 0; j < other.getNumberOfColumns(); j++){
        if(i == j)
          diag.set(i,j,other.get(i, j));
        else
          diag.set(i,j,0);
      }
    }
    return diag;
  }

  /* Método privado que calcula a norma euclideana
   e a eleva ao quadrado */
  private double frobeniusPow2(Matrix mat) {
    double total = 0.0;
    for(int i = 0; i < mat.getNumberOfRows(); i++){
      for(int j = 0; j < mat.getNumberOfColumns(); j++){
        total += Math.pow(mat.get(i,j).doubleValue(), 2.0);
      }
    }
    return total;
  }

  /* Método público que retorna o resultado desta métrica */
  public double getResult(){ return result; }
}

/* Arquivo MSE.java */
package br.ufsm.lacesm.transform.metrics.transforms;
```

```
import br.ufsm.lacesm.transform.basic.Matrix;
import br.ufsm.lacesm.transform.exceptions.DifferDimensionException;

/* Classe responsável pela implementação da métrica MSE */
public class MSE {

  private Double result;

  /* Método privado responsável pelo cálculo da matriz de
   covariância Rx*/
  private Matrix calculateRx(int matrixOrder){
    double rho = 0.95;
    Matrix rx = new Matrix(matrixOrder,matrixOrder);

    for(int i = 0; i < matrixOrder; i++)
      for(int j = 0; j < matrixOrder; j++)
        rx.set(i,j,Math.pow(rho, Math.abs(i-j)));

    return rx;
  }

  /* Construtor que recebe a matriz da DCT "original" e
  a matriz da aproximação "approx" e inicializa o
  cálculo do MSE */
  public MSE(Matrix original, Matrix approx){
    result = 0.0;
    Matrix expression;

    try {
      Integer matrixOrder = approx.getNumberOfColumns();

      expression = (original.sub(approx))
          .mult(calculateRx(matrixOrder))
          .mult((original.sub(approx)).transposed());

      result = 1.0/original.getNumberOfColumns() * Matrix.trace(expression);

    } catch (DifferDimensionException e) {
      System.out.println(e.getMessage());
    }
  }

  /* Método público que retorna o resultado desta métrica */
  public Double getResult(){ return result; }
}


/* Arquivo TotalErrorEnergy.java */
package br.ufsm.lacesm.transform.metrics.transforms;

import br.ufsm.lacesm.transform.basic.Matrix;

/* Classe responsável pela implementação da métrica Total Error Energy */
public class TotalErrorEnergy {
  private double result;

  /* Construtor que recebe os parametros "dct" e "transform"
  como sendo a matriz de transformação da DCT exata e da
```

```
  aproximação, respectivamente */
  public TotalErrorEnergy(Matrix dct, Matrix transform){
    result = calculate(dct, transform);
  }

  /* Método privado que faz, de fato, o calculo do
  Total Error Energy */
  private double calculate(Matrix dct, Matrix app){
    double finalsum = 0.0, sum = 0.0;
    for(int i = 0 ; i < dct.getNumberOfRows(); i++){
      for(int j = 0 ; j < dct.getNumberOfColumns(); j++){
        sum += Math.pow(
             (dct.get(i,j).doubleValue() -
              app.get(i,j).doubleValue())
           , 2.0);
      }
      finalsum += sum;
      sum = 0.0;
    }
    return Math.PI*finalsum;
  }

  /* Método público que retorna o resultado desta métrica */
  public double getResult(){ return result; }
}


/* Arquivo TransformCodingGain.java */
package br.ufsm.lacesm.transform.metrics.transforms;

import br.ufsm.lacesm.transform.basic.Matrix;
import br.ufsm.lacesm.transform.exceptions.DifferDimensionException;

/* Classe responsável pela implementação da métrica
Transform Coding Gain */
public class TransformCodingGain {

  double result;

  /* Construtor que recebe a matriz de transformação
   "transform" para o cálculo de Transform Coding Gain */
  public TransformCodingGain(Matrix transform) {
    result = calculate(transform);
  }

  /* Método privado que, de fato, faz os calculos do
  Transform Coding Gain */
  private double calculate(Matrix transform) {
    double sum = 0.0;
    double prod = 1.0;

    int N = transform.getNumberOfColumns();
    try {

      Matrix Ry = transform.mult(calculateRx(N))
          .mult(transform.transposed());

      for(int i = 0; i < N; i++){
```

```
      for(int j = 0; j < N; j++){
        sum += Math.pow(
        transform.get(i,j).doubleValue(),
        2.0);
      }
      prod *= Ry.get(i,i).doubleValue()*Math.sqrt(sum);
      sum = 0.0;
    }

    return 10*Math.log10(
    (Matrix.trace(Ry)/((double)(N)))
    /Math.pow(prod, 1.0/(double)(N)));

   } catch (DifferDimensionException e) {
    e.printStackTrace();
   }

   return 0;
  }

  /* Método privado responsável pelo cálculo da
   matriz de covariancia Rx */
  private Matrix calculateRx(int matrixOrder){
   double rho = 0.95;
   Matrix rx = new Matrix(matrixOrder,matrixOrder);

   for(int i = 0; i < matrixOrder; i++)
    for(int j = 0; j < matrixOrder; j++)
     rx.set(i,j,Math.pow(rho, Math.abs(i-j)));

   return rx;
  }

  /* Método público que retorna o resultado desta métrica */
  public double getResult() {
   return result;
  }
}


/* Arquivo TransformEfficiency.java */
package br.ufsm.lacesm.transform.metrics.transforms;

import br.ufsm.lacesm.transform.basic.Matrix;
import br.ufsm.lacesm.transform.exceptions.DifferDimensionException;

/* Classe responsável pela implementação da métrica
 Transform Efficiency */
public class TransformEfficiency {

  double result;

  /* Construtor que recebe a matriz de transformação
  "transform" para o cálculo da Transform Efficiency */
  public TransformEfficiency(Matrix transform) {
   result = 0.0;
   try {
    result = calculate(transform);
   } catch (DifferDimensionException e) {
```

```
      System.out.println(e.getMessage());
    }
  }

  /* Método público que retorna o resultado desta métrica */
  public double getResult() {
    return result;
  }

  /* Método privado responsável pelo cálculo da
  matriz de covariancia Rx */
  private Matrix calculateRx(int matrixOrder){
    double rho = 0.95;
    Matrix rx = new Matrix(matrixOrder,matrixOrder);

    for(int i = 0; i < matrixOrder; i++)
      for(int j = 0; j < matrixOrder; j++)
        rx.set(i,j,Math.pow(rho, Math.abs(i-j)));

    return rx;
  }

   /* Método privado que, de fato, faz os calculos do
   Transform Efficiency */
  private double calculate(Matrix transform) throws
      DifferDimensionException{
    if(transform.getNumberOfColumns() != transform.getNumberOfRows())
      throw new DifferDimensionException();

    double num = 0.0, denom = 0.0;
    Matrix ry = transform.mult(
    calculateRx(transform.getNumberOfColumns())
    );
    ry = ry.mult(transform.transposed());

    for(int i = 0; i < ry.getNumberOfRows(); i++){
      num += Math.abs(ry.get(i,i).doubleValue());
    }

    for(int i = 0; i < ry.getNumberOfRows(); i++){
      for(int j = 0; j < ry.getNumberOfColumns(); j++){
        denom += Math.abs(ry.get(i,j).doubleValue());
      }
    }

    return (num/denom) * 100.0;
  }

}

/* Arquivo BAS_2010.java */
package br.ufsm.lacesm.transform.sixteen;

import br.ufsm.lacesm.transform.basic.Matrix;
import br.ufsm.lacesm.transform.basic.Transform;
import br.ufsm.lacesm.transform.exceptions.DifferDimensionException;

/* Classe responsável pela implementação
```

```
das particularidades de BAS 2010 */
public class BAS_2010 extends Transform {

  /* Construtor padrão */
  public BAS_2010(int dimension) {
    super(dimension);

    Number[] values = new Number[]{
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
      1, 1, 1, 1, 1, 1, 1, 1,-1 ,-1 ,-1 ,-1 ,-1 ,-1 ,-1 ,-1,
      2, 1,-1,-2,-2,-1, 1, 2, 2, 1 ,-1 ,-2 ,-2 ,-1, 1, 2,
      2, 2, 1, 1,-1,-1,-2,-2, 2, 2, 1, 1 ,-1 ,-1 ,-2 ,-2,
      2, 1,-1,-2, 2, 1,-1,-2,-2 ,-1, 1, 2 ,-2 ,-1, 1, 2,
      2,-2,-1, 1,-1, 1, 2,-2, 2 ,-2 ,-1, 1 ,-1, 1, 2 ,-2,
      1, 1, 1, 1,-1,-1,-1,-1,-1 ,-1 ,-1 ,-1, 1, 1, 1, 1,
      1,-1,-1, 1, 1,-1,-1, 1,-1, 1, 1 ,-1 ,-1, 1, 1 ,-1,
      1,-1,-1, 1, 1,-1,-1, 1, 1 ,-1 ,-1, 1, 1 ,-1 ,-1, 1,
      1,-1,-1, 1,-1, 1, 1,-1,-1, 1, 1 ,-1, 1 ,-1 ,-1, 1,
      1,-1, 2,-2, 2,-2, 1,-1, 1 ,-1, 2 ,-2, 2 ,-2, 1 ,-1,
      1, 1,-2,-2, 2, 2,-1,-1, 1, 1 ,-2 ,-2, 2, 2 ,-1 ,-1,
      1, 1,-1,-1,-1,-1, 1, 1,-1 ,-1, 1, 1, 1, 1 ,-1 ,-1,
      1,-1, 1,-1,-1, 1,-1, 1,-1, 1 ,-1, 1, 1 ,-1, 1 ,-1,
      1,-2, 2,-1, 1,-2, 2,-1,-1, 2 ,-2, 1 ,-1, 2 ,-2, 1,
      1,-2, 2,-1,-1, 2,-2, 1, 1 ,-2, 2 ,-1 ,-1, 2 ,-2, 1
    };

    setValues(values);

    Number[] diagonal = new Number[]{
      1.0/(Math.sqrt(16.0)),1.0/(Math.sqrt(16.0)),
      1.0/(Math.sqrt(40.0)), 1.0/(Math.sqrt(40.0)),
      1.0/(Math.sqrt(40.0)), 1.0/(Math.sqrt(40.0)),
      1.0/(Math.sqrt(16.0)), 1.0/(Math.sqrt(16.0)),
      1.0/(Math.sqrt(16.0)), 1.0/(Math.sqrt(16.0)),
      1.0/(Math.sqrt(40.0)), 1.0/(Math.sqrt(40.0)),
      1.0/(Math.sqrt(16.0)), 1.0/(Math.sqrt(16.0)),
      1.0/(Math.sqrt(40.0)), 1.0/(Math.sqrt(40.0)),
    };

    try{
      Matrix diag = Matrix.DiagonalMatrix(dimension, diagonal);
      matrix = diag.mult(matrix);
    }catch(DifferDimensionException e){
      e.printStackTrace();
    }
  }
}

/* Arquivo BAS_2013.java */
package br.ufsm.lacesm.transform.sixteen;

import java.util.ArrayList;

import br.ufsm.lacesm.transform.basic.Matrix;
import br.ufsm.lacesm.transform.basic.Transform;
import br.ufsm.lacesm.transform.eight.SDCT;

/* Classe responsável pela implementação
```

```
das particularidades de BAS 2013 */
public class BAS_2013 extends Transform {

  /* Construtor padrão */
  public BAS_2013(int dimension) {
    super(dimension);
    buildBAS13Matrix(new SDCT(dimension).getMatrix(), new
        WHT(dimension).getUnboundedMatrix());
    matrix = matrix.mult(1/Math.sqrt(dimension));
  }

  /* Método privado que constroi a matriz de BAS 2013 dadas as
   matrizes de transformação da SDCT e da WHT */
  private void buildBAS13Matrix(Matrix sdct, Matrix wht) {
    int curCombination;
    Integer[] combinations = new Integer[sdct.getNumberOfRows()];
    for(int i = 0; i < sdct.getNumberOfRows(); i++){
      curCombination = getEqualsRow(i, sdct, wht);
      combinations[i] = curCombination;
      if(curCombination != -1){
        matrix.setRow(i, wht.getRow(curCombination));
      }
    }
    for(int i = 0; i < combinations.length; i++){
      if(combinations[i] == -1){
        curCombination = getSimilarRow(i, canUse(combinations), sdct, wht);
        combinations[i] = curCombination;
        matrix.setRow(i, wht.getRow(curCombination));
      }
    }
  }

  /* Método privado que retorna qual a linha de "B"
  é mais similar a linha "rowA" de "A" */
  private int getEqualsRow(int rowA, Matrix A, Matrix B){
    int count = 0;
    for(int i = 0; i < B.getNumberOfRows(); i++){
      for(int j = 0; j < A.getNumberOfColumns(); j++){
        if(A.get(rowA,j).doubleValue() == B.get(i, j).doubleValue()){
          count++;
        }
      }
      if(count == A.getNumberOfColumns())
        return i;
      else
        count=0;
    }
    return -1;
  }

  /* Método privado que retorna qual linha de uma
   matriz B similar a de uma matriz A pode ser usada */
  private int getSimilarRow(int rowA, Integer[] canUse, Matrix A, Matrix B){
    int count = 0;
    Integer[] counts = new Integer[canUse.length];
    for(int i = 0; i < canUse.length; i++){
      for(int j = 0; j < B.getNumberOfColumns(); j++){
        if(A.get(rowA, j).doubleValue() == B.get(canUse[i],
```

```
          j).doubleValue()){
        count ++;
       }
      }
      counts[i] = count;
      count = 0;
    }

    int min = 99999;

    for(int i = 0; i < counts.length; i++){
      if(min > counts[i]){
       min = counts[i];
       count = canUse[i];
      }
    }

    return count;
  }

  /* Método privado que guarda quais linhas da matriz da
  WHT já foram utilizadas e retorna quais não foram */
  private Integer[] canUse(Integer[] all){
    int count = 0;
    ArrayList<Integer> useIt = new ArrayList<Integer>();
    for(int i = 0; i < all.length; i++){
      for(int j = 0; j < all.length; j++){
       if(i == all[j]){
        count++;
        break;
       }
      }
      if(count == 0){
       useIt.add(i);
      }
      else{
       count = 0;
      }
    }

    Integer[] resp = new Integer[useIt.size()];

    for(int i = 0; i < resp.length; i++){
      resp[i] = useIt.get(i);
    }

    return resp;
  }
}

/* Arquivo Bayer_2012.java */
package br.ufsm.lacesm.transform.sixteen;

import br.ufsm.lacesm.transform.basic.Matrix;
import br.ufsm.lacesm.transform.basic.Transform;
import br.ufsm.lacesm.transform.exceptions.DifferDimensionException;

/* Classe responsável pela implementação
```

```
das particularidades de Bayer et al 2012 */
public class Bayer_2012 extends Transform {

  /* Construtor padrão */
  public Bayer_2012(int dimension) {
    super(dimension);

    Number[] values = new Number[]{
       1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
          1.0, 1.0, 1.0,
       1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 1.0,-1.0,-1.0,
          0.0,-1.0,-1.0,-1.0,-1.0,-1.0,
       1.0, 1.0, 1.0, 0.0, 0.0,-1.0,-1.0,-1.0,-1.0,-1.0,-1.0, 0.0, 0.0,
          1.0, 1.0, 1.0,
       1.0, 1.0, 1.0, 0.0,-1.0,-1.0,-1.0,-1.0, 1.0, 1.0, 1.0, 1.0,
          0.0,-1.0,-1.0,-1.0,
       1.0, 1.0,-1.0,-1.0,-1.0,-1.0, 1.0, 1.0, 1.0,
          1.0,-1.0,-1.0,-1.0,-1.0, 1.0, 1.0,
       1.0, 1.0,-1.0,-1.0,-1.0, 1.0, 1.0, 0.0, 0.0,-1.0,-1.0, 1.0, 1.0,
          1.0,-1.0,-1.0,
       1.0, 0.0,-1.0,-1.0, 1.0, 1.0, 0.0,-1.0,-1.0, 0.0, 1.0,
          1.0,-1.0,-1.0, 0.0, 1.0,
       1.0, 0.0,-1.0, 1.0, 1.0, 1.0,-1.0,-1.0, 1.0, 1.0,-1.0,-1.0,-1.0,
          1.0, 0.0,-1.0,
       1.0,-1.0,-1.0, 1.0, 1.0,-1.0,-1.0, 1.0, 1.0,-1.0,-1.0, 1.0,
          1.0,-1.0,-1.0, 1.0,
       1.0,-1.0,-1.0, 1.0,-1.0,-1.0, 0.0, 1.0,-1.0, 0.0, 1.0, 1.0,-1.0,
          1.0, 1.0,-1.0,
       1.0,-1.0, 0.0, 1.0,-1.0, 0.0, 1.0,-1.0,-1.0, 1.0, 0.0,-1.0, 1.0,
          0.0,-1.0, 1.0,
       0.0,-1.0, 1.0, 1.0,-1.0, 1.0, 1.0,-1.0, 1.0,-1.0,-1.0,
          1.0,-1.0,-1.0, 1.0, 0.0,
       1.0,-1.0, 1.0,-1.0,-1.0, 1.0,-1.0, 1.0, 1.0,-1.0, 1.0,-1.0,-1.0,
          1.0,-1.0, 1.0,
       1.0,-1.0, 1.0,-1.0, 0.0, 1.0,-1.0, 1.0,-1.0, 1.0,-1.0, 0.0,
          1.0,-1.0, 1.0,-1.0,
       0.0,-1.0, 1.0,-1.0, 1.0,-1.0, 1.0, 0.0, 0.0, 1.0,-1.0, 1.0,-1.0,
          1.0,-1.0, 0.0,
       1.0,-1.0, 0.0,-1.0, 1.0,-1.0, 1.0,-1.0, 1.0,-1.0, 1.0,-1.0, 1.0,
          0.0, 1.0,-1.0
    };

    setValues(values);

    Number[] diagonal = new Number[]{
       1.0/4.0,1.0/(Math.sqrt(14.0)),
       1.0/(2*Math.sqrt(3.0)), 1.0/(Math.sqrt(14.0)),
       1.0/4.0, 1.0/(Math.sqrt(14.0)),
       1.0/(2*Math.sqrt(3.0)), 1.0/(Math.sqrt(14.0)),
       1.0/4.0, 1.0/(Math.sqrt(14.0)),
       1.0/(2*Math.sqrt(3.0)), 1.0/(Math.sqrt(14.0)),
       1.0/4.0, 1.0/(Math.sqrt(14.0)),
       1.0/(2*Math.sqrt(3.0)), 1.0/(Math.sqrt(14.0)),
    };

    try{
      Matrix diag = Matrix.DiagonalMatrix(dimension, diagonal);
      matrix = diag.mult(matrix);
```

```
    }catch(DifferDimensionException e){
      e.printStackTrace();
    }
  }
}

/* Arquivo DCT_II.java */
package br.ufsm.lacesm.transform.sixteen;

import br.ufsm.lacesm.transform.basic.Transform;

/* Classe responsável pela implementação
das particularidades da DCT */
public class DCT_II extends Transform{

  /* Construtor padrão */
  public DCT_II(int dimension) {
    super(dimension);
    setValues();
  }

  /* Método privado que calcula os (i,j)-ésimos elementos
   da DCT */
  private void setValues(){
    for(int row = 0; row < matrix.getNumberOfRows(); row++){
      for(int col = 0; col < matrix.getNumberOfColumns(); col++){
        Double cij = 0.0;
        cij = alpha(row)*Math.cos(
            (Math.PI*row*(2*col+1))/
            (2*matrix.getNumberOfColumns())
            );
        matrix.set(row, col, cij);
      }
    }
  }

  /* Método privado que calcula a função alpha_k
   da formulacao da DCT */
  private Double alpha(Integer index){
    if(index==0)
      return 1.0/(Math.sqrt(matrix.getNumberOfColumns()));
    return (Math.sqrt(2.0/matrix.getNumberOfColumns()));
  }
}

/* Arquivo Proposed_2013.java */
package br.ufsm.lacesm.transform.sixteen;

import br.ufsm.lacesm.transform.basic.Matrix;
import br.ufsm.lacesm.transform.basic.Transform;
import br.ufsm.lacesm.transform.exceptions.DifferDimensionException;

/* Classe responsável pela implementação
das particularidades da transformada proposta */
public class Proposed_2013 extends Transform {

  /* Construtor padrão */
```

```
  public Proposed_2013(int dimension) {
    super(dimension);

    Number[] values = new Number[]{
      1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
      1, 1, 1, 1, 1, 1, 1, 1,-1, -1, -1, -1, -1, -1, -1, -1,
      1, 1, 1, 0, 0,-1,-1,-1,-1, -1, -1, 0, 0, 1, 1, 1,
      1, 1, 0, 0, 0, 0,-1,-1, 1, 1, 0, 0, 0, 0, -1, -1,
      1, 0, 0,-1,-1, 0, 0, 1, 1, 0, 0, -1, -1, 0, 0, 1,
      1, 1,-1,-1,-1,-1, 1, 1,-1, -1, 1, 1, 1, 1, -1, -1,
      1, 0,-1,-1, 1, 1, 0,-1,-1, 0, 1, 1, -1, -1, 0, 1,
      0, 0,-1, 1, 1,-1,-1, 1,-1, 1, 1, -1, -1, 1, 0, 0,
      1,-1,-1, 1, 1,-1,-1, 1, 1, -1, -1, 1, 1, -1, -1, 1,
      1,-1,-1, 1, 0, 0, 1,-1, 1, -1, 0, 0, -1, 1, 1, -1,
      1,-1, 0, 1,-1, 0, 1,-1,-1, 1, 0, -1, 1, 0, -1, 1,
      0, 0, 1, 1,-1,-1, 0, 0, 0, 0, 1, 1, -1, -1, 0, 0,
      0,-1, 1, 0, 0, 1,-1, 0, 0, -1, 1, 0, 0, 1, -1, 0,
      1,-1, 1,-1, 1,-1, 0, 0, 0, 0, 1, -1, 1, -1, 1, -1,
      0,-1, 1,-1, 1,-1, 1, 0, 0, 1, -1, 1, -1, 1, -1, 0,
      1,-1, 0, 0,-1, 1,-1, 1,-1, 1, -1, 1, 0, 0, 1, -1
    };

    setValues(values);

    Number[] diagonal = new Number[]{
      1.0/(Math.sqrt(16.0)),1.0/(Math.sqrt(16.0)),
      1.0/(Math.sqrt(12.0)), 1.0/(Math.sqrt(8.0)),
      1.0/(Math.sqrt(8.0)), 1.0/(Math.sqrt(16.0)),
      1.0/(Math.sqrt(12.0)), 1.0/(Math.sqrt(12.0)),
      1.0/(Math.sqrt(16.0)), 1.0/(Math.sqrt(12.0)),
      1.0/(Math.sqrt(12.0)), 1.0/(Math.sqrt(8.0)),
      1.0/(Math.sqrt(8.0)), 1.0/(Math.sqrt(12.0)),
      1.0/(Math.sqrt(12.0)), 1.0/(Math.sqrt(12.0)),
    };

    try{
      Matrix diag = Matrix.DiagonalMatrix(dimension, diagonal);
      matrix = diag.mult(matrix);
    }catch(DifferDimensionException e){
      e.printStackTrace();
    }
  }
}
```

```
/* Arquivo WHT.java */
package br.ufsm.lacesm.transform.sixteen;

import br.ufsm.lacesm.transform.basic.Matrix;
import br.ufsm.lacesm.transform.basic.Transform;
import br.ufsm.lacesm.transform.exceptions.OutOfBoundsMatrixException;

/* Classe responsável pela implementação
das particularidades de WHT */
public class WHT extends Transform {

  private Matrix unbounded;

  /* Construtor padrão */
```

```
  public WHT(int dimension) {
    super(dimension);
    try {
      Matrix m = new Matrix(2,2);
      m.set(new Number[]{
        1,1,
        1,-1
      });
      int size = m.getNumberOfRows();
      while(size != dimension){
        m = buildMatrix(size*2, m);
        size = m.getNumberOfRows();
      }
      matrix = m;
      unbounded = matrix;
    } catch (OutOfBoundsMatrixException e) {
      System.out.println(e.getMessage());
    }
    matrix = matrix.mult(1.0/Math.sqrt(dimension));
  }

  /* Método público que retorna a matriz de transformação
  sem a multiplicacao por escalar. Este método énecessário
  para o cálculo de BAS 2013 */
  public Matrix getUnboundedMatrix(){
    return unbounded;
  }

  /* Metodo que constroi a matriz de transformacao da WHT de
   ordem "order", dada uma matriz de ordem menor "primitive" */
  private Matrix buildMatrix(int order, Matrix primitive){
    int pi = 0, pj = 0, porder = primitive.getNumberOfColumns();
    Matrix resultant = new Matrix(order, order);
    for(int i = 0; i < order; i ++){
      for(int j = 0; j < order; j++){
        pi = i%porder;
        pj = j%porder;
        if(i >= porder && j >= porder)
          resultant.set(i,j,
          (-1.0)*primitive.get(pi,pj).doubleValue());
        else
          resultant.set(i,j,primitive.get(pi,pj));
      }
    }
    return resultant;
  }
}

 /* Arquivo SDCT.java */
package br.ufsm.lacesm.transform.eight;

import br.ufsm.lacesm.transform.basic.Matrix;
import br.ufsm.lacesm.transform.basic.Transform;
import br.ufsm.lacesm.transform.exceptions.OutOfBoundsMatrixException;

/* Classe responsável pela implementação
das particularidades de SDCT */
public class SDCT extends Transform{
```

```
  /* Construtor padrão */
  public SDCT(int dimension) {
    super(dimension);
    setValues();
  }

  /* Método privado que os (i,j)-ésimos elementos da
  matriz de transformacao da SDCT */
  private void setValues(){
    for(int row = 0; row < matrix.getNumberOfRows(); row++){
      for(int col = 0; col < matrix.getNumberOfColumns(); col++){
        Double cij = 0.0;
        cij = Math.cos(
            (Math.PI*row*(2*col+1))/
            (2*matrix.getNumberOfColumns())
            );
        if(cij < 0)
          cij = -1.0;
        else
          cij = 1.0;
        matrix.set(row, col, cij);
      }
    }
  }

  /* A SDCT não éortogonal e assim o método "inverse"
  deve ser sobrescrito caso se queira utilizá-la.
  Para a definicão da matriz de transformação de
  BAS 2013 não e necessário calcular a inversa da SDCT */

}
```